5. Express - advanced topics

Middleware

Middleware is an abstraction layer that works as an intermediate between the software layers. Express middleware is a function that is compiled during the lifecycle of the Express server.

The middleware function is basically a function that takes a request object and either returns a response to the client or passes control to another middleware function. In express, every route handler function that we have is technically a middleware function, because it takes a request object and returns a response to the client, so it terminates the request-response cycle.

When we receive a request on the server, that request goes through the *request processing pipeline*. The request processing pipeline in IIS (Internet Information Services) is the mechanism by which requests are processed beginning with a Request and ending with a Response. In this pipeline we have one or more middleware functions, each middleware function either terminates the req-res cycle by returning a response object or it will pass control to another middleware function.

Built-in middleware

Starting with version 4.x, Express no longer depends on Connect. The middleware functions that were previously included with Express are now in separate modules. Express has the following built-in middleware functions:

- `express.static` serves static assets such as HTML files, images, and so on.
- `express.json` parses incoming requests with JSON payloads. Available with Express 4.16.0+
- `express.urlencoded` parses incoming requests with URL-encoded payloads. Available with Express 4.16.0+

Third-party middleware

Use third-party middleware to add functionality to Express apps. Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level. The following example illustrates installing and loading the cookie-parsing middleware function `cookie-parser`.

```
npm i cookie-parser
```

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

Environments

Environment variables are a fundamental part of developing with Node.js, allowing your app to behave differently based on the environment you want them to run in. Wherever your app needs configuration, you use environment variables.

You may be setting a port number for an Express server. Often the port in a different environment (e.g.; staging, testing, production) may have to be changed based on policies and to avoid conflicts with other apps. As a developer, you shouldn't care about this, and really, you don't need to. Here is how you can use an environment variable in code to grab the port.

```
const port = process.env.PORT;
console.log(`Your port is ${port}`);
```

You can also change the way your code behaves according to the environment, for example:

```
if (app.get(env) === 'development')
      app.use(morgan('tiny'));
```

Configuration

You can create configuration files for application deployments. This allows you to define a set of default parameters and extend them for different deployment environments (development, qa, staging, production, etc.).

The configuration files are located in the default config directory. The location can be overridden with the NODE_CONFIG_DIR environment variable. The NODE_ENV environment variable contains the name of our application's deployment environment; it should be *development* by default.

Managing multiple configuration files across different environments can be challenging, and several tools are trying to solve this problem with different approaches. For example, `RC` and `config` are two examples of tools that you can use.

Node-config allows you to create configuration files in your Node application for different deployment environments. With it, you can define a default configuration file that you intend to repeat across environments, then extend the default config to other environments, such

as development, staging, etc. Node-config makes it easier to create and manage a consistent configuration interface shared among all of your deployment environments.

Because node-config is an npm package, you can install it with npm by running the following command:

```
npm i config
```

Node-config supports many file extensions, for example:

- .json
- .json5
- .hjson
- .yaml or .yml
- .coffee
- .js
- .cson
- .properties
- .toml
- .ts
- .xml

First, you have to create a `config` directory and add a `config/default.json` file to it. This will be the default config file and will contain all your default environment variables.

`config/default.json`

```
json
{
    "server": {
      "host": "localhost",
      "port": 0,
    }
  }
```

You will access it in our app by importing `config` and using the `get` method to access the variables.

```
const config = require('config');
const port = config.get('server.port');
const host = config.get('server.host');
```

You can extend the default config file by creating other configuration files. The configuration data of the application will be overridden according to the environment that is currently running.

You must never store passwords or sensitive data in a config file, you have to store all of that in environment variables and then read them in a config file. For this, you can create a file

called `custom-environment-variables.json` and then store the name of the environment variables in it like this:

```
{
      "mail": {
            "password": "app_password"
            }
}
```

Back in the index.js file you can access this information by using the `get()` method.

Debugging

The debug package in Node is a tiny JavaScript debugging utility modeled after Node.js core's debugging technique. Works in Node.js and web browsers.

```
npm install debug
```

Usage: `debug` exposes a function; simply pass this function the name of your module, and it will return a decorated version of `console.error` for you to pass debug statements to. This will allow you to toggle the debug output for different parts of your module as well as the module as a whole.

Every debug instance has a color generated for it based on its namespace name. This helps when visually parsing the debug output to identify which debug instance a debug line belongs to.

```
const debug = require('debug')('app:startup');
const morgan = require('morgan');
const express = require('express');
const app = express();

if (app.get('env') === 'development') {
  app.use(morgan('tiny'));
  debug('Morgan enabled...'); // console.log()
}

const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Listening on port ${port}...`));
```

This is very useful for when you want to see debugging messages according to a specific namespace.

If you want to see the debugging messages for more than one namespace, you can run on the terminal

```
EXPORT DEBUG=app:startup, app:db
```

Or if you want to see the debugging messages for all of the namespaces simply run

```
EXPORT DEBUG=app:*
```

Templating engines

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are Pug, Mustache, and EJS. The Express application generator uses Jade as its default, but it also supports several others.

To render template files, set the following application setting properties, set in app.js in the default app created by the generator:

- `views`, the directory where the template files are located. Eg: `app.set('views', './views')`. This defaults to the views directory in the application root directory.
- `view engine`, the template engine to use. For example, to use the *Pug* template engine: `app.set('view engine', 'pug')`.

Then install the corresponding template engine npm package; for example to install Pug:

```
$ npm install pug --save
```

After the view engine is set, you don't have to specify the engine or load the template engine module in your app; Express loads the module internally. Create a Pug template file named index.pug in the views directory, with the following content:

```
html
  head
    title= title
  body
    h1= message
```

Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the view file. Otherwise, you can omit it.

```
app.get('/', (req, res) => {
  res.render('index', { title: 'Hey', message: 'Hello there!' })
})
```

The view engine cache does not cache the contents of the template's output, only the underlying template itself. The view is still re-rendered with every request even when the cache is on.

Database integration

Adding the capability to connect databases to Express apps is just a matter of loading an appropriate Node.js driver for the database in your app. Some of the most popular databases are the following:

- Cassandra
- Couchbase
- CouchDB
- LevelDB
- MySQL
- MongoDB
- Neo4j
- Oracle
- PostgreSQL
- Redis
- SQL Server
- SQLite
- Elasticsearch

For example, if you want to connect to SQL Server, you need to install the `tedious` package:

```
$ npm install tedious
```

Example:

```
const Connection = require('tedious').Connection
const Request = require('tedious').Request

const config = {
  server: 'localhost',
  authentication: {
    type: 'default',
    options: {
      userName: 'your_username', // update me
      password: 'your_password' // update me
    }
  }
}

const connection = new Connection(config)

connection.on('connect', (err) => {
  if (err) {
    console.log(err)
  } else {
    executeStatement()
  }
})

function executeStatement () {
  request = new Request("select 123, 'hello world'", (err, rowCount) => {
    if (err) {
      console.log(err)
    } else {
      console.log(`${rowCount} rows`)
    }
    connection.close()
  })

  request.on('row', (columns) => {
    columns.forEach((column) => {
      if (column.value === null) {
        console.log('NULL')
      } else {
        console.log(column.value)
      }
    })
  })

  connection.execSql(request)
}
```

Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

MongoDB is a schema-less NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments.

To install `mongoose` simply run the following command on the terminal

```
npm install mongoose validator
```

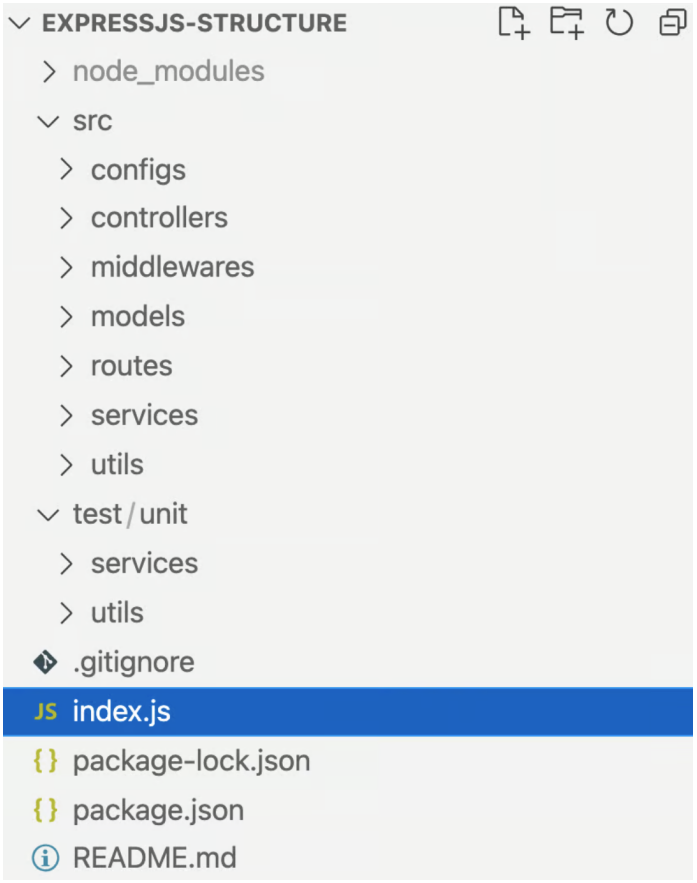
Then create a file ./src/database.js under the project root. Next, we will add a simple class with a method that connects to the database. Your connection string will vary based on your installation.

```
let mongoose = require('mongoose');

const server = '127.0.0.1:27017'; // REPLACE WITH YOUR DB SERVER
const database = 'fcc-Mail';      // REPLACE WITH YOUR DB NAME

class Database {
  constructor() {
    this._connect()
  }

_connect() {
     mongoose.connect(`mongodb://${server}/${database}`)
       .then(() => {
         console.log('Database connection successful')
       })
       .catch(err => {
         console.error('Database connection error')
       })
  }
}

module.exports = new Database()
```

Structuring Express applications

For a good web project, for instance, an API will surely have some routes and controllers. It will also contain some middleware like authentication or logging. The project will have some logic to communicate with the data store, like a database and some business logic.

This is an example structure that can help organize the code for the things I mentioned above. I will explain further how I organized this project below:



The main entry point of this organized Express application is the index.js file on the root, which can be run with Node using node index.js to start the application. It will require the Express app and link up the routes with relative routers. Any middleware will also be generally included in this file. Then it will start the server.

/Controllers- This folder would contain all the functions for your APIs.
Naming of files- xxxxx.controllers.js

/Routes- This folder would contain all the routes that you have created using Express Router and what they do would be exported from a Controller file
Naming of files- xxxxx.routes.js

/Models- This folder would contain all your schema files and and the functions required for the schema would also lie over here.
Naming of files- xxxxx.js

/Middleware- This folder would contain all the middleware that you have created, whether it be authentication/some other function.
Naming of files- xxxxx.middleware.js

/Utils(Optional)- The common functions that you would require multiple times throughout your code
Naming of files- Normal project file naming scheme

/Templates(Optional)- If your code requires you to send certain emails/ HTML code to the client-side, store it in this files
Naming of files- Normal project file naming scheme

/Config(Optional)- Configuration files for third party APIs/services like passport/S3,etc
Naming of files- Normal project file naming scheme

Files in the root of your project:

- app.js- This file would basically be the entry point of the Express application and should be as minimal as possible
- package.json- file which contains all the project npm details, scripts and dependencies.
- .gitignore- The files you don’t want to push to git