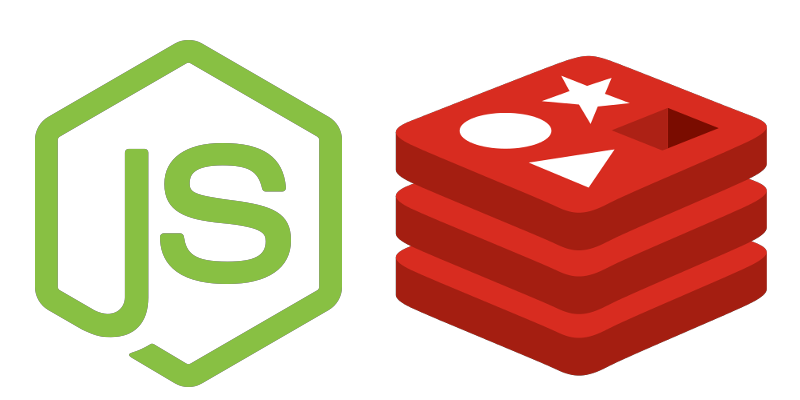
Make your node server faster by caching responses with redis ⏰

Oct 14, 2016 • 9 minute read

NodeJs is known for its speed with respect to async tasks, but there’s still a lot of potential to make responses from your server even faster, sometimes by orders of magnitude.

In this post, we are going to go through a brief introduction to the concept of caching, along with a tutorial on how to implement it using redis and an express server.



What is caching?

Normally, when you make a web server with a database, each request to the server entails one or more requests to the database, and some processing of the results before sending back a response.

For example, consider a database which has a list of peoples names, along with their age. Our server handles requests which contain a persons name, and returns their age after querying the database.

This seems pretty straightforward, but can sometimes be inefficient. If there are a lot of people who want to know Doug’s age, and we get many requests to our server asking for it, we should find a better way than to query the database each time we receive a request (Since so many database queries can be expensive).

**Enter : the cache layer**

The cache layer is a temporary datastore, which is much faster than the database. The server, after receiving a request, first checks if the cache has the response available. If so, it sends it to the client. If not, it queries the database as usual, and stores the response in the cache before sending it back to the client. This way, every response is either cached, or retrieved from the cache, and as a result, the load to our server and database is reduced.

Building our server

Prerequisites

This example uses [NodeJs](https://nodejs.org/en/) v6.x.x, along with [redis](http://redis.io/) for our cache, so make sure you have those installed first.

Start up your redis server using the command :

redis-server

Create a new directory for your project, and create a package.json file, and install the required node modules by running :

npm init

npm install --save redis express

Set up the mock database service

The first module we are going to set up is our mock database (since installing and operating an actual database would be a whole other post in itself)

Create a file called age-service.js

/\*

The mock data which we have in our database

\*/

const ages = {

John: '20',

Michelle: '34',

Amy: '31',

Doug: '22'

}

/\*

We create an async function, which accepts a name, and a callback function to

be called once we fetch the age from out database.

To simulate the time it takes to fetch results from an actual database, we set

a timeout of 1 second, and then return the age of the person requested.

\*/

const getAgeFromDb = (name, cb) => setTimeout(() => {

//This is to verify that out database is being called.

console.log('Fetching from db')

//Returns "Does not exist" if an unknown name is given

const age = ages[name] || 'Does not exist'

// Call the callback function with the result

cb(age)

}, 1000)

module.exports = getAgeFromDb

Set up the main server

Create a file called index.js. This will act as the entry file which starts our [express](https://expressjs.com/) server.

const express = require('express');

const app = express();

//This is the age service that we just made

const ageService = require('./age-service');

app.get('/', function(req, res) {

/\*

Get the name from the request query

For example, localhost:3000/?name=foo

would give "foo" as the name

\*/

const {name} = req.query

ageService(name, age => {

/\*

Once our age service gives us the age,

send it to the client as a response

\*/

res.end(age)

})

});

app.listen(3000, function() {

console.log('App listening on port 3000');

});

At this point, you can start the server by running

node index

Now, make a request to find out Dougs age by opening <http://localhost:3000/?name=Doug> on your browser, or running this command (on unix systems) :

curl -w "\ntime taken : %{time\_total}\n" "http://localhost:3000/?name=Doug"

which should give you something like :

22

time taken : 1.012

You should also see “Fetching from db” on your console every time you make a request.

Creating our cache layer

We create a new file, cache.js to initialize and expose the redis client :

const redis = require('redis')

const client = redis.createClient()

//Incase any error pops up, log it

client.on("error", function(err) {

console.log("Error " + err);

})

module.exports = client

We use the popular [redis](https://www.npmjs.com/package/redis) node library to do this.

Now that we have our redis client ready, let’s modify age-service.js to make use of it :

//Import the cache module we just created

const cache = require('./cache');

const ages = {

John: '20',

Michelle: '34',

Amy: '31',

Doug: '22'

}

const getAgeFromDb = (name, cb) => setTimeout(() => {

console.log('Fetching from db')

const age = ages[name] || 'Does not exist'

cb(age)

}, 1000)

//We now export a new function, which makes use of the cache

module.exports = (name, cb) => {

//First, check if the age exists in our cache

cache.get(name, (err, age) => {

if (age !== null) {

//If it does, return it in the callback

return cb(age)

}

/\*

At this point, we know that the data we want does not exist in the cache

So, we query it from our mock database, like before

\*/

getAgeFromDb(name, age => {

//Once we get the age from the database, store it in the cache.

cache.set(name, age, () => {

//At this point, our data is successfully stored in the redis cache

// We now return the age through the callback

cb(age)

})

})

})

}

To test this, start the server up again by running :

node index

And run :

curl -w "\ntime taken : %{time\_total}\n" "http://localhost:3000/?name=Doug"

This will give you a similar response like last time,

22

time taken : 1.012

But, try and run it again, and you will see a drastic change in the time taken to fetch the result :

22

time taken : 0.007

Also, you won’t see “Fetching from db” being logged.

Try doing this with any of the other names in our database, and you will observe that they take upwards of 1 second to give the response the first time, but give it almost immediately for successive requests.

Congratulations! You have successfully made a cache layer for your node server.

If you feel like you got stuck somewhere, or need a quick solution, you can find the working project [here](https://github.com/sohamkamani/blog-example__node-redis-cache).

Going forward

While this solution may work for small projects, you might want to opt for a more robust library like [cache manager](https://www.npmjs.com/package/cache-manager) for production grade applications. Although, it’s important to know base concepts first before running off with a shiny library.

There are also some cases where you should be careful with using a cache layer :

1. If the information you are caching changes (and it almost always does), it makes sense to include a timeout after which cached information expires. You could also manually clear the cache when a change occurs. In our example, since we were using age, and its only possible for someones age to increase by the day, we could clear our entire cache once daily. Alternatively, we could manually erase each persons individual cached age on their birthday.
2. *Caching sensitive information* : Any kind of sensitive data like passwords and account numbers is best left uncached, since caching creates an additional place whose security could be compromised. Additionally, any change in this info must be reflected in the cache as well, and immediately. We cannot afford to have a fixed refresh interval in this case.
3. *Rapidly changing information* : If the data you are caching changes faster than the rate of requests coming in, it’s not really worth it (and can sometimes even make your response times slower). You do not always need to cache your server responses. Caching is best suited for responses which take a lot of time to fetch and compute, but which are requested often.