Exploration — Express API for HTTP & Useful npm Packages

Express API for HTTP & Some Useful npm Package

In this exploration, we enhance our knowledge of writing server-side code by exploring the Express API for HTTP and by looking at a few very useful npm packages.

Express API for HTTP

Route handlers and middleware functions in Express receive `Request` and `Response` objects as parameters. Typically, in our programs, we name these parameters `request` or `req`, and `response` or `res`. Let us take a look at these objects to see how we can use them to get information about HTTP requests and set properties on HTTP responses.

The Request Object

Read about the **API for the Express Request Object.** (https://expressjs.com/en/4x/api.html#req) Here are some important properties to understand and use in our apps:

req.query

This object contains a property for each query string parameter in the route.

req.body

This object contains the body of the request as name-value pairs. However, our server program needs to use middleware to parse the body in order to have the body of the HTTP request available as `req.body`. Most web apps support request bodies sent in the following two formats:

1. The request body sent by submitting a form using the HTTP method `POST`.

In this case, the value of the request header `Content-Type` is `application/x-www-form-urlencoded`. We have been using the Express middleware `urlencoded` to parse the form by adding the following statement to our server program:

```
app.use(express.urlencoded({extended: true}));
```

2. The request body sent as a JSON object.

In this case, the value of the request header `Content-Type` is `application/json`. We can use the Express middleware `json` to parse the request by adding the following statement to our server program:

```
app.use(express.json())
```

req.headers

This object contains all the request headers as key-value pairs, where the keys are the header names and values are the header values. For example, we can log all the request headers by using the following `for in` statement:

```
for (const property in req.headers) {
  console.log(`${property}: ${req.headers[property]}`);
}
```

The Response Object

Read about the **API for the Express Response Object.** (https://expressjs.com/en/4x/api.html#res) Here are some important properties to understand and use in our apps:

res.set(name, value)

We can set response headers using this method. However, typically we do not manually set any headers, except perhaps `Content-Type`.

res.type(type)

A convenient method to set the `Content-Type` header instead of using `res.set` to set the header. By default, Express sets the content type in the response to `text/html`. But we can set the appropriate MIME type by using this method (we will study MIME types in a later module).

res.status(code)

By default, Express sets the status code to `200`. If we want to set the status code to some other value, we can use this method. Note that `res.status` returns the response object. This means we can chain calls. For example, `res.status(400).json({error: 'Must provide an id value'})`.

res.send(body)

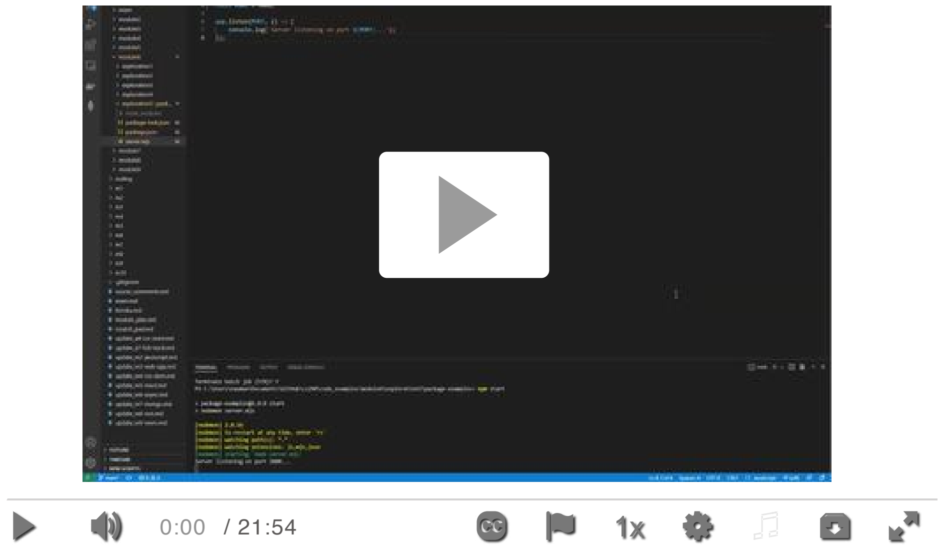
Calling this method sends the response to the client with the response body set to the specified argument, such as of type Buffer, String, object, Boolean, or Array. However, if we are sending back JSON in the response body, we should use `res.json`.

res.json(json)

This method sends back JSON to the client and sets the `Content-Type` header to `application/json`. Using this method, we can send any JSON value as the response body, including JSON arrays.

Useful npm Packages

Let us look at a few npm packages that we will use in the coming assignments.



dotenv

From now on, our programs will use the `.env` file to set the value of the port on which the Express server listens for requests. The `.env` file might be hidden when viewing files on your hard drive, but in VS Code, the file will be visible. To list it outside of an IDE, you can type the `-a` command in a terminal.

In our Express examples, we have been configuring the Express server to listen for requests at port 3000. We have done this by declaring a variable named `PORT` in our `server.js` file, assigning it the value 3000, and then passing it as the first argument in the call to `app.listen`. This works well when developing on our local hard drives.

If we want to deploy our app/site on a web server, it will make sense to separate the port configuration from the rest of our code, so one is not dependent on the other. It is best practice to use a separate file to specify various configuration parameters, such as the port on which the Express server should listen.

The **dotenv (https://github.com/motdotla/dotenv#readme)** package helps us do that. This package reads key-value pairs from a file named `.env` and making these key-value pairs available to your program. To use this package, we need to install it using `npm install` and import it into our program using `import 'dotenv/config'`. Now all the key-value pairs in the file `.env` are available to our program as properties of the object `process.env`.

For example, if the contents of the `.env` file are as follows:

PORT=3000

then the value of the property `process.env.PORT` will be 3000.

node-fetch

JavaScript code running in the browser can use the fetch() API to make HTTP calls. The npm package **node-fetch** **(https://github.com/node-fetch/node-fetch)** provides a more robust yet light-weight API to make HTTP requests from JavaScript code running in an Express app in the Node environment.

GitHub's Node-Fetch document **(https://github.com/node-fetch/node-fetch#common-usage)** provides examples of **asynchronously fetching data from a URL and responding with JSON.**

express-async-handler

Express allows us to specify asynchronous functions as route handlers. However, if an asynchronous route handler throws an exception and doesn't handle it, the Express server can crash. This means that we must properly handle any exceptions thrown by an asynchronous route handler. Proper handling of an exception typically involves

- Catching the exception,
- Logging it, so that at some later time, we can review all the exceptions thrown by our code and determine if we need to fix the code that's throwing the exception, and
- Sending back a response with an error message.

One way to achieve this is that whenever we use an asynchronous route handle, we wrap all the statements of the route handler function inside a `try ... catch` statement. Then in the `catch` block we catch any exception that is thrown and handle it. But there is a simpler way to achieve this by using the npm package **express-async-handler** **(https://github.com/Abazhenov/express-async-handler)**. This package provides a simple middleware function that catches exceptions thrown by asynchronous route handlers and passes these exceptions to Express error handlers.

> **Note:** Express 5, which has not yet been officially released, includes built-in support for handling rejected promises. Thus `express-asynch-handler` will no longer be needed in Express 5.

Summary

In this exploration we looked at the request and response objects provided by the Express API. We also looked at the npm packages `dotenv`, `node-fetch` and `express-async-handler` and how to use these packages.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- The **API docs** **(https://expressjs.com/en/api.html)** for Express 4.x.
- Express source code is available **on github** **(https://github.com/expressjs/express)**. For example, here is the **source code for the response object** **(https://github.com/expressjs/express/blob/4.18/lib/response.js)** for Express 4.18.