

Be part of a better internet. [Get 20% off membership for a limited time](#)

Build a Caching Layer in Node.js With Redis



Semaphore · Follow

15 min read · Feb 13, 2024



463



6



Your Node.js backend is slow? Speed it up by caching responses and start returning them immediately. This will also drastically increase the number of users who can simultaneously access your site. The problem is that the basic approach to caching in Node.js involves spreading read and write caching operations across the business logic. This leads to code duplication and makes your project more difficult to maintain. A better solution is a Node.js caching layer!

By using Redis as a cache and encapsulating all caching logic in a middleware function, you have control over which routes to apply caching behavior to. This way you get the benefits of caching without any downside to your codebase.

In this article, you will discover why you need this layer in your Node.js architecture and see how to implement it with Redis in a step-by-step tutorial.

Take the performance of your Express application to the next level!

Benefits of Using Redis in Node.js for Caching

Redis is an open-source in-memory data repository that serves as a key-value database. It provides extreme performance because it stores data in memory instead of on disk like other database technologies. In the context of a backend application, Redis is an excellent choice for implementing a caching system. Its speed and efficiency are critical for caching the most requested endpoint responses to reduce the server response times.

Caching with Redis in Node.js involves writing the responses produced by the servers to the in-memory database. This is great for both an API backend and a Node.js application that serves HTML pages. In the latter case, the key could be `<URL+parameters>` while the corresponding value is the associated HTML page. Subsequent requests referring to that “key” will be able to access the corresponding HTML page directly from Redis, skipping the execution of the same business logic again and again. By using Redis as a cache, Node.js can offload repetitive database queries and CPU calculations, saving considerable time and resources. The result is a more responsive application as a whole.

What Is a Caching Layer in Node.js and Why It Should Be a Middleware

In Node.js, a caching layer is the part of the backend application that contains the logic to implement response caching logic. This code must rely

on a caching provider, such as Redis. Now, the problem is that Redis works in memory, and RAM is expensive.

Caching all routes is not the best approach because it would quickly fill up the available RAM. The ideal scenario is that you could choose the routes to enable caching for. That can easily be achieved through a Node.js Redis caching middleware, which will encapsulate the logic necessary to:

1. Write to Redis the response produced by the server for the selected route.
2. Read it when a new request arrives for the same endpoint.

Once defined, you could then apply this middleware only to the routes where it makes sense. Time to learn how to build such a layer in Express!

How to Implement a Caching Middleware Using Redis in Node.js

In this step-by-step tutorial section, you will learn how to implement Redis-based middleware for caching requests in Express. Keep in mind that you can easily adapt the following implementation procedure to any other caching database system.

Prerequisites

To follow this guide, you need these two requirements:

- **Node.js 18+:** If you have not Node.js installed on your machine, [download the installer from the official site](#), launch it, and follow the wizard.
- **Redis:** [Follow the installation guide for your operating system](#) to set up a local Redis server on your computer. Alternatively, you can opt for a Redis provider in the cloud.

The following sub-chapters show only the main steps to integrate Redis into Node.js. To avoid getting lost, we recommend keeping the codebase of the final application at hand by cloning the [GitHub repository that supports this article](https://github.com/Tonel/nodejs-redis-demo):

```
git clone https://github.com/Tonel/nodejs-redis-demo
```

You are now ready to set up a Node.js Express application to add the caching layer to!

Set Up Your Express Project

Create a folder for your Node.js application, enter it, and run the `npm init` command below to bootstrap a new npm project:

```
npm init -y
```

Initialize your project with a `src` folder. This will contain all your Express business logic.

Next, add `express` and `dotenv` to your project's dependencies:

```
npm install express dotenv
```

dotenv allows you to store the Redis server URL in an env, instead of hardcoding it in the code. So, even though it is an optional dependency, it is highly recommended.

Add an empty `.env` file to your project's root folder. You will need it later on.

Then, create the `controllers` folder inside `src/` and add the following `users.js` file to it:

```
// ./src/controllers/users.js
```

```
const UserController = {
  getAll: async (req, res) => {
    // simulate the time to retrieve the user list
    await new Promise((resolve) => setTimeout(resolve, 250));

    // the user list retrieved with a query or an API call
    let users = [
      { id: 1, email: "john.doe@example.com", name: "John Doe" },
      { id: 2, email: "jane.smith@example.com", name: "Jane Smith" },
      { id: 3, email: "alice.jones@example.com", name: "Alice Jones"
    },
      { id: 4, email: "bob.miller@example.com", name: "Bob Miller" },
      { id: 5, email: "sara.white@example.com", name: "Sara White" },
      { id: 6, email: "mike.jenkins@example.com", name: "Mike
Jenkins" },
      { id: 7, email: "emily.clark@example.com", name: "Emily Clark"
    },
      { id: 8, email: "david.ross@example.com", name: "David Ross" },
      { id: 9, email: "lisa.hall@example.com", name: "Lisa Hall" },
      { id: 10, email: "alex.garcia@example.com", name: "Alex Garcia"
    },
    ];

    res.json({
      users: users,
    });
  },
};
```

```
module.exports = { UserController };
```

This defines an Express controller containing the business logic for a simple API endpoint that returns a list of users. Note the use of `setTimeout()` to simulate the delay introduced by a query or API call to retrieve the desired data. This will be useful to verify the performance improvement introduced by the Node.js Redis caching layer.

Create an `index.js` file to start your Express server and register the above controller function to a route:

```
// ./index.js

const express = require("express");
// populate proces.env
require("dotenv").config();
const { UserController } = require("../src/controllers/users");

// initialize an Express application
const app = express();
app.use(express.json());

// register an endpoint
app.get("/api/v1/users", UserController.getAll);

// start the server
const port = 3000;
app.listen(port, () => {
  console.log(`Server is running on http://localhost:\${port}`);
});
```

This is what your project's directory will contain so far:

```
├── node_modules/
├── src/
│   └── controllers/
│       └── users.js
├── .env
├── index.js
├── package-lock.json
├── package.json
└── README.md
```

In `package.json`, add the npm script below in the `scripts` object to make it easier to run your application:

```
"start": "node index.js"
```

You can now launch your backend locally with:

```
npm run start
```

[Open in app](#) ↗

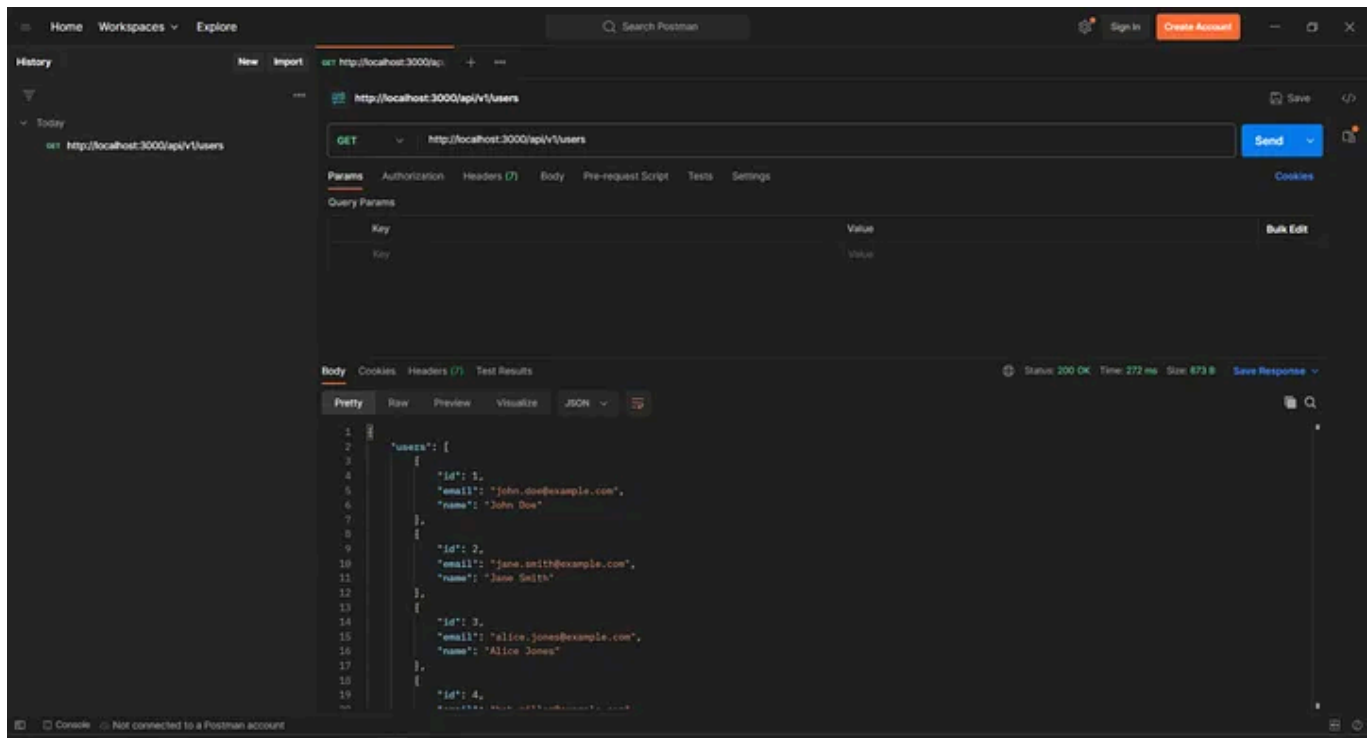
Medium

 Search

 Write



That should return a list of users as below:



Note that the API took 272 ms. Bear that number in mind as it is about to decrease dramatically!

Great, prepare your Node.js architecture to host a Redis caching layer.

Add the Caching Layer to Your Architecture

To build a Node.js Redis caching system for your routes, you need to define a custom middleware. Thus, create the `middlewares` folder inside `src/` and add an empty `redis.js` file to it. This file will contain the logic for:

- Connecting to the Redis server.
- Transforming an API call into a proper Redis key.
- Writing the response data for a request to the Redis database.
- Reading the response data for a request from the Redis database, if present.

In other words, that single file will centralize and encapsulate the entire caching logic of your backend application.

Your project's folder will now contain:

```
├── node_modules/
├── src/
│   ├── controllers/
│   │   └── users.js
│   ├── middlewares/
│   │   └── redis.js
│   └── index.js
├── .env
├── package-lock.json
├── package.json
└── README.md
```

Amazing! Time to populate the `redis.js` file with the required lines of code.

Get Started With Redis

The best way to deal with Redis in Node.js is through the `redis` npm package. Install it with the following command:

```
npm install redis
```

This package enables you to define a client object that exposes all the methods you need to read and write data to Redis. Since all functions in `redis.js` will rely on that object, store it in a global variable:

```
let redisClient = undefined;
```

Use the `createClient()` function to initialize `redisClient` and then call the `connect()` method to connect to the Redis server:

```
async function initializeRedisClient() {  
  // read the Redis connection URL from the envs  
  let redisURL = process.env.REDIS_URI  
  if (redisURL) {  
    // create the Redis client object  
    redisClient = createClient({ url: redisURL }).on("error", (e) => {  
      console.error(`Failed to create the Redis client with error:`);  
      console.error(e);  
    });  
  }  
}
```

```
try {  
  // connect to the Redis server  
  await redisClient.connect();  
  console.log(`Connected to Redis successfully!`);  
} catch (e) {  
  console.error(`Connection to Redis failed with error:`);  
  console.error(e);  
}  
}
```

Encapsulate the Redis connection logic in a function, as you will have to call it in the `index.js` file.

Do not forget to import `createClient` from the `redis` package:

```
const { createClient } = require("redis");
```

Note that the `createClient()` function accepts the URL to your Redis server. To avoid hardcoding it in the code, store in the `REDIS_URI` environment variable in the `.env` file as follows:

```
REDIS_URI="redis://localhost:6379"
```

If you omit this declaration, `initializeRedisClient()` will not do anything due to the top level `if` instruction. Therefore, storing the Redis connection URL in an env is also an effective approach to gain the ability to disable caching programmatically without having to touch the code.

From now on, we will assume that Redis is running locally on port 6379 — the default Redis port. If you are wondering how to specify authentication and database information in the URL, here is what the format of the Redis connection string looks like:

```
redis\[s]://[[username\[[:password]@]\[host\[[:port] [/db-number]
```

Since `initializeRedisClient()` is an async function, you cannot simply call it in `index.js`. Instead, you need to wrap the Express initialization logic in an `async` function and add the `initializeRedisClient()` inside it as below:

```
// index.js
```

```
const express = require("express");
// populate proces.env
require("dotenv").config();
const { UserController } = require("./src/controllers/users");
const { initializeRedisClient } = require("./src/middlewares/redis");

async function initializeExpressServer() {
  // initialize an Express application
  const app = express();
  app.use(express.json());

  // connect to Redis
  await initializeRedisClient();

  // register an endpoint
  app.get("/api/v1/users", UserController.getAll);

  // start the server
  const port = 3000;
  app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
  });
}

initializeExpressServer()
  .then()
  .catch((e) => console.error(e));
```

Try running the Express application again. If you followed the steps correctly, you will see these messages in the terminal:

```
Connected to Redis successfully!
Server is running on http://localhost:3000
```

You Express application now connects to Redis as expected!

Define the Redis Key Generation Logic

When the Node.js server receives a request for an exposed endpoint, Express intercepts it and translates it into the `req` object. Thus, a request is nothing more than a JavaScript object at the application level.

To produce a Redis key associated with the incoming request, you might think of serializing the `req` object into a JSON string and then hashing it with a built-in method. The problem is that this simple solution is not very effective. Why? Because you want the key to be the same even if the order of the query parameters or fields in the body is different.

If you follow that approach, `/api/v1/users?offset=10&page=1` and `/api/v1/users?page=1&offset=1` will produce two different keys. However, those are exactly the same API call. This Redis key generation strategy leads to overfilling the Redis database storage, limiting the server's caching capabilities as a result!

A more effective solution is to rely on `object-hash`, a popular npm package for generating order-insensitive, consistent, and reliable hashes. Add it to the project dependencies with:

```
npm install object-hash
```

Then, import it in your `redis.js` middleware file:

```
const hash = require("object-hash");
```

Now, instead of hashing the `req` object directly, you should define a custom object with only the data you want to involve in the hashing operation. This is a tip to have more control over the hashing process. Another tip is to avoid using the hash directly as the Redis key. The reason is that if you explore the data stored in the Redis database, you want to understand which API endpoint the `<key, value>` record refers to. Therefore, you should add the `req.path` before the hash as below:

```
function requestToKey(req) {  
  // build a custom object to use as part of the Redis key  
  const reqDataToHash = {  
    query: req.query,  
    body: req.body,  
  };  
}
```

```
  // `${req.path}@...` to make it easier to find  
  // keys on a Redis client  
  return `${req.path}@${hash.sha1(reqDataToHash)}`;  
}
```

Given the `http://localhost:3000/api/v1/users` GET call, `requestToKey()` will return something like:

```
"/api/v1/users@c0004b1d98e598127f787c287aaf7c0db94454f1"
```

Fantastic! It only remains to define the reading and writing logic.

Implement the Caching Middleware

The prerequisite for both write and read operations in Redis is that there is an active connection to a Redis server. Add a utility function to `redis.js` to verify that:

```
function isRedisWorking() {  
  // verify wheter there is an active connection  
  // to a Redis server or not  
  return !!redisClient?.isOpen;  
}
```

Thanks to the `isOpen` field, you can verify whether the Redis client's underlying socket is open or not.

The next step in implementing a Node.js Redis caching system is to define a function to write data to the Redis database and a function to read data from it.

This is what the write function looks like:

```
async function writeData(key, data, options) {  
  if (isRedisWorking()) {  
    try {  
      // write data to the Redis cache  
      await redisClient.set(key, data, options);  
    } catch (e) {  
      console.error(`Failed to cache data for key=${key}`, e);  
    }  
  }  
}
```

```
}  
}
```

That accepts a key, its value, and some options and passes them to the `set()` method of the `redisClient` object to store data in Redis. `options` is an object that involves the following fields:

```
{  
  EX, // the specified expire time in seconds  
  PX, // the specified expire time in milliseconds  
  EXAT, // the specified Unix time at which the key will expire, in seconds  
  PXAT, // the specified Unix time at which the key will expire, in milliseconds  
  NX, // write the data only if the key does not already exist  
  XX, // write the data only if the key already exists  
  KEeptTL, // retain the TTL associated with the key  
  GET, // return the old string stored at key, or "undefined" if key did not exist  
}
```

These correspond to the options supported by the `SET` command in Redis. [Find out more in the documentation.](#)

Similarly, you can implement the read function this way:

```
async function readData(key) {  
  let cachedValue = undefined;  
  
  if (isRedisWorking()) {  
    // try to get the cached response from redis  
    cachedValue = await redisClient.get(key);  
    if (cachedValue) {  
      return cachedValue;  
    }  
  }  
}
```



```

    }
  }
}

```

In this case, you can use the `get()` method of `redisClient` to read the value associated to the specified key from the Redis database.

Note that `requestToKey(req)` was not used in either of these functions. That is because the write and read function should remain as generic as possible. Now, use them to implement the Redis caching middleware function:

```

function redisCacheMiddleware(
  options = {
    EX: 21600, // 6h
  }
) {
  return async (req, res, next) => {
    if (isRedisWorking()) {
      const key = requestToKey(req);
      // if there is some cached data, retrieve it and return it
      const cachedValue = await readData(key);
      if (cachedValue) {
        try {
          // if it is JSON data, then return it
          return res.json(JSON.parse(cachedValue));
        } catch {
          // if it is not JSON data, then return it
          return res.send(cachedValue);
        }
      }
    } else {
      // override how res.send behaves
      // to introduce the caching logic
      const oldSend = res.send;
      res.send = function (data) {
        // set the function back to avoid the 'double-send' effect
        res.send = oldSend;

```

```

        // cache the response only if it is successful
        if (res.statusCode.toString().startsWith("2")) {

```

```
        writeData(key, data, options).then();
    }

    return res.send(data);
};

// continue to the controller function
next();
}
} else {
    // proceed with no caching
    next();
}
};
}
```

A lot is going on in the above snippet, so let's break it down step by step.

First, notice that the `redisCacheMiddleware()` is not an Express middleware function. Instead, it returns an Express middleware function. That is the recommended way to define a middleware with custom optional arguments. Change the default `options` object according to your needs.

Next, `redisCacheMiddleware()` generates the Redis key for the incoming request and uses it to read data from Redis. If there is an associated value, it returns it immediately, without proceeding to the controller function. If `JSON.parse()` raises an exception, it produces the response with the generic `res.send()` instead of the specific `res.json()`. In this way, the caching logic works with both JSON and non-JSON data.

Otherwise, the function overwrites the `res.send()` method to call `writeData()` on successful responses. That way, whenever an API with this middleware returns a 2XX response, the response data is stored on Redis. Since `res.json()` calls `res.send()` under the hood, you do not have to also overwrite `res.json()`. Then, it continues to the controller function. When this calls `res.json()` or `res.send()` the custom response function with

caching logic will be called instead. Thus, the produced data will be written to Redis and then returned by the server.

Wonderful! The next step is to use Redis middleware on an endpoint.

Register the Middleware to the Routes You Want to Cache

Import the `redisCachingMiddleware` middleware function in `index.js` and use it in the route definition as follows:

```
app.get("/api/v1/users", redisCachingMiddleware(), UserController.getAll);
```

Note the `()` after `redisCachingMiddleware`. That is required as `redisCachingMiddleware` is not a middleware function itself, but it returns a middleware function. So, you need to call the function to get the desired result.

If you want to specify custom options for a single route, you could do it this way:

```
app.get(
  "/api/v1/users",
  redisCachingMiddleware({
    options: {
      EX: 43200, // 12h
      NX: false, // write the data even if the key already exists
    },
  }),
  UserController.getAll
);
```

Well done! You just implemented a Node.js Redis caching layer!

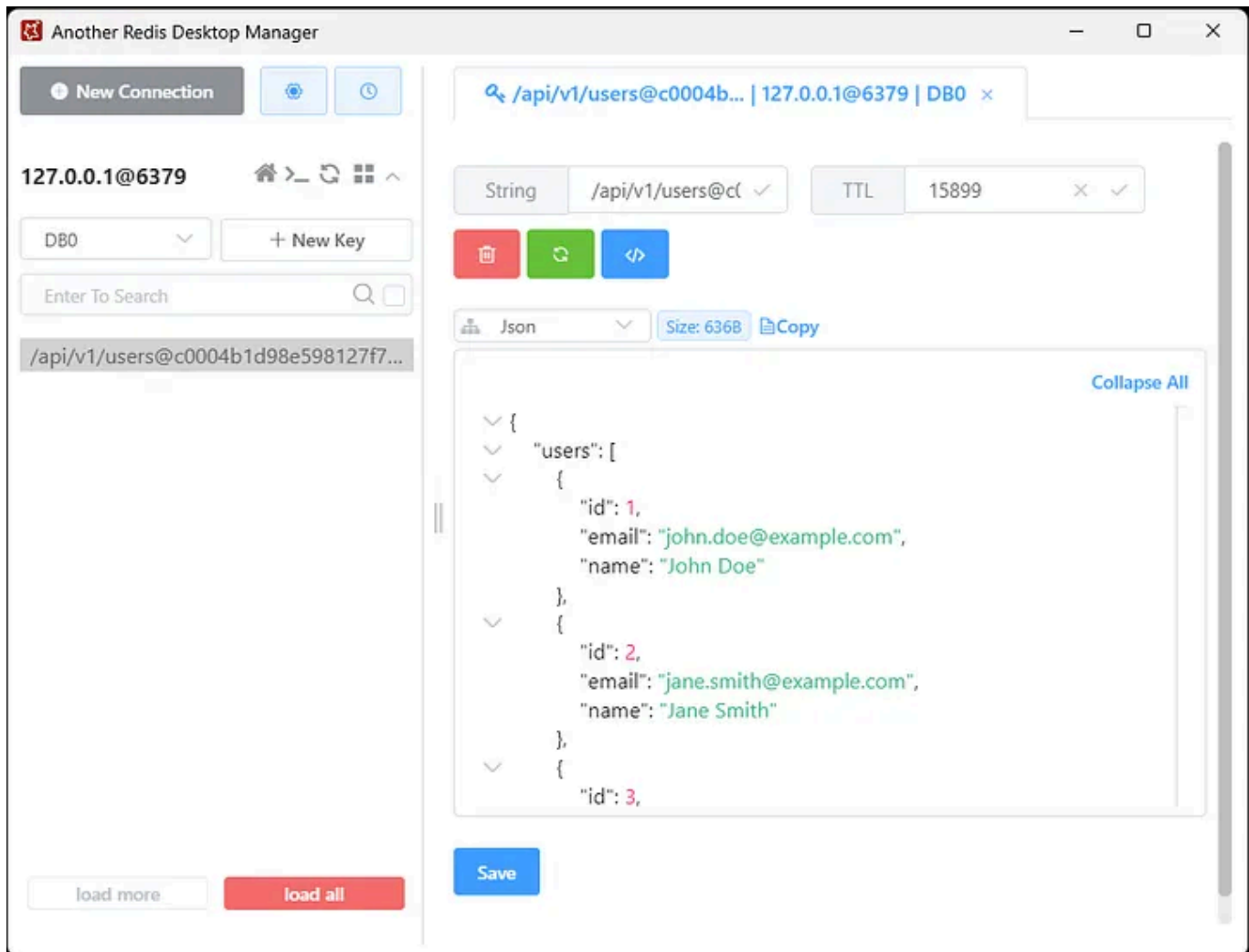
Put It All Together

If you follow all steps above carefully, your Express project will now contain the same code as the [repository that supports this guide](#). To test the caching system, start the Redis server locally and the demo application you retrieved from GitHub in the “Prerequisites” section. Enter the project folder, install the project’s dependencies, and execute the `start` npm script:

```
cd "nodejs-routing-demo"  
npm install  
npm run start
```

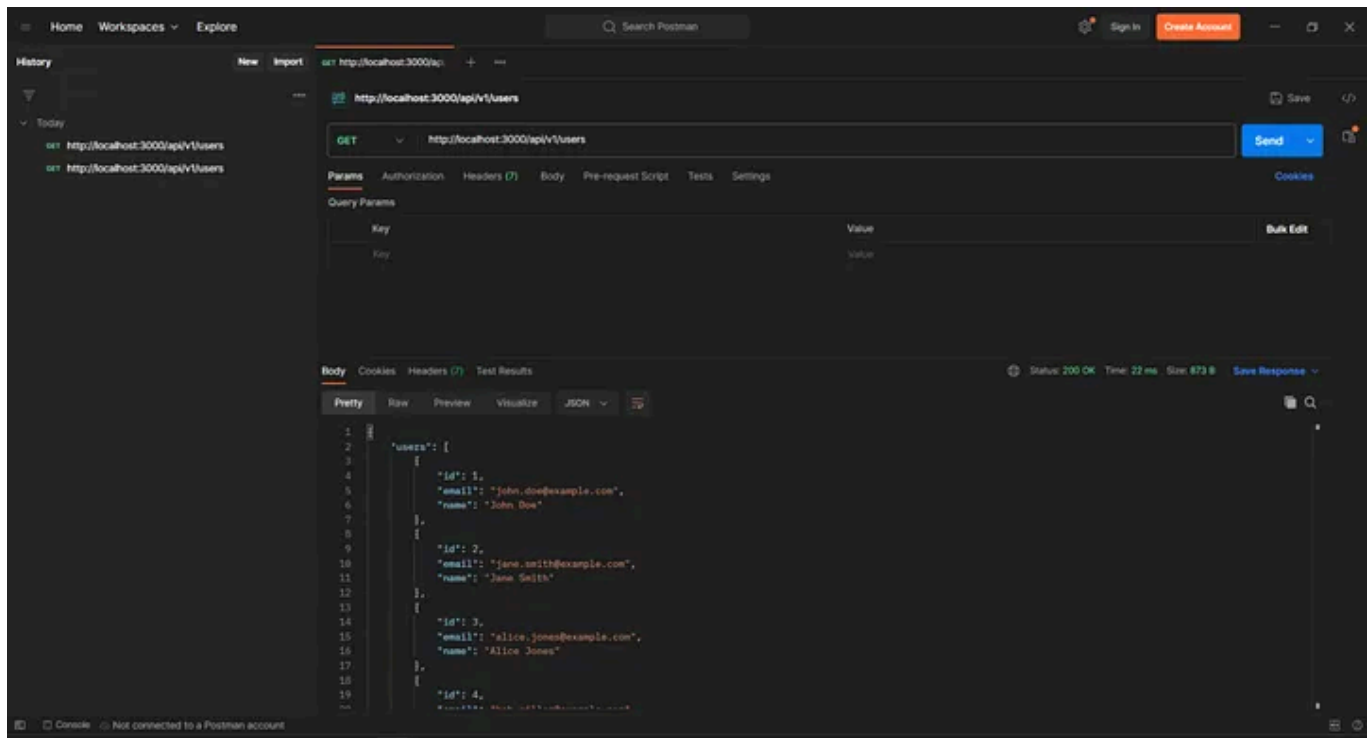
The demo server should now be listening on port `3000`.

Make a GET HTTP request to `http://localhost:3000/api/v1/users`. This will now trigger the caching logic and the API response will be stored in the Redis database. Verify that with a Redis client:



The value associated with the selected key contains the same data returned by the API.

Call the same endpoint a second time, and note how fast the server will produce a response:



Wow! This time, the server returns a response in 22 ms! Not surprisingly, that is exactly the difference between the 272 ms required by the API the first time and the 250 ms forced wait in the code.

Et voilà! Your Redis-based caching layer in Node.js works like a charm!

Extra: Compress Data Before Storing It in Redis

The approach to implementing the caching layer in Node.js presented above is great, but it has one major drawback. JSON is not the most byte-efficient data format. Storing raw data directly in Redis is good for readability, but it comes at the cost of memory usage.

To avoid that, you can compress the raw response produced by the server before writing it to Redis and then decompress it after reading it accordingly. All you have to do is add a compression option to your `writeData()` and `readData()` functions and use the Node.js built `zlib` library as below:

```

async function writeData(key, data, options, compress) {
  if (isRedisWorking()) {
    let dataToCache = data;
    if (compress) {
      // compress the value with ZLIB to save RAM
      dataToCache = zlib.deflateSync(data).toString("base64");
    }

    try {
      await redisClient.set(key, dataToCache, options);
    } catch (e) {
      console.error(`Failed to cache data for key=${key}`, e);
    }
  }
}

async function readData(key, compressed) {
  let cachedValue = undefined;
  if (isRedisWorking()) {
    cachedValue = await redisClient.get(key);
    if (cachedValue) {
      if (compressed) {
        // decompress the cached value with ZLIB
        return zlib.inflateSync(Buffer.from(cachedValue,
"base64")).toString();
      } else {
        return cachedValue;
      }
    }
  }

  return cachedValue;
}

```

Then, add an optional `compression` argument to the `redisCachingMiddleware()` function all well, and pass its value to `writeData()` and `readData()`:

```

function redisCachingMiddleware(
  options = {
    EX: 21600, // 6h
  },

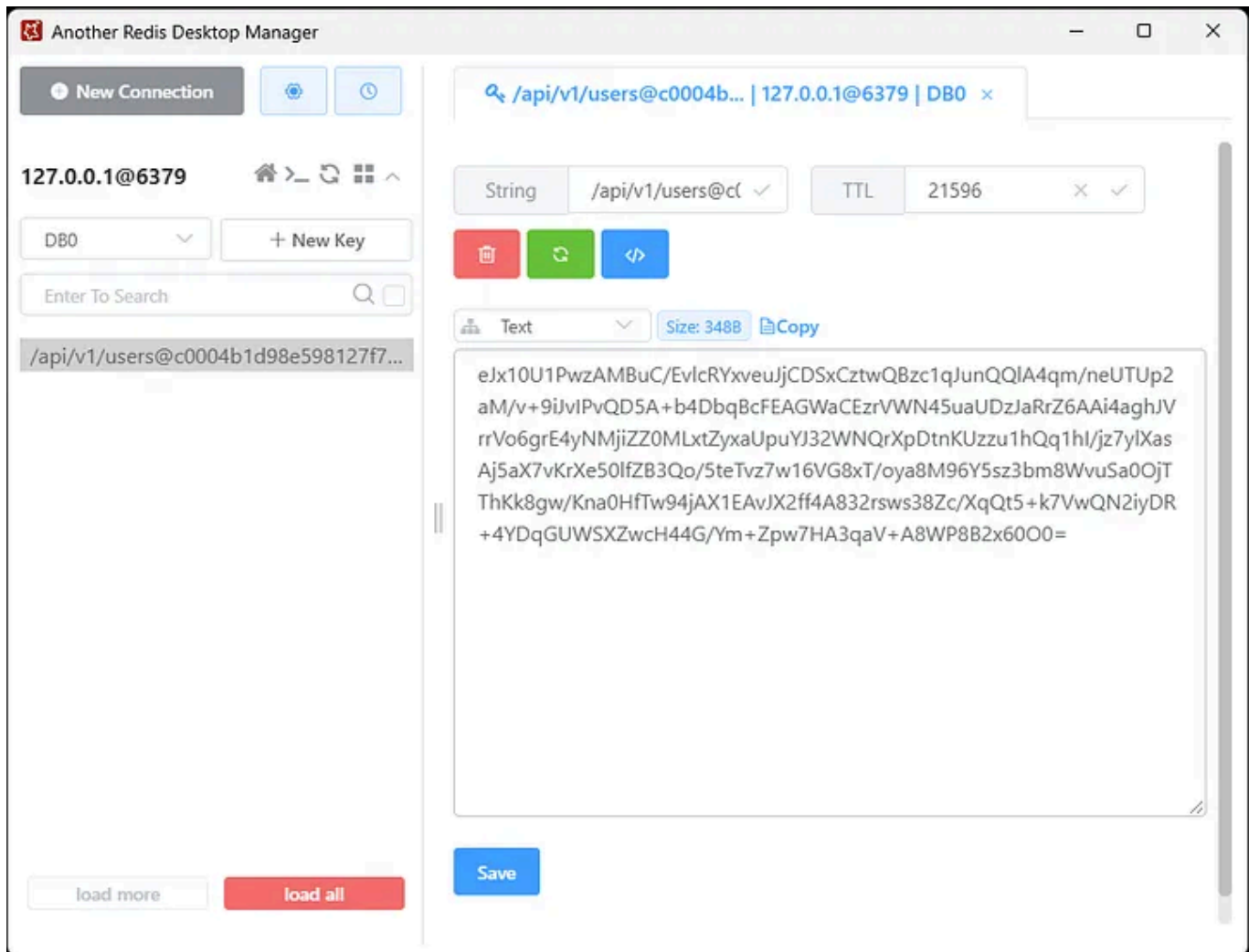
```

```

    compression = true // enable compression and decompression by default
  ) {
    return async (req, res, next) => {
      if (isRedisWorking()) {
        const key = requestToKey(req);
        // note the compression option
        const cachedValue = await readData(key, compression);
        if (cachedValue) {
          try {
            return res.json(JSON.parse(cachedValue));
          } catch {
            return res.send(cachedValue);
          }
        } else {
          const oldSend = res.send;
          res.send = function (data) {
            res.send = oldSend;
            if (res.statusCode.toString().startsWith("2")) {
              // note the compression option
              writeData(key, data, options, compression).then();
            }
            return res.send(data);
          };
          next();
        }
      } else {
        next();
      }
    };
  }
}

```

This time, the Redis client will show some compressed data:



Note that the size of the stored value decreased from 636 bytes to 348 bytes. That is a 45% memory savings! On larger responses, the size gain will be even more noticeable.

Thus, although data compressions and decompressions add a small server CPU overhead and response time, this new approach leads to much better memory utilization. Considering how expensive RAM is, that is a huge achievement that can save your architecture a lot of money!

To explore the entire code of the Node.js Redis caching layer with compression, move to the `compression` branch in the [tutorial's repository](#):

```
git checkout compression
```

Conclusion

In this article, you learned what a Node.js caching layer is, why Redis is the ideal cache provider to build it, and how to implement it. A caching layer is a set of files in a Express application that contain all the Redis caching logic. This allows you to keep your codebase tidy and organized while providing huge performance benefits. Adding such a layer to your backend is easy, and here you saw how to do it in this Redis Node.js guided tutorial.

Originally published at <https://semaphoreci.com> on February 13, 2024.

Nodejs

Node Js Tutorial

Redis

Software Development

JavaScript



Written by Semaphore

Follow

2.8K Followers

Supporting developers with insights and tutorials on delivering good software. ·

<https://semaphoreci.com>

More from Semaphore

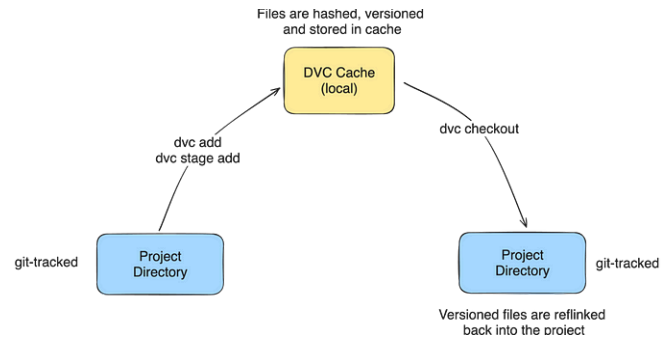


Semaphore

How to Handle Imbalanced Data for Machine Learning in Python

When dealing with classification problems in Machine Learning, one of the things we have...

Mar 7 189 1

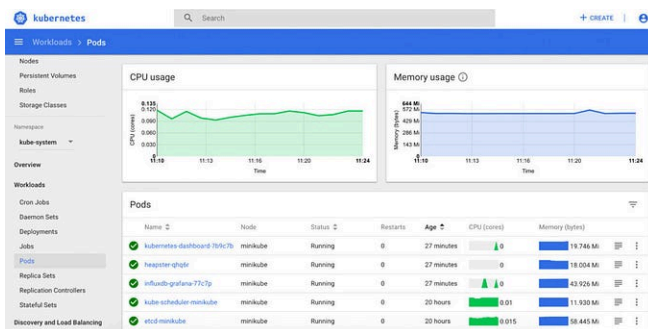


Semaphore

MLOps: From Jupyter to Production

Jupyter notebooks are great for learning and running experiments on Machine Learning...

Feb 1 652 4



Semaphore

10 Open-Source Tools for Optimizing Cloud Expenses



Semaphore

React Compiler: What Is It and How Will It Change Frontend...

In today’s time, many organizations are using cloud technologies. Businesses rely heavily...

React 18 has been around for more than two years, and it is finally time to welcome React...


Jun 11👍 54💬 3🔖+⋮

May 9👍 261💬 3🔖+⋮

See all from Semaphore

Recommended from Medium




 Suneel Kumar

Uploading Large Videos with Node.js: A Comprehensive Guide

In today’s digital age, video content has become an integral part of our online...

Jul 7👍 45🔖+⋮



 Rakesh Kumar in Nerd For Tech

Handling Large Numbers of Promises in Node JS

Practical Tips for Managing Multiple Promises in Node JS Applications

★ Jan 24👍 78💬 1🔖+⋮

Lists



Stories to Help You Grow as a Software Developer

19 stories · 1196 saves



General Coding Knowledge

20 stories · 1385 saves



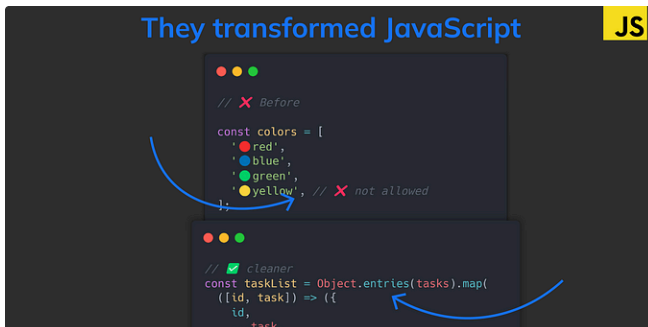
Coding & Development

11 stories · 697 saves



Good Product Thinking

11 stories · 631 saves

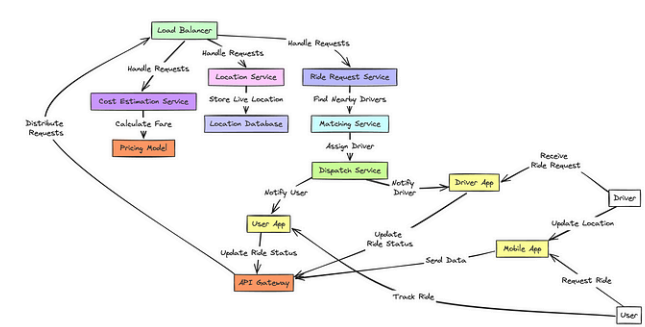


Tari Ibaba in Coding Beauty

The 5 most transformative JavaScript features from ES8

5 juicy ES15 features with new functionality for cleaner and shorter JavaScript code.

Jun 20 228 4

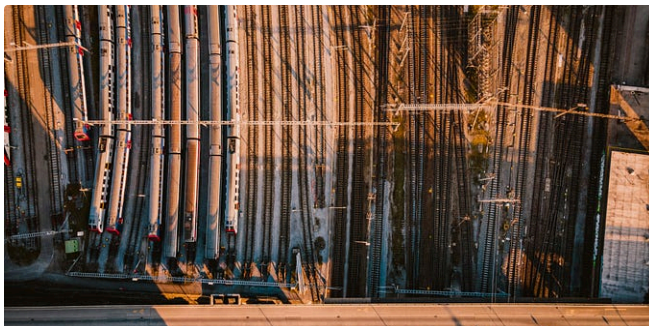


Kevin Wong

Uber System Design

Draft Notes

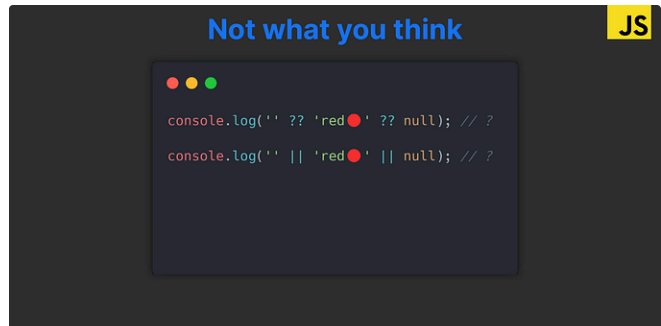
May 16 104



Andrew Zuo

Async Await Is The Worst Thing To Happen To Programming

I recently saw this meme about async and await.



Tari Ibaba in Coding Beauty

?? vs || in JavaScript: The little-known difference

Learn it once and for all and avoid painful bugs down the line.

★ Jun 21 🖱 1.7K 💬 111



★ Jun 8 🖱 489 💬 4



See more recommendations