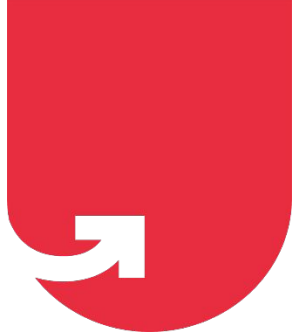
upGrad

#RahoAmbitious



Full Stack Software Development

upGrad

Course: Server-Side Development Using Node.js, Express.js and MongoDB

Lecture on: Setting Up Node.js Application With Express.js

Instructor: Rocky Jagtiani

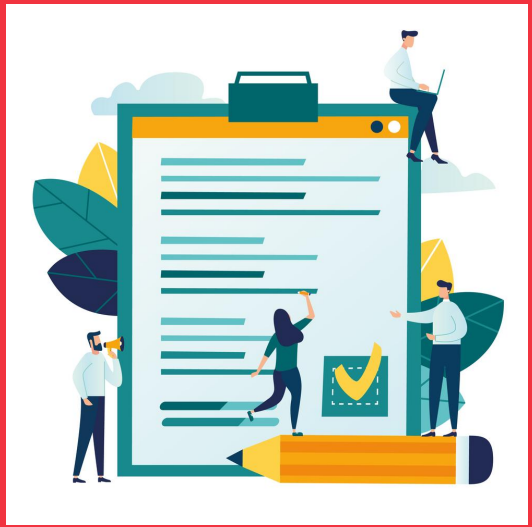
upGrad

In the previous class, we covered...

- Mongoose Relationship Models

Today's Agenda

- Introduction to Express.js
- Routing in Express.js
- Request Objects
- Router Objects
- Routing Functions
- Router Path
- Route Methods
- Cross-Origin Requests



Express.js

Express.js

upGrad

- Previously we learnt how web servers work with Node.js and MongoDB
- Now, we will learn about a web framework
- A web framework is a pre-defined application structure and a library of development tools, to make building a web application easier and more consistent
- Express.js is one of the most popular Node.js web frameworks

Express.js

upGrad

Few other popular Node.js frameworks:

- **Koa.js** designed by developers who built Express.js with a focus on a library of methods not offered in Express.js (http://koajs.com/)
- **Hapi.js** designed with a similar architecture to Express.js and a focus on writing less code (https://hapijs.com/)
- **Sails.js** built on top of Express.js, offering more structure, as well as a larger library and less opportunity for customization (https://sailsjs.com/)
- **Total.js** Built on the core HTTP module and acclaimed for its high-performance request handling and responses (https://www.totaljs.com/)

Express.js

upGrad

- Any Express.js application has four key parts:
  - The require statement
  - Middleware
  - Routing
  - app.listen() to start the server

Express.js

upGrad

- To get started with Express.js, you need to install it
- Create a folder called express-app, and in the folder initialise the project using
  npm init
- Install express.js in the project using
  npm install --save express

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello from Suven !')
})

app.listen(port, () => {
  console.log(`Example app listening at
http://localhost:${port}`)
})
```

Refer To: express_app/server.js

Express.js as Middleware

upGrad

- A web framework is designed to do a lot of tedious tasks for us hence giving us a simple structure for customizing our app
- Express.js provides a way to listen to the requests of specific URLs and respond by using a callback function
- A web framework like Express.js operates through functions considered to be middleware because they sit between HTTP interaction on the web and the Node.js platform

Express.js as Middleware

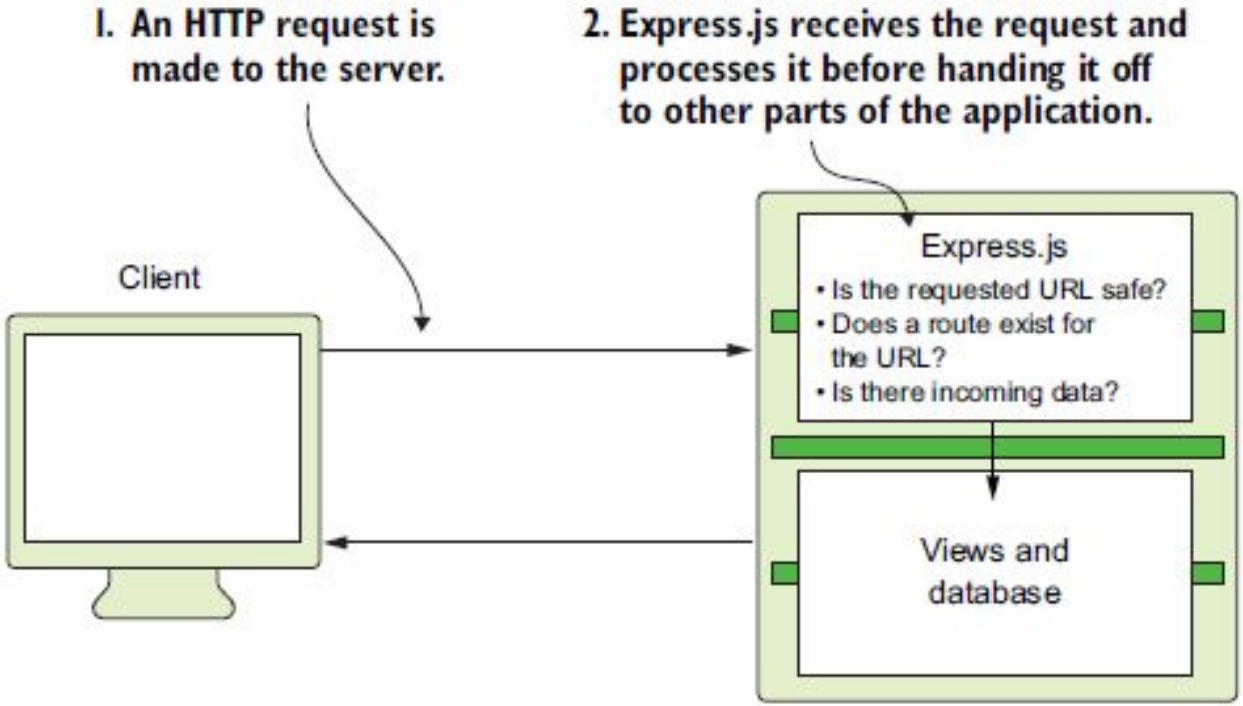
upGrad

- Middleware is a general term applied to code that assists in listening for, analyzing, filtering and handling HTTP communication before data interacts with application logic

- You can consider middleware as a courier company(like Bluedart). Before the package can go into the delivery network, a courier collector needs to inspect the size of the box to ensure that it’s properly paid for and adheres to delivery policies

Express.js as Middleware

upGrad



Express.js stands between the HTTP requests and your application code.

upGrad

Routing in Express.js

Routing in Express.js

upGrad

- Routing means assigning functions to respond to users' requests
- Express routers are basically **middleware** (*meaning they have access to the request and response objects and do lots of work for us. Hence codes written using Express.js is shorter and more readable than plain Node.js*)
- Routing in Express.js follows this basic format:

```
app.VERB('path', callback...);
```

Here, VERB can be either GET, POST, PUT, or DELETE

Refer To: express_app > example1.js

Routing in Express.js

upGrad

- We can add as many callbacks as we desire
- In example1.js, you would notice sayHello function is fired before the response is sent to the browser. The sayHello function takes three arguments (request, response, and next). The next() function, when called, moves control to the next middleware or route
- call to next() as the last step in the callback function is a necessary step, not optional

Calling next at the end of your function is necessary to alert Express.js that
your code has completed.

Not doing so leaves your request hanging. Middleware runs sequentially, so by not calling next, you're blocking your code from continuing until completion.

Refer To: express_app > example1.js

Request Object

Request Object

upGrad

- The request object contains information about the incoming request
- The most useful properties of request object are:
  - **request.params:** holds all GET parameters
  - **request.body** stores POST form parameters
  - **request.query** property is used to extract the GET form data
  - **request.headers** hold key/value pairs of the request received by the server. Servers and clients make use of headers to communicate their compatibility and constraints
  - **request.accepts(['json','html'])** holds an array of data formats and returns the the browser preferred format of data
  - **request.url** stores data about the URL of the request
  - **request.ip**: holds the IP (Internet Protocol) address of the browser requesting for information

Request Object

upGrad

A query string is text represented as key/value pairs in the URL following a question mark(?) after the hostname.

For example: http://localhost:3000?name=suven
here, you are sending the *name* (key) paired with *suven* (value). This data can be extracted and used in the route handler.

Request Object

upGrad

- The ***request.params*** variable stores *GET* request parameters

```
const express = require('express')
const app = express()
const port = 3000


app.get('/:name/:age', (request, response) => {
    response.send(request.params);
});


//Binding the server to a port(3000)
app.listen(3000, () => console.log('express server
started at port 3000'));
```

- Run the given code and on the browser hit, https://localhost:3000/suven/17

Request Object

upGrad

- The **colon** before the parameters differentiates the route parameters from normal route path
- Route parameters are handy for specifying data objects in our application. When you start saving user accounts and course listings in a database, for example, you might access a user's profile or specific course with the /users/:id and /course/:type paths, respectively. This structure is necessary for developing a representational state transfer **(REST)** architecture

```
const express = require('express')
const app = express()
const port = 3000


app.get('/:name/:age', (request, response) => {
    response.send(request.params);
});


//Binding the server to a port(3000)
app.listen(3000, () => console.log('express server
started at port 3000'));
```

Request Object

upGrad

- The request.query property is used to extract the GET form data

```
<!-- form.html -->
<!-- action specifies that form be handled on the same page -->
<form action='/' method='GET'>
  <input type='text' name='name' placeholder='ur name'/>
  <input type='email' name='email' placeholder='ur email'/>
  <input type='submit'/>
</form>
```

Request Object

upGrad

```
// server side code for express-query example

// loading express
const express = require('express');
app = express();

//route serves both the form page and processes form data
app.get('/', (request, response)=>{
  console.log(request.query);
  response.sendFile(__dirname +'/form.html');
});


app.listen(3000,() => console.log('Express server started at port 3000'));
```

- Run the code with node server.js, hit localhost:3000, and fill and submit the form in your browser. After submitting the form, the data you filled out gets logged to the console

Refer To: express-query

Request Object

upGrad

- There are multiple ways to send data from client to server:
  - query strings: **?(symbol)** is used to concat key=value pairs to the URL
  - URL parameters: **:(symbol)** is used to concat

- However, HTTP protocol provides more ways to pass information from a client to a server
- Sending information through HTTP POST body is very common
- **request.body** is used to extract the POST parameters

Request Object

upGrad

- Data can be sent using HTTP POST call via an HTML <form> or an API request. Such data can take on a few different forms like:
  - application/x-www-form-urlencoded: Data encoded in this format is seen as the query string in a URL
    example: *course=mern&session=1&platform=upgrad*

    ***Note**: This is the default encoding*

  - multipart/form-data: This encoding is used for sending files

  - text/plain: This data is just sent as unstructured plain text. This approach is not used much

Request Object

upGrad

- The ***request.body*** property contains key-value pairs of data submitted in the request body
- By default, it is undefined and is populated when we use a middleware called body-parser. Such as ***bodyParser.urlencoded()*** or ***bodyParser.json()***

Note*: The body-parser package, can handle many forms of data. This package is a middleware. It intercepts the raw body and parses it into a form that our application code can easily use.*

```
Method : POST,
URL : http://localhost:3000/post-test
JSON body
{
   "username": "rocky" ,
   "password": "rocky123" ,
   "website": "abc.edu.in"
}

USE Swagger or POSTMAN
```

The request object – request.body property

upGrad

Notice, we call app.use(...) before defining our route. The order matters. This will ensure that the body-parser will run before our route, which ensures that our route can then access the parsed HTTP POST body.

Refer : express-post > post1.js

Refer : express-post > post2.js

Router Object

Router Object

upGrad

- Express.js router, allows us to break our application into fragments that can have their own instances of express to work with. We can then bring them together in a very clean and modular way.

- Consider following URLs:
  - localhost:3000/myorders/prachi
  - localhost:3000/cart/addtocart
  - localhost:3000/cart/removefromcart
  - localhost:3000/products
  - localhost:3000/products/laptop

Router Object

upGrad

```
//Different routes
app.get('myorders/prachi',(request,response)=>{
    response.send(`Prachi past orders page`);
});

app.get('/cart/addtocart',(request,response)=>{
    response.send(`Some Code here to add product to cart collection in MongoDB`);
});

app.get('/cart/removefromcart',(request,response)=>{
  response.send(`Some Code here to delete product from cart collection in MongoDB`);
});

app.get('/products',(request,response)=>{
  response.send(`Some Code here to fetch all product categories from product collection in MongoDB`);
});

app.get('/products/laptop',(request,response)=>{
  response.send(`Some Code here to fetch only products matching with laptop from product collection in
MongoDB`);
});
```

Router Object

upGrad

- There's nothing wrong with this pattern. But it has potential errors
- When our routes are basically just five or six, there isn't much of a problem. But when things grow and lots of functionality required(like in an e-commerce web app), putting all that code in our server.js is not the best practice

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

router.get('myorders/prachi',(request,response)=>{
    response.send(`Prachi past orders page`);
});

router.get('/cart/addtocart',(request,response)=>{
    response.send(`Some Code here to add product to
cart collection in MongoDB`);
});

// similar more routing code comes here.

//exporting the router to other modules
module.exports = router;
```

Router Object

upGrad

- In such cases, we should use Express.js router for doing the routing in a separate router.js file
- The given code snippet does a similar routing job. Here the routing code is shifted from the server.js to router.js
- In this way, only minimal necessary business logic remains in server.js and all routing code is shifted to router.js file. This helps in easy maintainability and debugging

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

router.get('myorders/prachi',(request,response)=>{
    response.send(`Prachi past orders page`);
});

router.get('/cart/addtocart',(request,response)=>{
    response.send(`Some Code here to add product to
cart collection in MongoDB`);
});

// similar more routing code comes here.



//exporting the router to other modules
module.exports = router;
```

Refer To: 2_react-router

Poll 1 (15 sec)

Which parameter contains contents of the request, like data coming from a POST request, such as a submitted form. Usually we collect this information and save it in a database.

1. request.params
2. request.body
3. request.url
4. request.query

Poll 1 (Answer)

Which parameter contains contents of the request, like data coming from a POST request, such as a submitted form. Usually we collect this information and save it in a database.

1. request.params
2. **request.body**
3. request.url
4. request.query

Poll 2 (15 sec)

Web frameworks make development work a lot easier. Web development is fun, and the best parts aren't the tedious tasks that are most subject to errors. With web frameworks, developers and businesses alike can focus on the more interesting parts of applications.

1. True
2. False

Poll 2 (Answer)

Web frameworks make development work a lot easier. Web development is fun, and the best parts aren't the tedious tasks that are most subject to errors. With web frameworks, developers and businesses alike can focus on the more interesting parts of applications.

1. **True**
2. False

Poll 3 (60 sec)

What happens when a request is made to */items/lettuce*?

1. the request is processed first by our middleware function and then by the app.get("/items/:vegetable") route. The output on the console is **C**, **B** and then **A**.

2. the request is processed first by app.get("/items/:vegetable") route and then by the middleware function. The output on console come as **C**, then we see **A** on the browser and at last we see **B** on the console.

```
const port = 3000,
express = require("express"),
app = express();

app.get("/items/:vegetable", (req, res) => {
  let veg = req.params.vegetable;
  res.send(`This is the page for ${veg}`);  // A
});

// this is the middleware function
app.use((req, res, next) => {
  console.log(`request made to: ${req.url}`); // B
  next();
});

app.listen(port, () => {
  console.log(`Server running on port: ${port}`); // C
});
```

Poll 3 (60 sec)

What happens when a request is made to ***/items/lettuce***?

1. **the request is processed first by our middleware function and then by the app.get("/items/:vegetable") route. The output on the console is C, B and then A.**

2. the request is processed first by app.get("/items/:vegetable") route and then by the middleware function. The output on console come as **C**, then we see **A** on the browser and at last we see **B** on the console.

```
const port = 3000,
express = require("express"),
app = express();

app.get("/items/:vegetable", (req, res) => {
  let veg = req.params.vegetable;
  res.send(`This is the page for ${veg}`);  // A
});

// this is the middleware function
app.use((req, res, next) => {
  console.log(`request made to: ${req.url}`); // B
  next();
});

app.listen(port, () => {
  console.log(`Server running on port: ${port}`); // C
});
```

upGrad

Routing Function

Routing Function

upGrad

- There is a special routing method, ***app.all()***, used to load middleware functions at a path for all HTTP request methods
- The below handler is executed for requests to the route **"/secret"** whether using GET, POST, PUT, DELETE, or any other HTTP request method supported in the *HTTP module*

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next()  // pass control to the next handler
})
```

Route Path

Route Path

upGrad

- Route paths, in combination with a request method, define the endpoints at which requests can be made
- Route paths can be strings, string patterns, or regular expressions

Examples of route paths based on strings.

```
//This route path will match requests to the root route, /.
app.get('/', function (req, res) {
  res.send('root')
})


//This route path will match requests to /aboutus.
app.get('/aboutus', function (req, res) {
  res.send('about us page')
})


//This route path will match requests to /Bank.accountNumber.
app.get('/Bank.accountNumber', function (req, res) {
  res.send('random.number')
})
```

Route Path

upGrad

- The characters ?, +, *, and () as used in regular expression can be used in route paths as well
- The hyphen (-) and the dot (.) are interpreted literally by string-based paths
- If you need to use the dollar character ($) in a path string, enclose it escaped within ([and]). For example, the path string for requests at "/data/$laptop", would be "/data/([\$])laptop"

Examples of route paths based on strings.

```
//This route path will match requests to the root route, /.
app.get('/', function (req, res) {
  res.send('root')
})


//This route path will match requests to /aboutus.
app.get('/aboutus', function (req, res) {
  res.send('about us page')
})


//This route path will match requests to /Bank.accountNumber.
app.get('/Bank.accountNumber', function (req, res) {
  res.send('random.number')
})
```

Route Path

upGrad

Examples of route paths based on string patterns:

```
// the character before ? is optional.   This route path will match acd and abcd.
app.get('/ab?cd', function (req, res) {
  res.send('ab?cd')
})
```

```
// + means 1 or more.  This route path will match abcd, abbcd, abbbcd, and so on.
app.get('/ab+cd', function (req, res) {
  res.send('ab+cd')
})
```

Route Path

upGrad

Examples of route paths based on string patterns:

```
// * means 0 or more.  This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.
app.get('/ab*cd', function (req, res) {
  res.send('ab*cd')
})
```

```
// ()?  means optional.   This route path will match /abe and /abcde.
app.get('/ab(cd)?e', function (req, res) {
  res.send('ab(cd)?e')
})
```

Route Path

upGrad

Hyphen(-) and the **dot (.)** are interpreted literally, they can be used along with route parameters for useful purposes

Route path: /flights/:from-:to
Request URL: http://localhost:3000/flights/MUM-DEL
req.params: { "from": "MUM", "to": "DEL" }

Route path: /plantae/:type.:species
Request URL: http://localhost:3000/plantae/Iris.setosa
req.params: { "type": "Iris", "species": "setosa" }

Route Path

upGrad

- To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses (()):

Route path: /students/:studentId(\d+)
Request URL: http://localhost:3000/students/12
req.params: {"studentId": "12"}

Route Methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded
res.end()	End the response process
res.json()	Send a JSON response
res.jsonp()	Send a JSON response with JSONP support.

upGrad

Method	Description
res.redirect()	Redirect a request
res.render()	Render a view template
res.send()	Send a response of various types
res.sendFile()	Send a file as an octet stream
res.sendStatus()	Set the response status code and send its string representation as the response body.

Hands-on Exercise 1 (20 mins)

upGrad

- Create a project folder named express_practice_1
- Initialise the project
  npm init
- Install express if required
  npm install --save express
- Code a middleware route to handle the below get requests :
  - for get request http://localhost:3000/999 we should see the following response
    The id you specified is 999
  - for get request http://localhost:3000/mern_stack/12 we should see the following response
    id: 12 and enrolled in : mern_stack

Hands on Exercise 1 (20 mins)

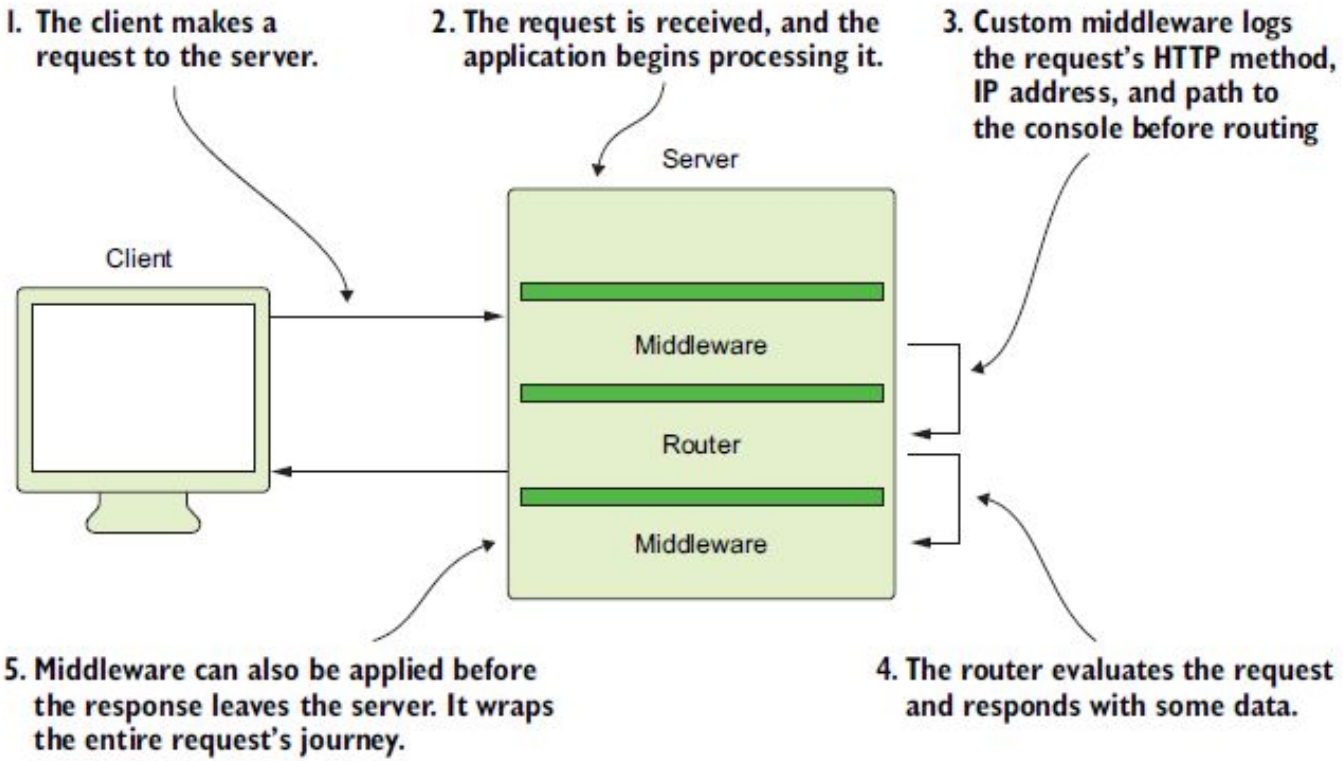
upGrad

- for get request http://localhost:3000/12345 we should see the following response
  id: 12345
- for all other invalid URL's, show a message on the browser as
  Sorry, this is an invalid URL

Refer To: express_practice_1

Summary

upGrad



See complete list of Middleware functions here: https://expressjs.com/en/resources/middleware.html

Poll 4 (15 sec)

Your friend has defined a middleware function as:

```
app.use("/items", <callback>),
```

This middleware function would:

1. Run for every request made to a path starting with items.
2. Result in an error, as the path is incomplete.

Poll 4 (Answer)

Your friend has defined a middleware function as:

```
app.use("/items", <callback>),
```

This middleware function would:

1. **Run for every request made to a path starting with items.**
2. Result in an error, as the path is incomplete.

Poll 5 (45 sec)

Given the following specifications

```
//Route path:
/students/:studentId/books/:bookId

//middleware route definition is :
app.get('/students/:studentId/books/:bookId',
function (req, res) {
  res.send(req.params)
})
```

What will be output, if the request URL is:

```
http://localhost:3000/students/23/books/9767
```

1. req.params: { "studentId": "23", "bookId": "9767" }
2. req.params: { "23", "9767" }
3. { "studentId": "23", "bookId": "9767" }
4. { "23", "9767" }

Poll 5 (45 sec)

Given the following specifications

```
//Route path:
/students/:studentId/books/:bookId

//middleware route definition is :
app.get('/students/:studentId/books/:bookId',
function (req, res) {
  res.send(req.params)
})
```

What will be output, if the request URL is:

```
http://localhost:3000/students/23/books/9767
```

1. req.params: { "studentId": "23", "bookId": "9767" }
2. req.params: { "23", "9767" }
3. **{"studentId": "23", "bookId": "9767"}**
4. { "23", "9767" }

Cross-Origin Requests

CORS

upGrad

- If we send a fetch request to another website, it will probably fail
- For instance, let's try fetching http://amazon.com/laptops from http://upgrad.com:

```
try {
  await
  fetch('http://amazon.com/laptops');
}
catch(err) {
  alert(err);    // Failed to fetch
}
```

CORS

upGrad

- Cross-origin requests are sent to another domain (even a subdomain) or protocol or port and require special headers from the remote side

- That policy is called "**CORS**": Cross-Origin Resource Sharing

CORS

upGrad

- CORS stands for Cross-Origin Resource Sharing
- It is a mechanism to allow or restrict requested resources on a web server to be accessed from a different origin
- This policy is used to secure a certain web server from access by other website or domain. For example, only the allowed domains will be able to access hosted files in a server such as a stylesheet, image, or a script
- If you are currently on **http://upgrad.com/page1** and you are referring to an image from **http://shuttershock.com/myimage.jpg** you won't be able to fetch that image unless **http://shuttershock.com/** allows cross-origin sharing with **http://upgrad.com/**

CORS

upGrad

- There is an HTTP header called origin in each HTTP request. It defines from where the domain request has originated. We can use header information to restrict or allow resources from our web server to protect them.

- For example, if you are using a frontend React, your front end application will be served on http://localhost:3001. Meanwhile, your Express server might be running on a different port such as http://localhost:3000.

- Because of this, you'll need to allow CORS between those servers.

CORS

upGrad

- Install cors module

```
projectFolder> npm install cors
```

- require it in the Code file

```
const cors = require('cors');
```

- If you wish to enable CORS for all requests

```
app.use(cors());
```

- If you wish to enable CORS for only specific requests

```
app.get('/someURL/', cors(),
function (req, res, next) {
  // handle and respond
})
```

upGrad

- To set the CORS options and use it, do like this

```
var corsOptions = {
    origin: 'http://localhost:3000',
    optionsSuccessStatus: 200, // For legacy browser support
    methods: "GET, PUT"  // would allow only GET and PUT request
}
app.use(cors(corsOptions));
```

Refer To: enableAllCORSRequestExample.js

Refer To: corsOptionsExample.js

Refer To: enableSingleCORSRequestExample.js

upGrad

Project Work - Checkpoint 5

Checkpoint 5

upGrad

- We would now add code to server.js file
- Load express and create an express app object
- Load the body-parser module, so that we can read ***request.body*** parameters from an HTTP POST request
- Load cors module
  Note: *node-express APIs are going to be called from React/ POSTMAN/Swagger. CORS allows another application to call our APIs.*
- Set the default route for the index or root path

```
// simple route
app.get("/", (req, res) => {
  res.json({ message: "Welcome to Upgrad InSession application development."});
});
```

- Set the PORT and start the server (i.e LISTEN on PORT for request)

Checkpoint 5

upGrad

Code the ***app/controllers/tutorial.controller.js*** for the following APIs:

- create() - to create and save the course
- findAllTitle() - to search the course by title
- findAllCategory() - to get all course categories distinctly
- findCoursesByCategory - to search the course by category
- findOne() - to fetch all details of a course given its _id
- update() - to update one or more details of a course given its _id
- delete() - to delete a course given its _id
- deleteAll() - to delete all courses
- findAllPublished() - to fetch all Courses with published parameter as true

Refer To: **server.js** and **app/controllers/tutorial.controller.js**

Checkpoint 5

upGrad

Code the ***app/controllers/user.controller.js*** for following APIs

- signUp() - to create a USER object and save it in USER schema
- login() - to check the entered email_id and password is matching with data in USER schema. If yes, then the person has logged in
- logout() - This requires the unique Id of the logged in person. His logged in status is set to false

Refer **server.js** and **app/controllers/user.controller.js**

Checkpoint 5

upGrad

Code the ***app/controllers/enrollment.controller.js*** for following APIs

- enroll() - to record an enrollment, when a student registers/enrolls for a course

Refer server.js and app/controllers/enrollment.controller.js

Homework 1

upGrad

In this assignment you are given a book_Author_Route.js file. There are multiple routes defined using GET, POST or PUT method types in the given file.

Your task is to identify the purpose of each route, for those routes which are marked as "??"

Refer To: book_Author_Routes.js

Homework 2

upGrad

Code a simple script which does the following:

- Collects book and author information like bookName, AuthorName, Pages and Price through a simple html file.
- Serves the above html file on the root route.
- When the user submits (using POST) the html file, express route should fetch bookName and AuthorName from the request body and respond the same on the browser.

Refer To: book_server.js and views/index.html

Doubt Clearance (5 mins)

Key Takeaways

- We got introduced to the concept of Express.js and learnt about its applications.

The following tasks are to be completed after today's session:

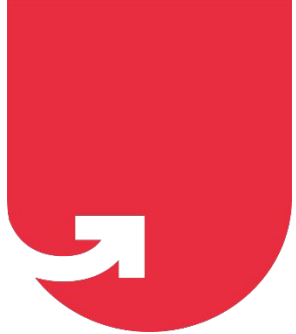
Homework
MCQs
Coding Questions
Course Project - Checkpoint 6

In the next class, we will discuss...

- We will learn about Express.js modules

upGrad

#RahoAmbitious



Thank you!