How to do logging in an Express API? Part 1

A best practice guide for Winston, Morgan, Sequelize logs

Hargitai Soma · Follow
4 min read · Feb 11, 2020

Logging is a default need for any APIs. Even though, there is no real standard about how to do it. So first, let's ask the main questions.

(note: I won't separately remind you to npm install all the packages you will import, but of course you should do it.)

Main Concepts of Logging

When do we want to see logs?

Logs should be shown if something interesting is happening. If our API is already running on the production server, we are not interested in all the small events, but the unexpected ones: something breaks in the API or something happens which is not dangerous but unusual. First one will be an Error, the second one will be a Warning. Both should tell us what happened, an Error should even provide information about where did it exactly happen in the code.

In development, we want more information. A frontend developer won't touch the backend code, but it expects to have some basic logs about: Did it receive the GET or the POST request? Was it successful? Were the parameters okay?

The backend developer needs full control: see how the process is running, sometimes even log all the parameters, whole requests.

What can be a log source?

It is important to see that logs won't simply coming from our direct commands.

```
console.log('Something is happening here!');
```

A log message may come from:

- direct log in the code, made by the developer
- Javascript engine, for example interpreter errors
- a logging library which creates logs through a background worker
- any library which creates its own logs

What can be a log destination?

Usually we see logs in the console as the Standard Output. If you run your application on a cloud service like AWS's EC2, it will catch logs of the Standard Output and handle them in its own log handler service. You can also send your logs to a file.

These options can be set to work in parallel: you may want to create a log file for errors and warnings, another log file for a detailed, verbose output and use standard output to have log messages to debug.

Creating our logger

```
console.error('This is an error!');
console.info('This is some info about the process');
console.trace('This is data');
```

If you want more functionalities, you will rather choose a more robust solution. In server-side Javascript it will be Winston.js. Let's create our logger.js file!

```
import { createLogger, transports } from 'winston';

const loggerConfiguration = { … };
export const winstonLogger = createLogger();
winstonLogger.add(new transports.Console(loggerConfiguration));

export default winstonLogger;
```

It is pretty short and simple. We create a logger, add console (the terminal where you run your API) as a destination for the logs and configure how to format these logs.

Your loggerConfiguration object will define the formatting of your log messages. You can color it, add a timestamp and so on. I use ES6 syntax for imports and exports. Now my export is a default export, so you can import it with any name you'd like to use. I have added a named export for winstonLogger, we will need it. I will provide more detailed code below about these things.

If you would like to use this logger, you can simply do it in any of your files.

```
import myNiceNewLogger from '../loggerFolder/logger';

myNiceNewLogger.info('My very first Winston log!');
```

An error may come from a service, for example a database call may run to error if the database is not available. Now I put a *throw Error()* into the code to fake this situation.

```
try {
  throw Error('Some service throws an error');
} catch (errorResponse) {
  myNiceNewLogger.error(
    ‘An error happened when I was calling my service!’,
  );

myNiceNewLogger.error(errorResponse);

// we should also make sure to have an appropriate response here.
//   If it is an endpoint, we will return something like
//   400 — Bad Request
}
```

Adding Morgan

Our first additional log source will be Morgan. This library is able to catch all the requests and provide all the information coming from a client: browser type, queried endpoint etc. Morgan itself can directly log what it has found, but our logs will be much more consistent if we forward its messages into our Winston logger and let Winston process it.

Add this code to your logger.js

```
  import morgan from ‘morgan’

  export const morganLogger = morgan(
    ‘combined’,
    {
      stream: {
        write: (text) => winstonLogger.debug(`Morgan says: ${text}`)
      },
    },
  );
```

So morganLogger will be a Morgan logger instance, data output will be ‘combined’ (it is a detailed format, you can also choose ‘tiny’ for less) and we will stream everything to Winston.

To be able to have Morgan logs, you have to add it to your Express server. It is pretty likely that you already have a server.js for your Express API, so you simply have to add the newly created morganLogger to be used by the app. It will look something like:

```
import express from ‘express’;
import { winstonLogger, morganLogger } from ‘./loggerFolder/logger’;

const app = express();
winstonLogger.info(‘Adding Morgan to the application…’);
app.use(morganLogger);

…
```

It’s done! Now any hits on the Express endpoints will go through Morgan, then it will give its message to Winston and Winston will send logs to the console.

You are ready with the main logger structure, well done! Read the second part for details:

How to do logging in an Express API? Part 2

Winston Morgan Logging Express Loglevels

Written by Hargitai Soma

Follow

1 Follower

Mathematician, fullstack Dev, Typographer. Interested in everybody and everything… obviously

More from Hargitai Soma

Hargitai Soma

How you should have started to add Swagger to your Express API

You will get answers for these questions:

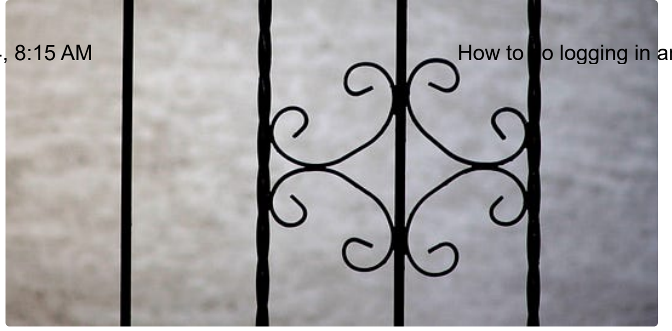
Mar 17, 2020 2

Hargitai Soma

Sequelize, associations & Jest

How to pack everything to one create database call and how to test it

Feb 10, 2020

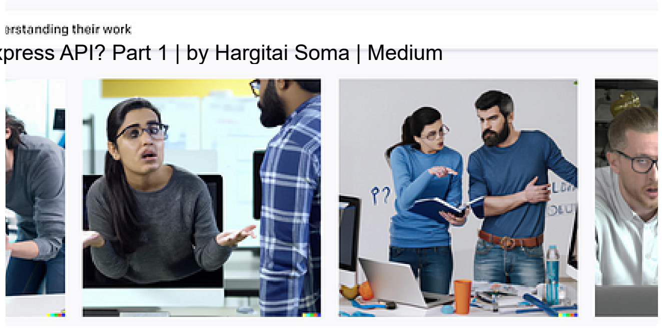


Hargitai Soma

How to do logging in an Express API? Part 2

A best practice guide for Winston, Morgan, Sequelize logs

Feb 11, 2020



Hargitai Soma

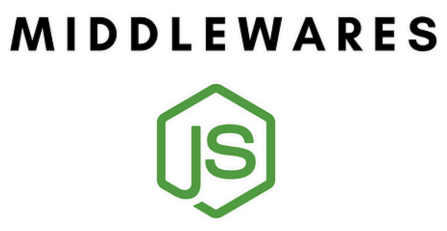
The ChatGPT hype: How AI would change the life of software...

Last autumn was about AI. Talking about DALL-E became mainstream, then the...

Feb 10, 2023

See all from Hargitai Soma

Recommended from Medium



Muhammad Adeel

Express.js Middlewares

An Introduction to Middlewares, Types, and Usage in Express.js

Feb 12 11

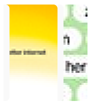
Amir Mustafa

Winston: Production Level Logger in Javascript

→ In today's article we will learn about a production-level logger Winston that is used...

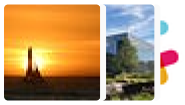
Mar 8 2

Lists



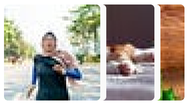
Staff Picks

670 stories · 1081 saves



Stories to Help You Level-Up at Work

19 stories · 661 saves



Self-Improvement 101

20 stories · 2165 saves

Productivity 101

20 stories · 1923 saves

Munira Akter

Choosing the Right Logging System for Your Node Js...

Logging is crucial for developers working on Server Side Applications, providing insights...

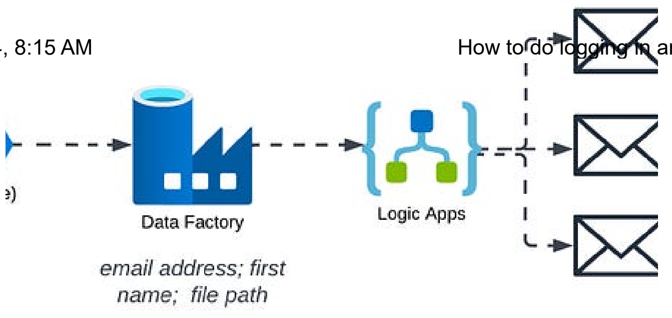
May 5

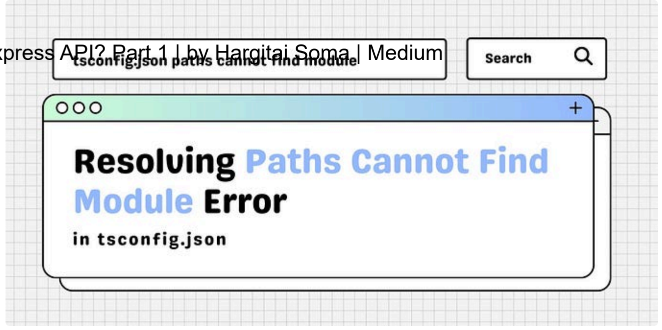
The Tech Platform

Advanced NLog features in ASP.NET Core

In ASP.NET Core development, logging plays a pivotal role in the lifecycle of an application...

Feb 6 1




tsconfig.json paths cannot find module
Search
Resolving Paths Cannot Find Module Error
in tsconfig.json

Peiqing Zhang

Sending Different Attachments to Different Emails by Using Azure…

This article is based on the below YouTube video, but adjusted to allow sending differen…

Feb 11

Monique McIntyre

Resolving 'Paths Cannot Find Module' Error in tsconfig.json

Relative imports can quickly clutter up your codebase, making it difficult to maintain and…

Mar 26 14

See more recommendations