Let's use what we know about Node and Express and create a small web application with a REST-like API and some middleware.

Basic Application Requirements

- Create an Express Application that listens on a port for incoming requests and returns the appropriate response to the client.
- Create an Express middleware that logs the request path and the timestamp for each incoming request.
- Use Express static-file-serving to serve up an index file, and some images and HTML files on request.
- Create an API with three different public endpoints: `1) user 2) note and 3) number`.
- For all other undefined routes, return a `"not found"` page to the client.

Setup

Run the following commands in the `node project directory`.

```
$ npm install express
$ npm install body-parser
$ npm install nodemon
```

Require in Necessary Modules

Open your `app.js` file. This will be the entry point for the application when the node server starts up. Require in the necessary modules at the top.

```
// ================================
// Require modules
// ================================
var express = require('express');
var path = require('path');
var bodyParser = require('body-parser');
```

Here is a brief description of each module:

- `express` : Express web framework wrapper for creating APIs.
- `path` : Basic utility module for making path manipulation easy.
- `body-parser` : Allows middleware that parses json from the client.

Create Express App

It's very easy to create a new Express application. Just invoke the `express` module and assign the result to a variable called `app`.

```
// ================================
// Create Express App
// ================================
var app = express();
```

Add Custom Middleware

On every single incoming request, we want to log the timestamp of the request, and also the path of the request. Using express middleware, this is easy. Just call `app.use()` and pass in the middleware function you want invoked. This middleware will get called for every single request that comes in. If you forget to call `next()` at the end of middleware then the thread will just sit idle and not continue along to the next middleware.

```
// ================================
// Custom Middleware
// ================================
app.use(function(req, res, next) {
  console.log('Incoming Request at time: ' + Date.now() + ' with path ' + req.path)
  next();
});
```

Add Express Middleware

`express.static` is used for serving static files, such as CSS, HTML and JavaScript. `The files are looked up relative to the static directory.`

`The files will be looked up in the order the static directories were set using the express.static middleware.`

```
// ================================
// Express Middleware
// ================================
app.use(express.static('./public'));
app.use(express.static('./images'));
app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-
urlencoded
```

Create the User Router

Open the `index.js` file underneath the `/api/user/` directory. This is where the routes and controllers for the `/api/user/` endpoint should be defined. We will define 5 different routes and map them to their respective controllers.

- `/api/user` (get) : **getAllUsers()**
- `/api/user` (post) : **newUser()**
- `/api/user/:id` (get) : **getUser()**
- `/api/user/:id` (delete) : **deleteUser()**
- `/api/user/:id` (put) : **updateUser()**

`user-service` is a module you can use to mimic a database service in memory. It holds all the users in memory and gives acces to several methods for adding, deleting and updating the user list.

Start by creating a new Router instance.

```
// ================================
// Create new router instance
// ================================
var router = express.Router();
```

Inside the `module.exports` function, let's assign our routes to their controllers. Right now, the controller methods are just stubbed out. We'll implement them later.

```
// ================================
// Assign Controllers to Routes
// ================================
router.get('/', getAllUsers);
router.post('/', newUser);
router.get('/:id', getUser);
router.delete('/:id', deleteUser);
router.put('/:id', updateUser);
```

After mounting the endpoints onto the controllers, you need to tell our Express app to use the router.

```
// ================================
// Mount router instance to app
// ================================
app.use('/api/user', router);

return router;
```

So now the only thing remaining is actually implementing the details of each controller.

```
// ================================
// Controller function definitions
// ================================
function getAllUsers(req, res, next) {
  res.json({users: userService.getAllUsers()});
}
function newUser(req, res, next) {
  var firstName = req.body.firstName, lastName = req.body.lastName, email = req.body.email;
  if (!firstName || !lastName || !email) {
```

```
    return res.sendStatus(400);
  }
  var user = {
    firstName: firstName,
    lastName: lastName,
    email: email,
    id: userService.generateId()
  };
  userService.addUser(user);
  res.sendStatus(200);
}
function getUser(req, res, next) {
  var userId = req.params.id;
  if (!userId) {
    return res.sendStatus(400);
  }
  var user = userService.getUserById(userId);
  if (user) {
    res.json({success: true, user: user});
  } else {
    res.send({sucess: false})
  }
}
function deleteUser(req, res, next) {
  var userId = req.params.id;
  if (!userId) {
    return res.sendStatus(400);
  }
  var success = userService.deleteUser(userId);
  res.json({success: success});
}
function updateUser(req, res, next) {
  var userId = req.params.id;
  if (!userId) {
    return res.sendStatus(400);
  }
  var firstName = req.body.firstName, lastName = req.body.lastName, email = req.body.email;
  if (!firstName || !lastName || !email) {
    return res.sendStatus(400);
  }
  var user = {
    firstName: firstName,
    lastName: lastName,
    email: email,
    id: userId
  };
  userService.updateUser(user);
  res.sendStatus(200);
}
```

Random Number Generator

The `/api/randomnumber` route should return a random number between 1 and 100 when called. Use the same pattern that we did above in the user router to achieve this. Create a new route instance, `var router = express.Router()`. Then mount the `getRandomNumber()` controller on the `/api/randomnumber` **(get)** route.

Here is the controller definition for `getRandomNumber`. When the controller is called, it sends an HTTPS (post) request to a url at `www.random.org`. When the response `data` event occurs, it returns the value received to the client as a JSON object.

```
function getRandomNumber(req, res, next) {
  https.get('https://www.random.org/integers/?
num=1&min=1&max=100&col=1&base=10&format=plain&rnd=new', function(response) {
    response.on('data', function(chunk) {
      res.json({number: chunk.toString()});
    });
    response.on('error', function() {
      res.sendStatus(500);
    })
  });
}
```

Note Router

The note router has two endpoints:

- `/api/note` (get) : **getNote()**
- `/api/note` (post) : **addNote()**

The controllers for the endpoints will both use the `fs` module provided by node to write to and read from a file called note. Again, use the same pattern as above to create a new router, and map the routes to the controllers.

Here are the implementations for the `addNote()` and `getNote()` controllers.

```
function getNote(req, res, next) {
  var readableStream = fs.createReadStream('note');
  var note = "";

  readableStream.on('data', function(chunk) {
    note += chunk.toString();
  });

  readableStream.on("end", function() {
    res.json({note: note});
  });

  readableStream.on('error', function(e) {
    console.log(e);
    res.sendStatus(500);
  })
}
function addNote(req, res, next) {
  var stream = fs.createWriteStream("note");
  stream.once('open', function(fd) {
    stream.write(req.body.text+"\n");
    stream.end();
  });

  stream.on('close', function() {
```

```
    res.sendStatus(200);
  })

  stream.on('error', function() {
    res.sendStatus(500);
  })
}
```

Initialize Routes

Each of the router modules **(user, note, number)** returns a function that accepts the Express app as a parameter. This function needs to be called to initialize our routes.

In the `app.js`, require in your API routes and execute the return function by doing the following:

```
// ================================
// Init our API Routes
// ================================
require('./api/note')(app);
require('./api/number')(app);
require('./api/user')(app);
```

Forward all other routes

Finally, for any other `undefined` routes, return the `404.html` page in the public folder..

```
// ================================
// All other routes should
// send our 404 image
// ================================
app.route('/*')
  .get(function (req, res, next) {
    res.sendFile(path.join(__dirname, '/public/404.html'))
  });
```

Startup the App

All that there is left to do is tell the Express app to start listening to the port of our choosing.

```
// ================================
// Tell our express app to listen
// to the port 3030
// ================================
app.listen(3030);
```