

Advisory boards aren't only for executives. [Join the LogRocket Content Advisory Board today](#) →



Jul 27, 2023 · 12 min read

Build a REST API with Node.js, Express, and MySQL

Geshan Manandhar



Geshan is a seasoned software engineer with more than a decade of software engineering experience. He has a keen interest in REST architecture, microservices, and cloud computing. He also blogs at geshan.com.np.

Table of contents



Prerequisites

What is MySQL?

Setting up our MySQL database

Adding demo rows for programming languages

What is REST API?

Setting up Express.js for our REST API

REST API project structure

GET popular programming languages

POST a new programming language

PUT to update an existing programming language

DELETE a programming language

Testing our APIs

Advanced example: Using store procedures

Why not just use an ORM?

Conclusion

Editor's note: This tutorial was last updated on 27 July 2023 to provide advanced use cases for the REST API, such as using store procedures instead of inline SQL statements.



Generally, Node.js is coupled with MongoDB and other NoSQL databases, but Node.js also performs well with relational databases like MySQL. If you want to [write a new microservice with Node.js for an existing database](#), you'll likely use [MySQL](#), one of the world's most popular open source databases.

In this tutorial, we'll learn how to build a REST API using MySQL as our database and Node.js as our language. We'll also use the Express.js framework to make our task easier. Our example REST API will track the most popular programming languages.

We'll cover the following:

- [What is MySQL?](#)
- [Setting up our MySQL database](#)
- [What is REST API?](#)
- [Setting up Express.js for our REST API](#)
- [REST API project structure](#)
- [GET popular programming languages](#)

- **POST** a new programming language
- **PUT** to update an existing programming language
- **DELETE** a programming language
- Testing our APIs
- Advanced example: Using store procedures
- Why not just use an ORM?

Prerequisites

To follow along with this article, you should have the following:

- Understanding of how MySQL and relational databases work in general
- Basic knowledge of Node.js and Express.js
- Understanding of what **REST (representational state transfer)** APIs are and how they function
- Knowledge of what CRUD (create, read, update, delete) is and how it relates to the HTTP **GET** , **POST** , **PUT** , and **DELETE** methods

The code in this tutorial is performed on a Mac with Node 14 LTS installed. You can use [Node.js](#), [Docker](#), and [Docker Compose](#) to improve your developer experience. You can also access the full code at the [GitHub repository](#). Now, let's get started!

What is MySQL?

MySQL is one of the most popular databases worldwide. Per the 2022 [Stack Overflow survey](#), MySQL was the most-loved database, with more than 46 percent of respondents using it. The [community edition](#) is available for free, and supported by a large and active community.

MySQL is a feature-packed relational database first released in 1995. It runs on all major operating systems, like Linux, Windows, and macOS. Because of its features

and its cost-effectiveness, MySQL is used by big enterprises and new startups alike. For our example REST API, we'll use a free MySQL service instead of setting up a local MySQL server.

Setting up our MySQL database

To host our testing MySQL 8.0 database, we'll use db4free.net. First, go to the [db4free signup page](https://db4free.net/signup), then fill out the required details by choosing your database name and username:

Signup

By registering for a db4free.net account you agree that:

- db4free.net is a testing environment
- db4free.net is not suitable for production
- if you decide to use your db4free.net database in production despite the warnings, you do that at your own risk (very frequent backups are highly recommended)
- data loss and outages can happen at any time (any complaints about that will likely be ignored)
- the db4free.net team is not granting any warranty or liability of any kind
- the db4free.net team reserves the right to delete databases and/or accounts at any time without notice
- it is up to you to get the latest information from [Twitter](#) and the [db4free.net blog](#)
- db4free.net provides only a MySQL database, but no web space (there is nowhere to upload any files)

Further:

- db4free.net is a service for testing, not for hosting. Databases that store more than 200 MB data will be cleared at irregular intervals without notification
- Please remove data which you no longer need, or [delete your no longer needed account](#). This makes it easier to recover if a server crash occurs.

MySQL database name:	6-16 chars., no upper-case, 1st must be char.
MySQL username:	6-16 chars., no upper-case, 1st must be char.
MySQL user password:	Min. 8 chars.
MySQL user password verification:	Min. 8 chars.

Email address:

Enter your email address
Email addresses of certain domains are not allowed!

☐ I have read the [conditions of use](#) and I agree with them.

Signup

Click on **Signup**, and you should receive a confirmation email. Confirm your account by clicking on the link in the email. Next, on the sidebar, click on **phpMyAdmin**. In the phpMyAdmin login, enter the username and password you chose and click **Go**:

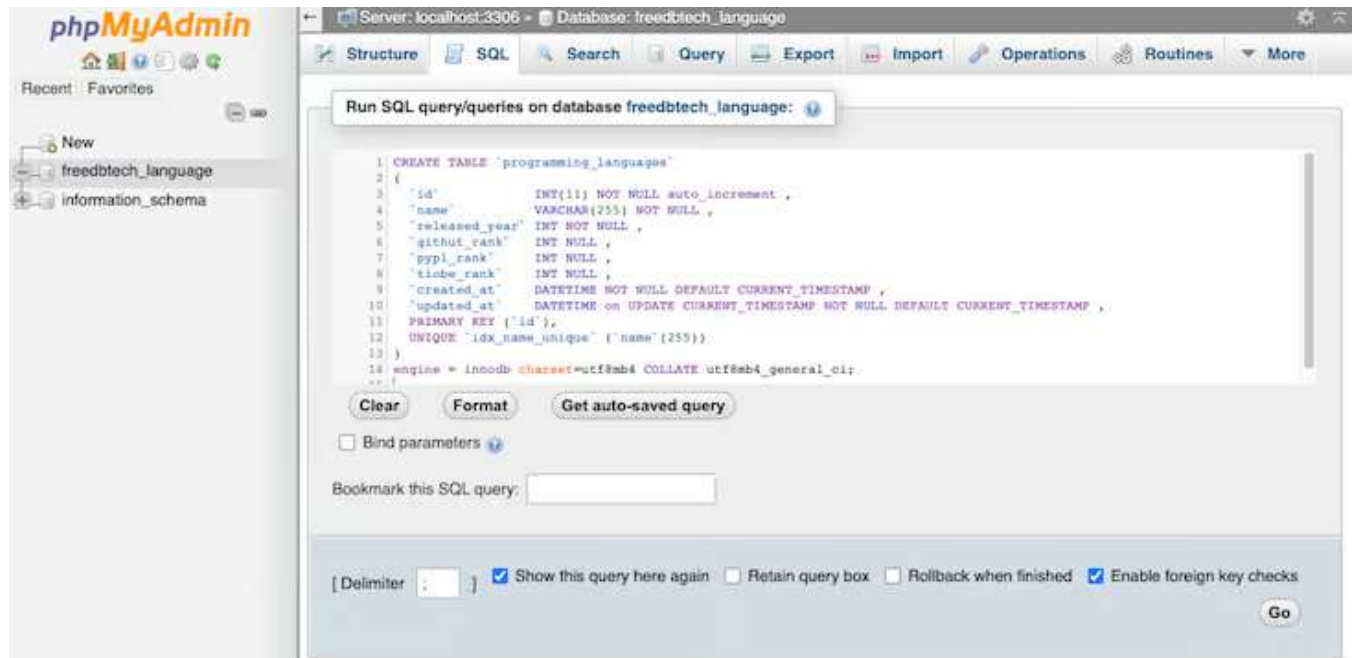


Now, we have an empty database. Let's add the `programming_languages` table. First, click on the database name on the left; for me, it was **restapitest123**. Then, click **SQL** on the top menu, and put the following code for `CREATE TABLE` in the text area:

```
CREATE TABLE `programming_languages`
(
  `id`          INT(11) NOT NULL auto_increment ,
  `name`        VARCHAR(255) NOT NULL ,
  `released_year` INT NOT NULL ,
  `github_rank` INT NULL ,
  `pypi_rank`   INT NULL ,
  `tiobe_rank`  INT NULL ,
  `created_at`  DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ,
  `updated_at`  DATETIME on UPDATE CURRENT_TIMESTAMP NOT NULL DEFA
PRIMARY KEY (`id`),
UNIQUE `idx_name_unique` (`name`(255))
```

```
)  
engine = innodb charset=utf8mb4 COLLATE utf8mb4_general_ci;
```

Click the **Go** button, as shown below:



The code will return with a green check box and a message along the lines of **MySQL** returned an empty result set (i.e. zero rows). With that, we've created a table called **programming_languages** with eight columns and a primary key called **id**, an internet and auto-increment. The **name** column is unique, and we also added **released_year** for the programming language. We have three columns to input the rank of the programming language, sourced from the following resources:

- [GitHub](#): GitHub language stats for Q4 2020
- [PYPL](#): The Popularity of Programming Language Index
- [TIOBE index](#)

The **created_at** and **updated_at** columns store dates to keep track of when the rows were created and updated.

Adding demo rows for programming languages

Next, we'll add 16 popular programming languages to our `programming_languages` table. Click the same **SQL** link on the top of the page and copy and paste the code below:

```
INSERT INTO programming_languages(id,name,released_year,githut_rank,VALUES
(1,'JavaScript',1995,1,3,7),
(2,'Python',1991,2,1,3),
(3,'Java',1995,3,2,2),
(4,'TypeScript',2012,7,10,42),
(5,'C#',2000,9,4,5),
(6,'PHP',1995,8,6,8),
(7,'C++',1985,5,5,4),
(8,'C',1972,10,5,1),
(9,'Ruby',1995,6,15,15),
(10,'R',1993,33,7,9),
(11,'Objective-C',1984,18,8,18),
(12,'Swift',2015,16,9,13),
(13,'Kotlin',2011,15,12,40),
(14,'Go',2009,4,13,14),
(15,'Rust',2010,14,16,26),
(16,'Scala',2004,11,17,34);
```

You should receive a message like “16 rows inserted.” Then, the data from our three sources is collected and added to the table in bulk by the `INSERT` statement, creating 16 rows, one for each programming language. We'll return to this when we fetch data for the `GET` API endpoint.

If we click on the `programming_languages` table, visible on the left, we'll see the rows that we just added:

Server: localhost:3306 - Database: freedbtech_language - Table: programming_languages

Showing rows 0 - 15 (16 total, Query took 0.0005 seconds.)

`SELECT * FROM `programming_languages``

Options: ☐ Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key: None

	id	name	released_year	github_rank	pypi_rank	tiobe_rank	created_at	updated_at
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	1	JavaScript	1995	1	3	7	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	2	Python	1991	2	1	3	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	3	Java	1995	3	2	2	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	4	TypeScript	2012	7	10	42	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	5	C#	2000	9	4	5	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	6	PHP	1995	8	6	8	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	7	C++	1985	5	5	4	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	8	C	1972	10	5	1	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	9	Ruby	1995	6	15	15	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	10	R	1993	33	7	9	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	11	Objective-C	1984	18	8	18	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	12	Swift	2015	16	9	13	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	13	Kotlin	2011	15	12	40	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	14	Go	2009	4	13	14	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	15	Rust	2010	14	16	26	2021-02-01 10:09:50	2021-02-01 10:09:50
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	16	Scala	2004	11	17	34	2021-02-01 10:09:50	2021-02-01 10:09:50

Next, we'll set up Express.js for our REST API with Node.js and MySQL.

What is REST API?

REST API, short for Representational State Transfer API, is a popular architectural style for designing web services and APIs. REST API allows communication between the client and server through standard HTTP methods like GET, POST, PUT, and DELETE, using the principles of statelessness and resource-based interactions.

Some of the important guidelines of REST API include:

- Client-server architecture: REST API is divided into client and server components, allowing them to evolve independently
- Statelessness: Client requests contain all the necessary information to understand and fulfill the request. The server doesn't store any client state between requests
- Cacheability: REST API can utilize caching mechanisms to improve performance and reduce the load on the server

- Uniform interface: REST API has a consistent and uniform interface, including using standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources

Setting up Express.js for our REST API

To set up a Node.js app with an Express.js server, we'll first create a directory for our project to reside in:

```
mkdir programming-languages-api && cd programming-languages-api
```

Then, we can create a `package.json` file with `npm init -y` as follows:

```
{
  "name": "programming-languages-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

To install Express, we'll run `npm i express`, adding Express as a dependency in the `package.json` file. Next, we'll create a slim server in the `index.js` file. It will print an `ok` message on the main path `/`:

```
const express = require("express");
const app = express();
const port = 3000;
```

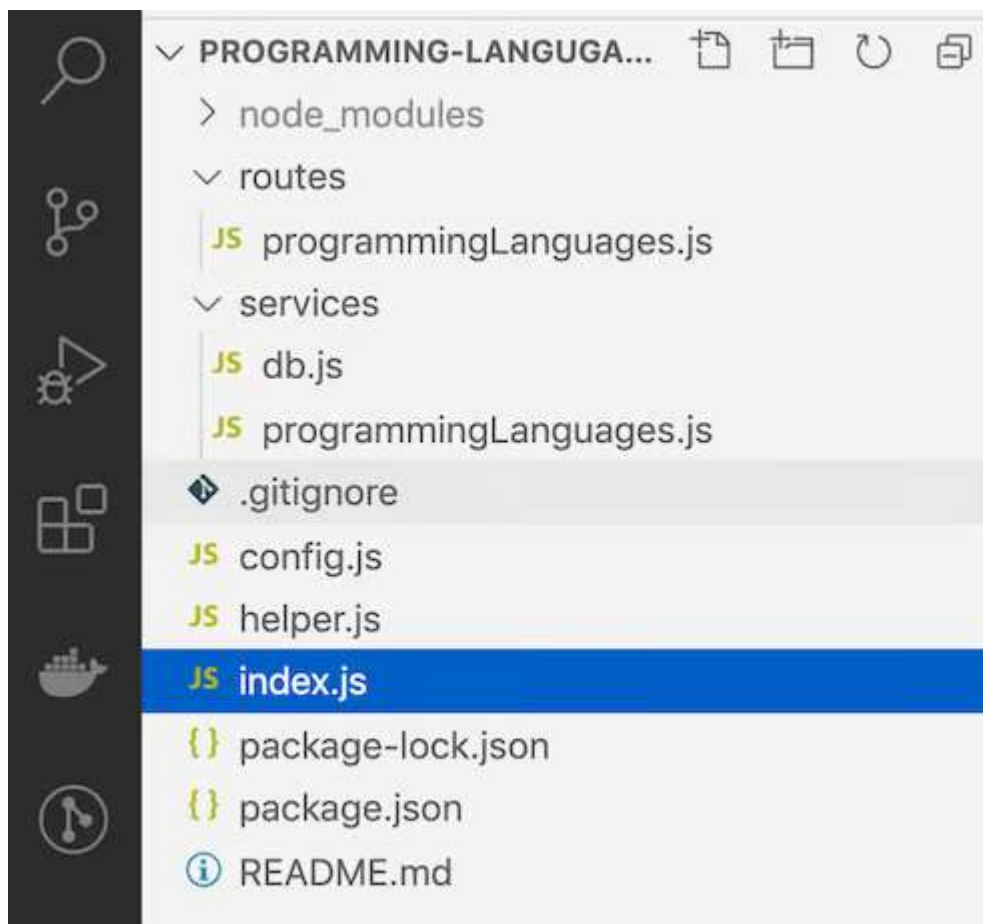
```
app.use(express.json());
app.use(
  express.urlencoded({
    extended: true,
  })
);
app.get("/", (req, res) => {
  res.json({ message: "ok" });
});
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

There are a few important things to note in the code above. For one, we'll use the built-in Express JSON parser middleware to parse JSON in the next steps. We'll also use the `express.urlencoded()` middleware to parse the URL encoded body.

If `PORT` is not provided as an environment variable, our app will run on port number 3000. We can run the server with `node index.js` and hit `http://localhost:3000` to see `{message: "ok"}` as the output.

REST API project structure

We'll structure our project in the following manner to arrange our files logically in folders:



`config.js` will contain configuration for information like the database credentials and the rows we want to show per page when we paginate results. `helper.js` is the home for any helper functions, like calculating offset for pagination.

The `routes/programmingLanguages.js` file will be the glue between the URI and the corresponding function in the `services/programmingLanguages.js` service. The `services` folder will house all our services. One of them is `db.js`, which we use to talk with the MySQL database.

Another service is `programmingLanguages.js`, which will have methods like `getMultiple`, `create`, etc., to get and create the programming language resource. Basic mapping of the URI and the related service function will look like the code below:

```
GET /programming-languages → getMultiple()
POST /programming-languages → create()
```

```
PUT /programming-languages/:id → update()  
DELETE /programming-languages/:id → remove()
```

Now, let's code our `GET` programming languages API with pagination.



GET popular programming languages

We will need to link our Node.js server with MySQL to create our `GET` programming languages API. We'll use the `mysql2` package to interact with the MySQL database.

First, we need to install `mysql2` using the command below at the project root directory:

```
npm i mysql2
```

Next, we'll create the `config` file on the root of the project with the following contents:

```
const config = {
  db: {
    /* don't expose password or any sensitive info, done only for de
    host: "db4free.net",
    user: "restapitest123",
    password: "restapitest123",
    database: "restapitest123",
    connectTimeout: 60000
  },
  listPerPage: 10,
};
module.exports = config;
```

It is worth noting that we set the `connectTimeout` to 60 seconds. The default is ten seconds, which may not be enough. Consequently, we'll create the `helper.js` file with the code below:

```
function getOffset(currentPage = 1, listPerPage) {
  return (currentPage - 1) * [listPerPage];
}

function emptyOrRows(rows) {
  if (!rows) {
    return [];
  }
  return rows;
}

module.exports = {
```

```
    getOffset,  
    emptyOrRows  
  }  
}
```

For the fun part, we'll add the route and link it to the services. First, we'll connect to the database and enable running queries on the database in the `services/db.js` file:

```
const mysql = require('mysql2/promise');  
const config = require('../config');  
  
async function query(sql, params) {  
  const connection = await mysql.createConnection(config.db);  
  const [results, ] = await connection.execute(sql, params);  
  
  return results;  
}  
  
module.exports = {  
  query  
}
```

Now, we'll write up the `services/programmingLanguages.js` file that acts as the bridge between the route and the database:

```
const db = require('./db');  
const helper = require('../helper');  
const config = require('../config');  
  
async function getMultiple(page = 1){  
  const offset = helper.getOffset(page, config.listPerPage);  
  const rows = await db.query(  
    `SELECT id, name, released_year, github_rank, pypl_rank, tiobe_r  
    FROM programming_languages LIMIT ${offset},${config.listPerPage}  
  );
```

```
const data = helper.emptyOrRows(rows);
const meta = {page};

return {
  data,
  meta
}

module.exports = {
  getMultiple
}
```

After that, we'll create the `routes` file in `routes/programmingLanguages.js`, which looks like the following:

```
const express = require('express');
const router = express.Router();
const programmingLanguages = require('../services/programmingLanguages');

/* GET programming languages. */
router.get('/', async function(req, res, next) {
  try {
    res.json(await programmingLanguages.getMultiple(req.query.page))
  } catch (err) {
    console.error(`Error while getting programming languages `, err);
    next(err);
  }
});

module.exports = router;
```

For the final piece of our `GET` endpoint, we need to wire up the route in the `index.js` file as follows:


```
const express = require("express");
const app = express();
const port = 3000;
const programmingLanguagesRouter = require("./routes/programmingLang
app.use(express.json());
app.use(
  express.urlencoded({
    extended: true,
  })
);
app.get("/", (req, res) => {
  res.json({ message: "ok" });
});
app.use("/programming-languages", programmingLanguagesRouter);
/* Error handler middleware */
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  console.error(err.message, err.stack);
  res.status(statusCode).json({ message: err.message });
  return;
});
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

We made two important changes in our entry point `index.js` file. First, we added the code below:

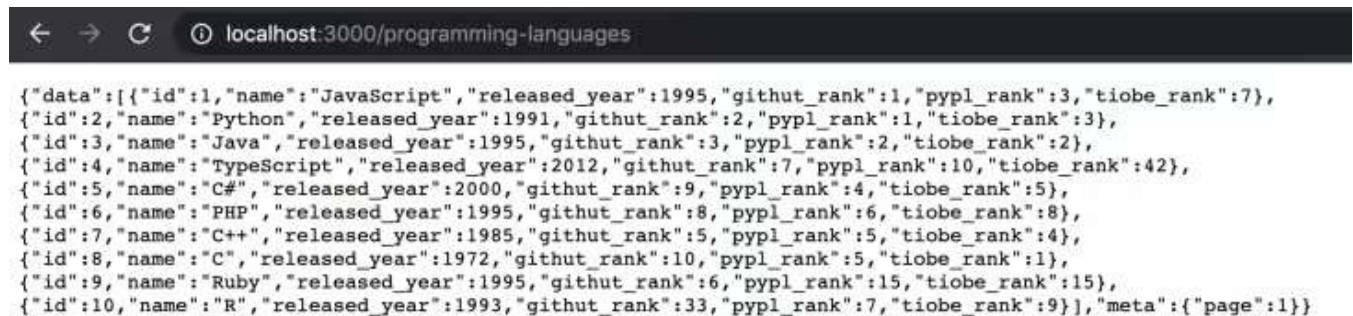
```
const programmingLanguagesRouter = require('./routes/programmingLang
```

Then, we linked up the `/programming-languages` route to the router we just created:

```
app.use('/programming-languages', programmingLanguagesRouter);
```

We also added an error handler middleware to handle errors and provide a proper status code and message. After adding the `GET` endpoint, when we run our app again with `node index.js` and hit the browser with

`http://localhost:3000/programming-languages`, we'll see an output like this:



The screenshot shows a web browser at the URL `localhost:3000/programming-languages`. The response is a JSON array of 10 programming languages, each with an id, name, released_year, github_rank, pypl_rank, and tiobe_rank. The languages are: JavaScript, Python, Java, TypeScript, C#, PHP, C++, C, Ruby, and R. The response also includes a meta object with a page number of 1.

```
{
  "data": [
    {
      "id": 1,
      "name": "JavaScript",
      "released_year": 1995,
      "github_rank": 1,
      "pypl_rank": 3,
      "tiobe_rank": 7
    },
    {
      "id": 2,
      "name": "Python",
      "released_year": 1991,
      "github_rank": 2,
      "pypl_rank": 1,
      "tiobe_rank": 3
    },
    {
      "id": 3,
      "name": "Java",
      "released_year": 1995,
      "github_rank": 3,
      "pypl_rank": 2,
      "tiobe_rank": 2
    },
    {
      "id": 4,
      "name": "TypeScript",
      "released_year": 2012,
      "github_rank": 7,
      "pypl_rank": 10,
      "tiobe_rank": 42
    },
    {
      "id": 5,
      "name": "C#",
      "released_year": 2000,
      "github_rank": 9,
      "pypl_rank": 4,
      "tiobe_rank": 5
    },
    {
      "id": 6,
      "name": "PHP",
      "released_year": 1995,
      "github_rank": 8,
      "pypl_rank": 6,
      "tiobe_rank": 8
    },
    {
      "id": 7,
      "name": "C++",
      "released_year": 1985,
      "github_rank": 5,
      "pypl_rank": 5,
      "tiobe_rank": 4
    },
    {
      "id": 8,
      "name": "C",
      "released_year": 1972,
      "github_rank": 10,
      "pypl_rank": 5,
      "tiobe_rank": 1
    },
    {
      "id": 9,
      "name": "Ruby",
      "released_year": 1995,
      "github_rank": 6,
      "pypl_rank": 15,
      "tiobe_rank": 15
    },
    {
      "id": 10,
      "name": "R",
      "released_year": 1993,
      "github_rank": 33,
      "pypl_rank": 7,
      "tiobe_rank": 9
    }
  ],
  "meta": {
    "page": 1
  }
}
```

Depending on the extensions you have installed on your browser, your output might look a bit different. Note that we've already implemented pagination for our `GET` API, which is possible because of the `getOffset` function in `helper.js` and how we run the `SELECT` query in `services/programmingLanguage.js`. Try `http://localhost:3000/programming-languages?page=2` to see languages 11–16.

POST a new programming language

Our `POST` API will allow us to create a new programming language in our table. To create a `POST` programming language API in the `/programming-languages` endpoint, we'll add code to the `service` and the `routes` files. In the service method, we'll get the name, the release year, and other ranks from the request body, then insert them into the `programming_languages` table.

Append the following code to the `services/programmingLanguages.js` file:

```
async function create(programmingLanguage){
  const result = await db.query(
    `INSERT INTO programming_languages
    (name, released_year, github_rank, pypl_rank, tiobe_rank)
    VALUES
    ('${programmingLanguage.name}', ${programmingLanguage.released_y
  );

  let message = 'Error in creating programming language';

  if (result.affectedRows) {
    message = 'Programming language created successfully';
  }

  return {message};
}
```

Make sure you also export the following function:

```
module.exports = {
  getMultiple,
  create
}
```

For the function above to be accessible, we need to add a route to link it up in the `routes/programmingLanguages.js` file:

```
/* POST programming language */
router.post('/', async function(req, res, next) {
  try {
    res.json(await programmingLanguages.create(req.body));
  } catch (err) {
    console.error(`Error while creating programming language`, err.m
    next(err);
  }
}
```

```
}  
});
```

PUT to update an existing programming language

We'll use the `/programming-languages/:id` endpoint to update an existing programming language, where we'll get the data to update the language. To update a programming language, we'll run the `UPDATE` query based on the data we got in the request.

`PUT` is an idempotent action, meaning that if the same call is made over and over again, it will produce the same results. To enable updating existing records, we'll add the following code to the programming language service:

```
async function update(id, programmingLanguage){  
  const result = await db.query(  
    `UPDATE programming_languages  
    SET name="${programmingLanguage.name}", released_year=${programmingLanguage.released_year},  
    pypl_rank=${programmingLanguage.pypl_rank}, tiobe_rank=${programmingLanguage.tiobe_rank}  
    WHERE id=${id}`  
  );  
  
  let message = 'Error in updating programming language';  
  
  if (result.affectedRows) {  
    message = 'Programming language updated successfully';  
  }  
  
  return {message};  
}
```

Make sure you also export this function, as we did before:

```
module.exports = {  
  getMultiple,  
  create,  
  update,  
};
```

To wire up the code with the `PUT` endpoint, we'll add the code below to the programming languages route file, just above `module.exports = router;`:

```
/* PUT programming language */  
router.put('/:id', async function(req, res, next) {  
  try {  
    res.json(await programmingLanguages.update(req.params.id, req.body));  
  } catch (err) {  
    console.error(`Error while updating programming language`, err.message);  
    next(err);  
  }  
});
```

Now, we can update any existing programming language. For instance, we can update a language's name if we see a typo.

DELETE a programming language

We'll use the `/programming-languages/:id` path with the HTTP `DELETE` method to add the functionality to delete a programming language. Go ahead and run the code below:

```
async function remove(id){  
  const result = await db.query(  
    `DELETE FROM programming_languages WHERE id=${id}`  
  );  
}
```

```
);

let message = 'Error in deleting programming language';

if (result.affectedRows) {
  message = 'Programming language deleted successfully';
}

return {message};
}
```

Don't forget to export this function as well. Once again, to link up the service with the route, we'll add the following code to the `routes/programmingLanguages.js` file:

```
/* DELETE programming language */
router.delete('/:id', async function(req, res, next) {
  try {
    res.json(await programmingLanguages.remove(req.params.id));
  } catch (err) {
    console.error(`Error while deleting programming language`, err.m
    next(err);
  }
});
```

Testing our APIs

After running the Node.js Express server with `node index.js`, you can test all the API endpoints. To create a new programming language, let's go with Dart, and run the following cURL command. Alternatively, you can use Postman or any other HTTP client:

```
curl -i -X POST -H 'Accept: application/json' \
  -H 'Content-type: application/json' http://localhost:3000/progra
  --data '{"name":"dart", "released_year": 2011, "github_rank": 13
```

The code above will result in the following output:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 55
ETag: W/"37-3mETlnRrtfrms6w1AjdGAXKq9GE"
Date: Mon, 01 Feb 2021 11:20:07 GMT
Connection: keep-alive

{"message":"Programming language created successfully"}
```

You can remove the `X-Powered-By` header and add other security response headers using [Express.js Helmet](#), which will greatly improving the API's security. For now, let's update the GitHub rank of `Dart` from 13 to 12:

```
curl -i -X PUT -H 'Accept: application/json' \
  -H 'Content-type: application/json' http://localhost:3000/progra
  --data '{"name":"dart", "released_year": 2011, "github_rank": 12
```

The code above will generate an output like the one below:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 55
ETag: W/"37-0QPAQsRHsm23S9CNV3rPa+AFuXo"
Date: Mon, 01 Feb 2021 11:40:03 GMT
Connection: keep-alive
```

```
{"message": "Programming language updated successfully"}
```

To test out the `DELETE` API, you can use the following cURL to delete Dart with `ID 17`:

```
curl -i -X DELETE -H 'Accept: application/json' \
  -H 'Content-type: application/json' http://localhost:3000/progra
```

The code above will result in the following output:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 55
ETag: W/"37-aMzd+8NpWQ09igvHbNLorsXxGFo"
Date: Mon, 01 Feb 2021 11:50:17 GMT
Connection: keep-alive

{"message": "Programming language deleted successfully"}
```

If you're more used to a visual interface for testing, such as Postman, you can [import the cURL commands](#) into Postman.

In this tutorial, we kept our example fairly simple. However, if this were a real-life API and not a demo, I'd highly recommend the following:

- Use a robust validation library like [Joi](#) to validate the input precisely, for example, to ensure the programming language's name is required. It doesn't already exist in the database
- Improve security by adding [Helmet.js](#) to [Express.js](#)
- Streamline logs in a more manageable way using a [Node.js logging library](#) like [Winston](#)

- Use Docker for the Node.js application

Advanced example: Using store procedures

In this tutorial, we use inline SQL statements for simplicity. We should consider using store procedures in real-world projects. Using store procedures instead of inline SQL has several advantages: improved performance, easier maintainability, and, more importantly, better security.

Let's add a new route `GET /programming-languages/:id` in our app to use a store procedure. First, we need to create it in the database. Run the following SQL script to create a store procedure for searching the programming language by id:

```
DELIMITER $$  
CREATE PROCEDURE `sp_search_programming_languages_by_id` (in langid int)  
BEGIN  
    SELECT name, github_rank, pypl_rank, tiobe_rank, created_at  
    FROM programming_languages  
    where id = langid;  
END $$
```

To use the newly created store procedure, we need to add this setting to the `config.js`:

```
const config = {  
  db: {  
    ...  
    multipleStatements: true  
  },  
}
```

Then, we can add a new helper function to `db.js`:

```
async function callSpSearch(id) {  
  const connection = await mysql.createConnection(config.db);  
  const [results, ] = await connection.query('CALL sp_search_progr  
  
  return results;  
}
```

The new function needs to be exported as well:

```
module.exports = {  
  query,  
  callSpSearch  
}
```

Next, add this function to the `services/programmingLanguages.js`, and add it to the exports:

```
async function search(id){  
  const rows = await db.callSpSearch(id);  
  const data = helper.emptyOrRows(rows);  
  return {  
    data  
  }  
}  
  
module.exports = {  
  getMultiple,  
  create,  
  update,  
  remove,  
  search  
}
```

The last step is to add a new route into `routes/programmingLanguages.js`:

```
router.get('/:id', async function(req, res, next) {  
  try {  
    res.json(await programmingLanguages.search(req.params.id));  
  } catch (err) {  
    console.error(`Error while searching programming languages `,  
      next(err);  
    }  
  }  
});
```

That's it! Now, we can restart the server and give it a go. Enter the browser with `http://localhost:3000/programming-languages/1` , and we should see an output similar to the one below:



```
{  
  "data": [{  
    "name": "JavaScript",  
    "github_rank": 1,  
    "pypi_rank": 3,  
    "tiobe_rank": 7,  
    "created_at": "2023-07-11T21:19:17.000Z"  
  }],  
  "fieldCount": 0,  
  "affectedRows": 0,  
  "insertId": 0,  
  "info": "",  
  "serverStatus": 16386,  
  "warningStatus": 0,  
  "stateChanges": {  
    "systemVariables": {},  
    "schema": "restblog",  
    "gtids": []  
  },  
  "trackStateChange": null,  
  "changedRows": 0  
}
```

Check out the full source code in [this CodeSandbox editor](#).

Why not just use an ORM?

An ORM (Object-Relational Mapping) is a library connecting object-oriented code with relational databases. It lets developers interact with the database using programming language concepts instead of writing raw SQL queries.

While using an ORM has its advantages, it also has limitations and disadvantages compared to the vanilla approach:

- ORMs add an additional layer of abstraction between the app and the database. It could lead to suboptimal performance in some use cases
- Using an ORM can introduce significant complexity to the app and may need a fair bit of a learning curve for developers to understand the framework and its conventions. For a small project, those overheads might outweigh the benefits

- ORMs are designed to be database agnostic, providing a consistent interface across several different databases. This abstraction can lead to limitations to certain database-specific features. If our app depends on some database-specific features unsupported by the ORM, the vanilla approach will be a better option

In summary, whether to use an ORM depends on many factors, including application requirements, performance considerations, project size, complexity, and team skillset.

Conclusion

We now have a functioning API server that uses Node.js and MySQL. This tutorial taught us how to set up MySQL as a free service. We then created an Express.js server that can handle various HTTP methods concerning how it translates to SQL queries.

The example REST API in this tutorial is a good starting point and foundation for building real-world, production-ready REST APIs where you can practice the additional considerations described above. I hope you enjoyed this article. Happy coding!

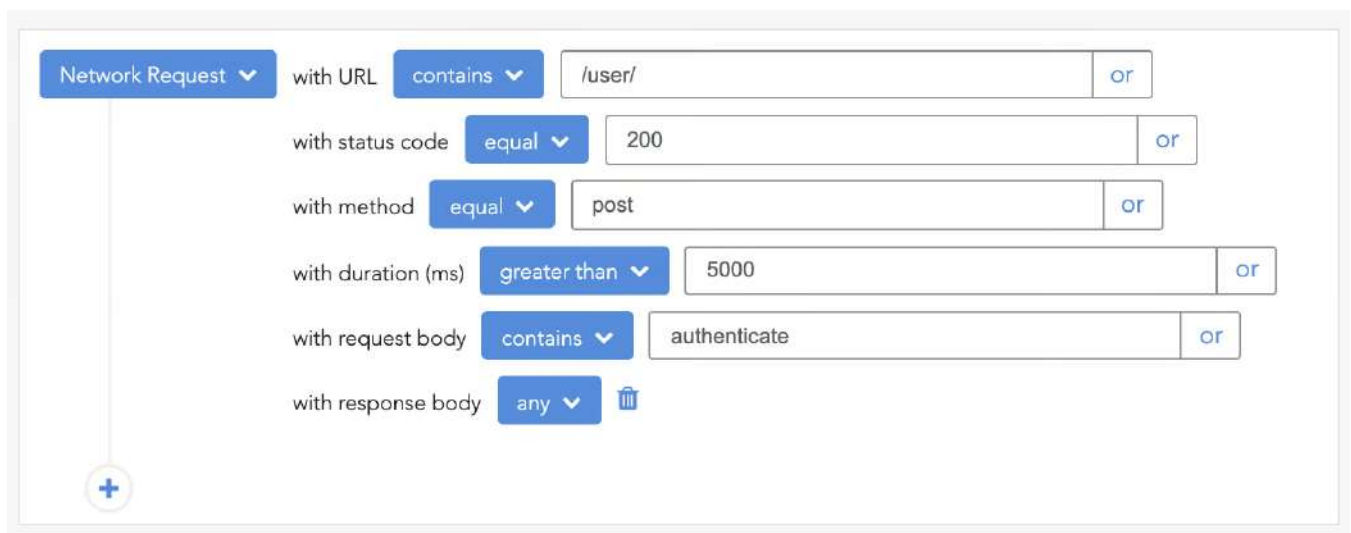
More great articles from LogRocket:

- Don't miss a moment with [The Replay](#), a curated newsletter from LogRocket
- [Learn](#) how LogRocket's Galileo cuts through the noise to proactively resolve issues in your app
- Use React's `useEffect` [to optimize your application's performance](#)
- Switch between [multiple versions of Node](#)
- [Discover](#) how to use the React children prop with TypeScript
- [Explore](#) creating a custom mouse cursor with CSS
- Advisory boards aren't just for executives. [Join LogRocket's Content Advisory Board](#). You'll help inform the type of content we create and get access to

exclusive meetups, social accreditation, and swag.

200s only ✓ Monitor failed and slow network requests in production

Deploying a Node-based web app or website is the easy part. Making sure your Node instance continues to serve resources to your app is where things get tougher. If you're interested in ensuring requests to the backend or third-party services are successful, [try LogRocket](#).



The screenshot shows the LogRocket network request filter interface. It features a series of filter rules stacked vertically, each with a dropdown menu for the filter type, a dropdown for the operator, and a text input for the value. The filters are connected by 'or' buttons. The first filter is 'Network Request' with the operator 'contains' and the value '/user/'. The second filter is 'with status code' with the operator 'equal' and the value '200'. The third filter is 'with method' with the operator 'equal' and the value 'post'. The fourth filter is 'with duration (ms)' with the operator 'greater than' and the value '5000'. The fifth filter is 'with request body' with the operator 'contains' and the value 'authenticate'. The sixth filter is 'with response body' with the operator 'any' and a trash icon. A plus sign button is at the bottom left.

Filter Type	Operator	Value	Connector
Network Request	contains	/user/	or
with status code	equal	200	or
with method	equal	post	or
with duration (ms)	greater than	5000	or
with request body	contains	authenticate	or
with response body	any		

[LogRocket](#) is like a DVR for web and mobile apps, recording literally everything that happens while a user interacts with your app. Instead of guessing why problems happen, you can aggregate and report on problematic network requests to quickly understand the root cause.

LogRocket instruments your app to record baseline performance timings such as page load time, time to first byte, slow network requests, and also logs Redux, NgRx, and Vuex actions/state. [Start monitoring for free](#).

Share this:

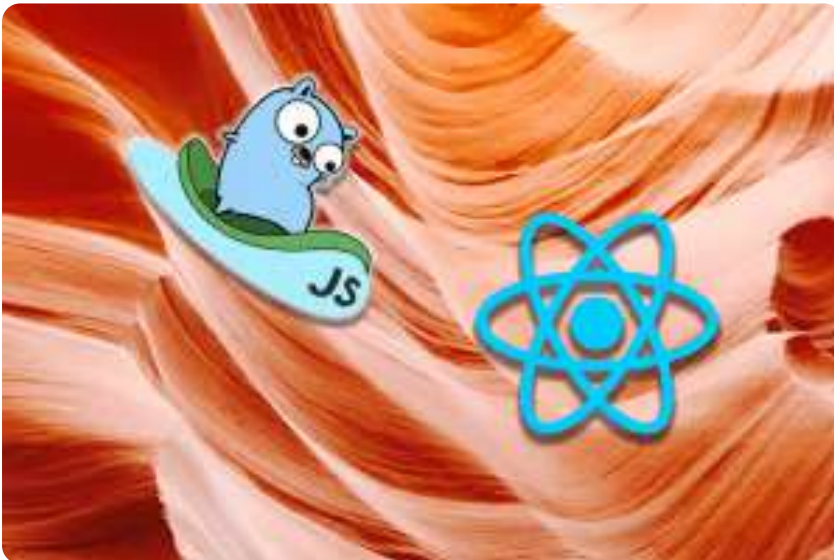


#node

Stop guessing about your digital experience with LogRocket

Get started for free

Recent posts:



Build a full-stack app with React and Goxygen

We show how to use Goxygen to scaffold a full-stack React app. See how to integrate React with Go and modify Goxygen to suit your project requirements.

Clara Ekekenta



Dec 6, 2023 · 8 min read



Express.js adoption guide: Overview, examples, and alternatives

Express.js is a Node.js framework for creating maintainable and fast backend web applications in JavaScript. In the fast-paced world of [...]

**Antonello Zanini**

Dec 6, 2023 · 17 min read



Nesting web components in vanilla JavaScript

Web components are underrated for the performance and ergonomic benefits they provide in vanilla JS. Learn how to nest them in this post.

**Mark Conroy**

Dec 5, 2023 · 10 min read



Using defer in Angular 17 to implement lazy loading

Angular's new `defer` feature, introduced in Angular 17, can help us optimize the delivery of our apps to end users.

**Lewis Cianci**

Dec 4, 2023 · 10 min read

[View all posts](#)

19 Replies to "Build a REST API with Node.js, Express, and MySQL"

**pedro** says:

February 24, 2021 at 5:50 am

[Reply](#) ↩

thank you good tutorial !!!



Geshan Manandhar says:

Reply ↩

February 26, 2021 at 5:52 pm

You are welcome. I hope it helped you.



B M Rafiul Alam says:

Reply ↩

March 13, 2021 at 12:06 am

Simple and easiest. Thank you very much for your nice tutorial.



Geshan Manandhar says:

Reply ↩

March 20, 2021 at 7:09 am

Thanks for the positive comment!



thisisramann says:

Reply ↩

April 9, 2021 at 5:48 am

"INSERT INTO newsession (req_datettime) VALUES (" +newSession.req_datettime +")");
gives undefined value, can you please guide



thisisramann says:

Reply ↩

April 9, 2021 at 6:42 am

```
async function create(newwSession){  
  const result = await db.query(  
    `INSERT INTO newsession  
    (req_datettime )  
    VALUES  
    (? )`,  
    [  
      newwSession.req_datettime
```

]

);

can you please guide what i am missing i have created a new router and its says “message”:
“Bind parameters must not contain undefined. To pass SQL NULL specify JS null”

**Christopher Moreno** says:

Reply ↩

July 8, 2021 at 6:36 am

Why am I getting this error?

Error: Incorrect arguments to mysqld_stmt_execute

**Johnny R.** says:

Reply ↩

September 8, 2021 at 10:00 pm

Found out that with MySQL 8.0.22+ the args need to be passed as a string. Changing the line in programmingLanguages.js from:

```
`SELECT id, name, released_year, github_rank, pypl_rank, tiobe_rank
```

```
FROM programming_languages LIMIT ?,?`,
```

```
[offset, config.listPerPage]
```

to:

```
`SELECT id, name, released_year, github_rank, pypl_rank, tiobe_rank
```

```
FROM programming_languages LIMIT ?,?`,
```

```
[offset + "", config.listPerPage + ""]
```

Corrected this error.

**Samson** says:

Reply ↩

February 2, 2022 at 2:14 pm

THANK YOU!! this was killing me

**amirah** says:

Reply ↩

October 13, 2021 at 6:35 am

very nice! save my hours

thanks so much, I hope you have a nice day sir 😊



Geshan Manandhar says:

Reply ↩

October 13, 2021 at 3:43 pm

Great to know it helped you.



Yogendra Kumar Sonwani says:

Reply ↩

October 31, 2021 at 2:26 am

Hi, I am calling the API using axios in my react project but its showing 500 internal server error, I can't figure why. Please Help



Difi says:

Reply ↩

November 4, 2021 at 6:46 pm

Hello, Thank you so much for this perfect Tutorial ! I just have a question on get parts. I make a query to have some post with comments. But in the result I have one object for each comments (for all informations post +comment) . I would like to have the result like a tree. So 1 object per post which includes the 1 list of comments. Do you know how can I do ?



Fahad Qureshi says:

Reply ↩

February 14, 2022 at 4:54 am

One of the amazing way to deliver the knowledge, keep rocking. Thank you



Rahul says:

Reply ↩

February 18, 2022 at 5:10 am

For POST create function. I am getting error as "Cannot read properties of undefined (reading 'id')"



john says:

Reply ↩

September 15, 2022 at 1:53 pm

Hello, I'm getting the same issue, do you have any tips to solve this problem ?



Jacob says:

Reply ↩

September 29, 2022 at 4:34 pm

When you are trying to do a POST request then the parameters that are strings need to be wrapped with "".

So `${programmingLanguage.name}` should be `"${programmingLanguage.name}"` 😊



Jauffray says:

Reply ↩

March 22, 2023 at 3:12 pm

Thank you, I was getting an error while doing the POST request because of that: "Unknown column 'dart' in 'field list'". If anyone else encounters the same error, this is the reason why.



Mario says:

Reply ↩

August 14, 2023 at 3:08 pm

Thank you for the tutorial. When creating the connection to the db, you commented "don't expose password or any sensitive info". Does that mean that there is another way to set the password for this connection? Could you please share how else to do it? Thanks very much.

Leave a Reply