24. Dotenv, Router, Middleware, and MVC

Table of Content

1. Dotenv

Environment variables are variables that are set outside of a program, often through a cloud provider or operating system.

In Node, environment variables are a great way to securely and conveniently configure things that don't change often, like URLs, authentication keys, and passwords.

Environment variables are supported out of the box with Node and are accessible via the `env` object (which is a property of the `process` global object.)

DotEnv is a lightweight npm package that automatically loads environment variables from a `.env` file into the `process.env` object.

To use DotEnv, first install it using the command: `npm i dotenv`. Then in your app, require and configure the package like this: `require('dotenv').config()`.

2. Express Router

Use the express.Router class to create modular, mountable route handlers. A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.

```
const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

```
const birdsRouter = require('./birds')

// ...

app.use('/birds', birdsRouter)
```

3. MVC Architecture

MVC is simply a design or architectural pattern used in software engineering. While this isn't a hard rule, but this pattern helps developers focus on a particular aspect of their application, one step at a time.

The main goal of MVC is to split large applications into specific sections that have their own individual purpose.

1. Model

   As the name implies, a model is a design or structure. In the case of MVC, the model determines how a database is structured, defining a section of the application that interacts with the database. This is where we will define the properties of a user that will be store in our database.

2. View

The view is where end users interact within the application. Simply put, this is where all the HTML template files go.

3. Controller

   The controller interacts with the model and serves the response and functionality to the view. When an end user makes a request, it's sent to the controller which interacts with the database.

4. Middleware

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

- This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
```

- This example shows a middleware function mounted on the `/user/:id` path. The function is executed for any type of HTTP request on the `/user/:id` path.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

- This example shows a route and its handler function (middleware system). The function handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  res.send('USER')
})
```

- Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the `/user/:id` path.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

- Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the `/user/:id` path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

  This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
```

```
}, (req, res, next) => {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

Assignments

1. Build a Product Route - `/product`

   Product Schema:

productId	String
title	String
price	Number

   1. `POST /add` (pass data in body)
   2. `GET /`
   3. `DELETE /:productId`
   4. `PUT /:productId` (pass data in body)

2. Build a Review Route - `/review`

   Review Schema:

reviewId	String
userId	String
productId	String
reviewText	String

   1. `POST /add` (pass data in body)
   2. `GET /getAll/:productId`

3. `DELETE /:reviewId`
4. `PUT /:productId` (pass data in body)