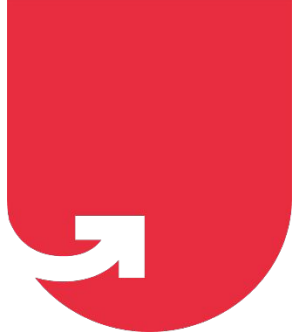
upGrad

#RahoAmbitious



Full Stack Software Development

upGrad

Course: Server-Side Development Using NodeJS, Express and MongoDB

Lecture on: Routes serving static web applications

Instructor: Rocky Jagtiani

In the previous class...

- You learnt about web servers and understood why Node.js applications are built on web servers
- You saw how servers handle the incoming data
- You were introduced to the concept of routes
- You understood the problem statement and architecture of the project

Poll 1 (15 Sec)

A route is a way of determining how an application should respond to a request made to a specific URL.

Is the above statement true or false?

1. True
2. False

Poll 1 (Answer)

A route is a way of determining how an application should respond to a request made to a specific URL.

Is the above statement true or false?

1. **True**
2. False

Poll 2 (15 Sec)

The mapping of routes to responses is called _____

1. routeResponseMap
2. routeMap
3. responseRouteMap
4. None of the above

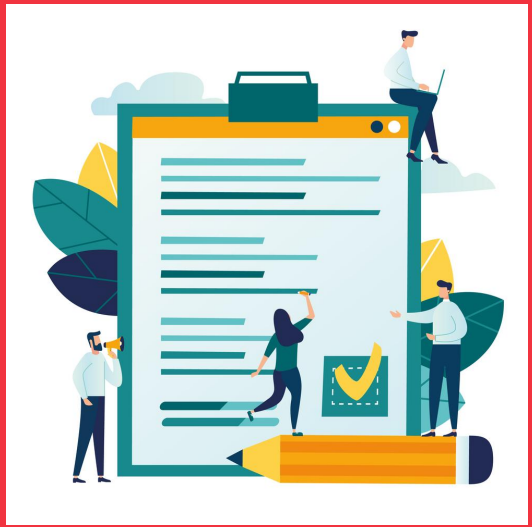
Poll 2 (Answer)

The mapping of routes to responses is called _____

1. **routeResponseMap**
2. routeMap
3. responseRouteMap
4. None of the above

Today's Agenda

- Core Module
- fs Module
- fs Module APIs
- Model-View-Controller
- Project Checkpoint 2



Modules/Packages

upGrad

There are three types of modules/packages in Node.js:

- **Custom modules/packages:** These modules/packages are created and defined by the user. You need not install a custom module. However, you need to use the ***require*** function to import a custom module by providing its path to start using it.

- **Third-party modules/packages:** These modules/packages are provided by Node Package Manager. They have already been created by someone else for you. You need to install a third-party module via NPM and import the module using the ***require*** function while providing the name of the module as the ID to start using it

- **Core modules:** These modules are provided by Node by default; you need not define them. Also, these are not required to be installed prior to their usage. You can simply use the ***require*** function to import a core module to start using it

Core Modules

Core Modules

upGrad

- So far, you have learnt about the third-party modules that can be used after installing them using npm
- Now, let’s take a look at some of the modules that do not require to be installed separately and can be used directly. Such modules are called 'Core Modules'
- Core modules are available by default in each Node.js module
- Some commonly used core modules in Node.js are as follows:
  - console
  - buffer
  - events
  - http/https
  - fs

Core Modules

upGrad

- To view the list of all the core modules, you can use the following command:

```
console.log(require('module').builtinModules);
```

- This command will return as an array containing all the Node.js core modules

fs Module

fs Module

upGrad

- Here, fs stands for 'File System'
- This module provides the capability of local file management, which is not provided by the client-side JavaScript
- The fs module offers APIs for interacting with the file system
- All the file system operations in Node.js follow synchronous as well as asynchronous forms
  - The ***asynchronous*** form takes a completion ***callback*** as its last argument, which is invoked when the asynchronous file operation finishes its operation. The first argument of this completion callback is reserved for the exception, which is assigned the value ***null*** or ***undefined*** when the file operation is successful
  - The ***synchronous*** form is written inside the ***try-catch*** block, and if an error or exception occurs, it is handled by the ***catch*** block

Poll 3 (15 Sec)

Which of the following commands allows us to import the fs module in our application?

1. const fs = module('fs');
2. const fs = require('fs');
3. const fs = builtinModule('fs');
4. All of the above

Poll 3 (Answer)

Which of the following commands allows us to import the fs module in our application?

1. const fs = module('fs');
2. **const fs = require('fs');**
3. const fs = builtinModule('fs');
4. All of the above

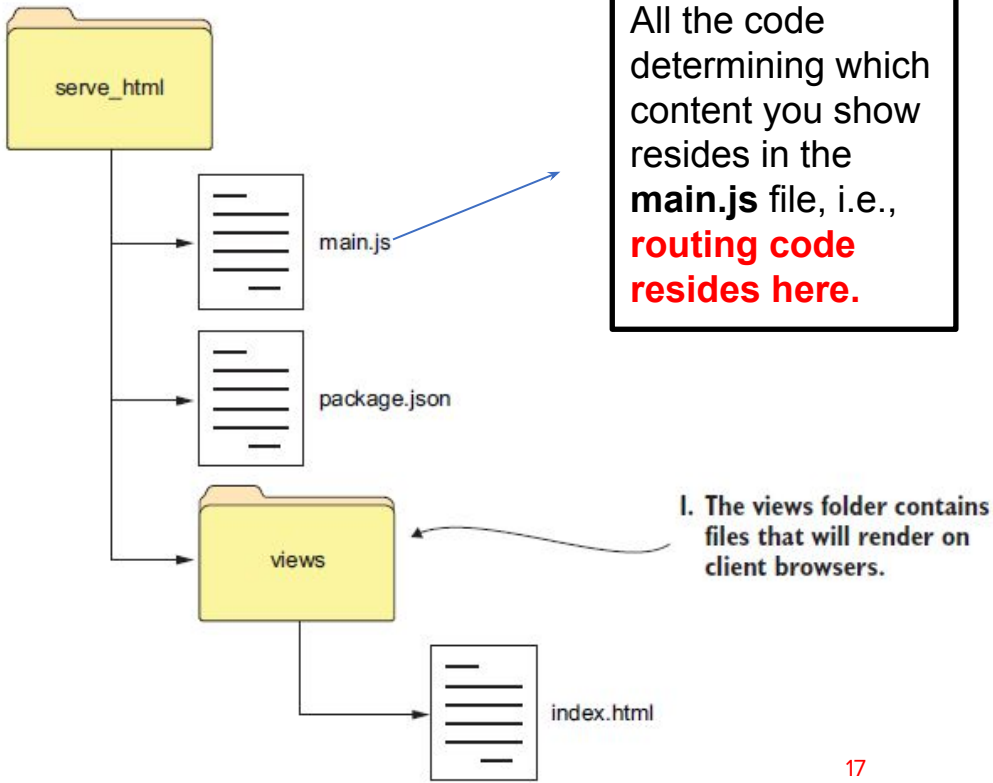
fs Module

upGrad

In the previous session, we directed the URL traffic with a routing system that matched the request URLs to custom responses. However, our responses were only for a few lines of plain HTML, i.e., one-liner responses

In this session, you will learn how to serve the entire HTML file. In the next session, you will learn how to serve HTML files along with its CSS and JavaScript files

To serve static HTML files, we will use the fs core module



All the code determining which content you show resides in the **main.js** file, i.e., **routing code resides here.**

fs Module

upGrad

- The fs module offers APIs for interacting with the file system
- The most common fs module APIs can perform the following operations:
  - Read files
    - fs.readFile()
  - Create files
    - fs.appendFile()
    - fs.open()
    - fs.writeFile()
  - Update files
    - fs.appendFile()
    - fs.writeFile()
  - Delete files
    - fs.unlink()
  - Rename files
    - fs.rename()

fs.readFile()

fs.readFile()

upGrad

- The fs.readFile() method is used to asynchronously read files on your computer
- The following syntax is used for the fs.readFile() method:

```
fs.readFile(path[, options], callback)
```

- path <string> | <Buffer> | <URL> | <integer>: filename or file descriptor
    - options <Object> | <string>
        - encoding <string> | <null> **Default:** null
        - flag <string> See support of file system flags. **Default:** 'r'.
        - signal <AbortSignal> allows aborting an in-progress readFile
    - callback<function>
        - err <Error>
        - data <string> | <Buffer>

fs.readFile()

upGrad

Sample Code Snippets:

```
import { readFile } from 'fs';

readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

```
import { readFile } from 'fs';

var http = require('http');
var fs = require('fs');

http.createServer(function (req, res) {
  fs.readFile('demofile.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(port);
```

https://stackoverflow.com/questions/36795819/when-should-i-use-curly-braces-for-es6-import

fs.readFile()

upGrad

```
const routeMap = {
  "/": "views/index.html"
};

http
  .createServer((req, res) => {
    res.writeHead(httpStatus.OK, {
      "Content-Type": "text/html"
    });
    if (routeMap[req.url]) {
      fs.readFile(routeMap[req.url], (error, data) => {
        res.write(data);
        res.end();
      });
    } else {
      res.end("<h1>Sorry, not found.</h1>");
    }
  })
  .listen(port);
console.log(`The server has started and is listening
➥ on port number: ${port}`);
```

Set up route mapping for HTML files.

Read the contents of the mapped file.

Respond with file contents.

When the files on your computer are being read, the files could be corrupt, unreadable, or missing

Our code does not necessarily detect such errors before it executes. So, if something goes wrong, ***we should expect an error as the first parameter in the callback function***

Poll 4 (15 Sec)

What happens if you try to read a file that does not exist on your computer?

1. httpStatus.StatusCodes.OK response is returned

2. httpStatus.StatusCodes.NOT_FOUND response is returned

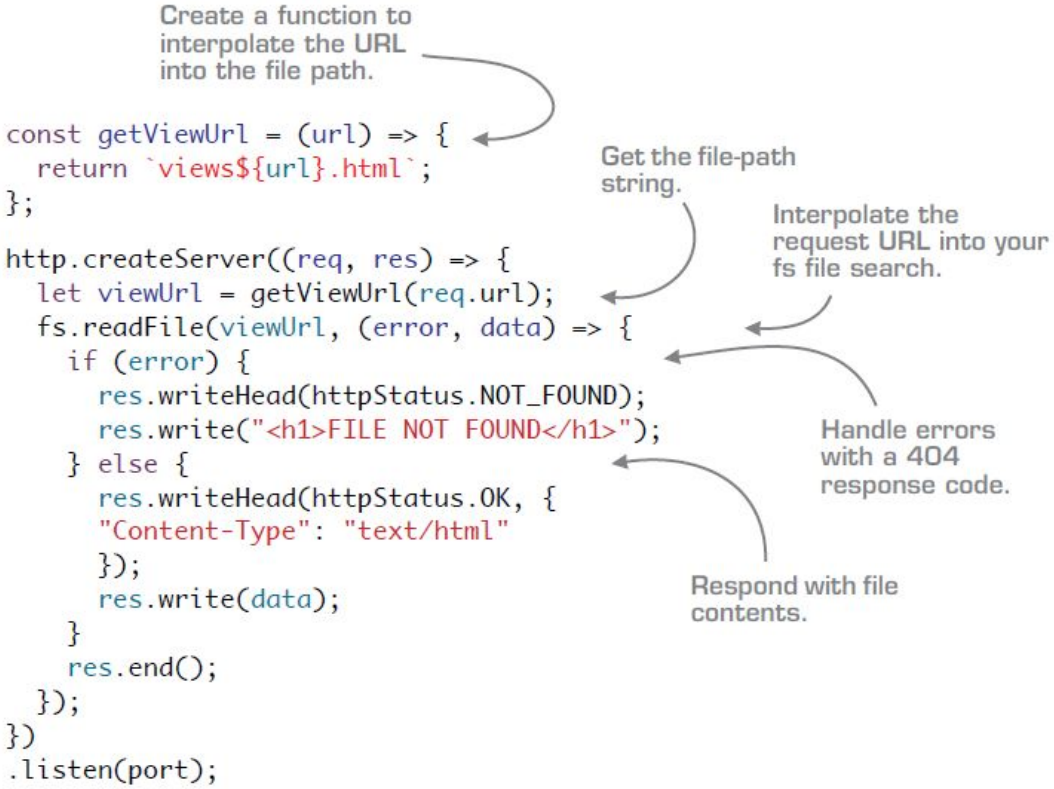
Poll 4 (Answer)

What happens if you try to read a file that does not exist on your computer?

1. httpStatus.StatusCodes.OK response is returned

2. **httpStatus.StatusCodes.NOT_FOUND response is returned**

fs.readFile()




```
const getViewUrl = (url) => {
  return `views${url}.html`;
};

http.createServer((req, res) => {
  let viewUrl = getViewUrl(req.url);
  fs.readFile(viewUrl, (error, data) => {
    if (error) {
      res.writeHead(httpStatus.NOT_FOUND);
      res.write("<h1>FILE NOT FOUND</h1>");
    } else {
      res.writeHead(httpStatus.OK, {
      "Content-Type": "text/html"
      });
      res.write(data);
    }
    res.end();
  });
})
.listen(port);
```

If someone visits the /index path, for example, getViewUrl returns views/index.html

If the file does not exist in the views folder, this command will fail, responding with an error message and the ***httpStatus.StatusCodes.NOT_FOUND*** code

Hands-On Exercise 1 (20 mins)

upGrad

- Create a project to serve four static HTML files:
  - index.html
  - contactUs.html
  - thankyou.html
  - aboutus.html

- You need to build this project in a phased manner:
  - Phase 1: Using the fs module to serve only the index.html file
  - Phase 2: Using the fs module and routing to dynamically read and serve contactUs.html, thankyou.html, aboutus.html files in main.js

***Note**: The aboutus.html file may contain images, which will be seen as broken Images. The reasons for this will shared after the completion of the Hands-on Exercise*

Phase 1

- Create a project folder called ***serve_html***
- Create a main.js file:
  - Declare the fs module
  - Define routeMap only for the index.html page
  - Check the req.url and accordingly serve the index.html page
- Create a folder named ***views*** within ***serve_html***
- Inside the views folder, create an ***index.html*** file

Using the **fs** module in server responses in **main.js**

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  fs = require("fs");
```

Require the fs module.

Expected Output:

When you access **http:// localhost:3000**, you should see your index.html page being rendered

Importan : The client can view this page being rendered in a browser only with the help of another Node.js core module: fs, which interacts with the filesystem on behalf of your application

Using the fs module, your server can access and read your index.html file and other html files as well

Phase 2

upGrad

In Phase 1, you served only one HTML file, i.e., index.html. In **Phase 2,** you need to enhance your code such that it should dynamically create the URL depending on the user's request

Example: If someone visits http://localhost:3000/sample.html, your code grabs the request's URL, /sample.html, and appends it to views to create one string: views/sample.html

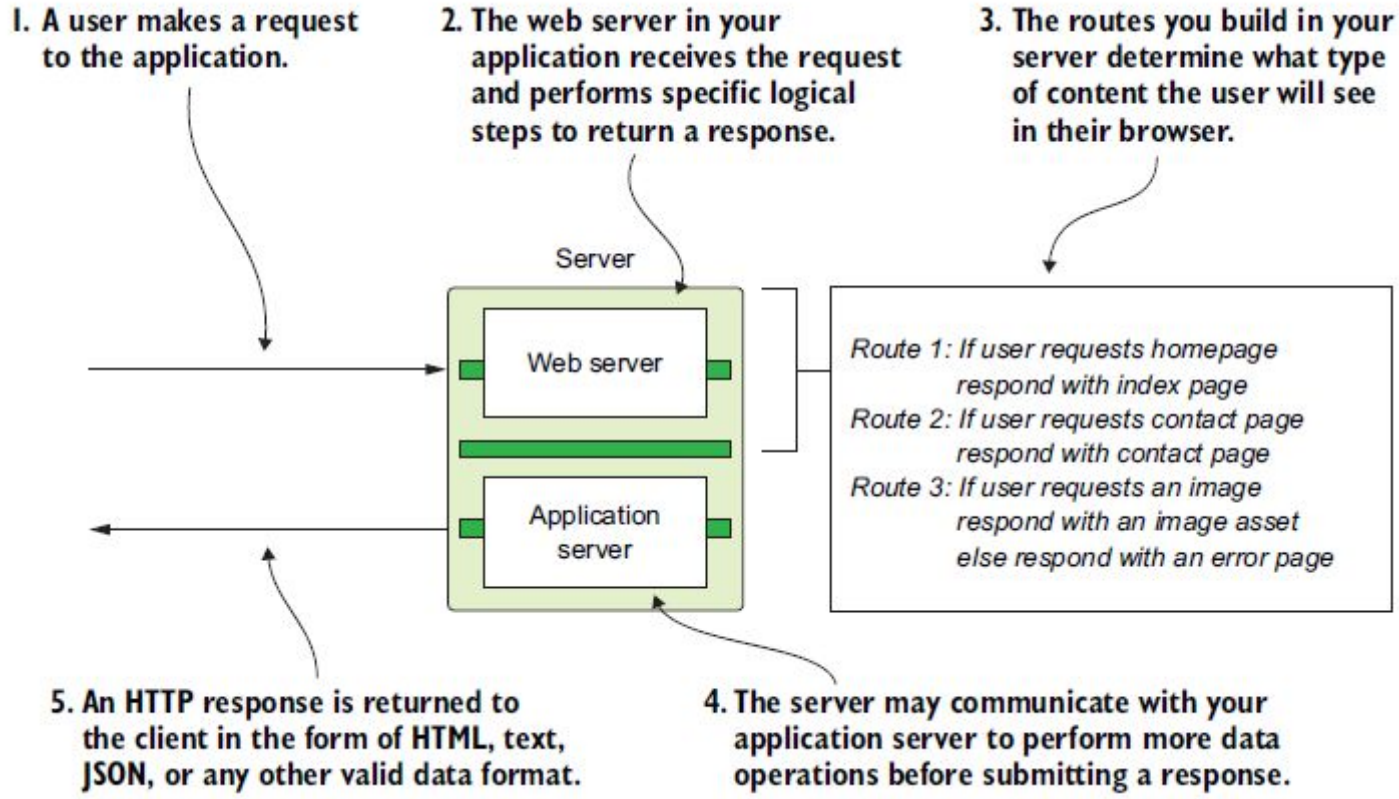
Now, edit the ***main.js*** file in ***serve_html*** as follows:

- Comment the routing and create-server code of Phase 1 *i.e., from the const* ***routeMap*** *to* ***.listen(port)***
- Create a new getViewUrl function to take the request's URL and interpolate it into the view file's path
- Replace the hardcoded filename in **fs.readFile** with the results from the call to getViewUrl

Expected Output:
For http:// localhost:3000/index,
the server will look for the URL at views/index and serve the page, and for http:// localhost:3000/randomPage, the server should server a HTML page with the message 'FILE NOT FOUND'

Refer: server_html

Routing Using the fs Module



Poll 5 (15 Sec)

State whether the following statement is true or false.

Like HTML files, other file types, such as .jpg, .png and .css, would need their own routing logic.

1. True
2. False

Poll 5 (Answer)

State whether the following statement is true or false.

Like HTML files, other file types, such as .jpg, .png and .css, would need their own routing logic.

1. **True**
2. False

fs.appendFile()

fs.appendFile()

upGrad

- The fs.appendFile() method is used to asynchronously append data into a file or create a file if it does not exist

- The following syntax is used for the fs.appendFile() method:

```
fs.appendFile(path, data[, options], callback
```

  - path <string> | <Buffer> | <URL> | <number> filename or file descriptor
  - data <string> | <Buffer>
  - options <Object> | <string>
    - encoding <string> | <null> **Default:** 'utf8'
    - mode <integer> **Default:** 0o666
    - flag <string> See support of file system flags. **Default:** 'a'.
  - callback <Function>
    - err <Error>

fs.appendFile()

upGrad

Sample Code Snippets

```
import { appendFile } from 'fs';

appendFile('message.txt', 'data to append', (err) =>
{
  if (err) throw err;
  console.log('The "data to append" was appended');
});
```

```
import { readFile } from 'fs';

var fs = require('fs');
fs.appendFile('demofile1.txt', 'Hello Upgrad!',
function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

fs.open()

fs.open()

upGrad

- The fs.open() method is used to asynchronously open the file if it already exists or create a new file and open it if it does not already exists

- The following syntax is used for the fs.open() method:

```
fs.open(path[, flags[, mode]], callback)
```

  - path <string> | <Buffer> | <URL>
  - flags <string> | <number> See support of file system flags. **Default:** 'r'.
  - mode <string> | <integer> **Default:** 0o666 (readable and writable)
  - callback <Function>
    - err <Error>
    - fd <integer>

- If the mode is 'r', i.e., the read mode, then an existing file would be opened for reading. An exception occurs if the file does not exist

fs.open()

upGrad

Sample Code Snippets

```
import { open } from 'fs';

var fs = require('fs');
fs.open('demofile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

fs.writeFile()

fs.writeFile()

upGrad

- The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file containing the specified content is created

- The following syntax is used for the fs.writeFile() method:

```
fs.writeFile(file, data[, options], callback)
```

  - file <string> | <Buffer> | <URL> | <integer> filename or file descriptor
  - data <string> | <Buffer> | <TypedArray> | <DataView> | <Object>
  - options <Object> | <string>
    - encoding <string> | <null> Default: 'utf8'
    - mode <integer> Default: 0o666
    - flag <string> See support of file system flags. Default: 'w'.
    - signal <AbortSignal> allows aborting an in-progress writeFile
  - callback <Function>
    - err <Error>

fs.writeFile()

upGrad

Sample Code Snippets

```
import { writeFile } from 'fs';

const data = new Uint8Array(Buffer.from('Hello Node.js'));
writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

Notes:

1. The **Buffer** class in Node. js is designed to handle raw binary data. Each **buffer** corresponds to some raw memory allocated outside V8. **Buffers** act like the arrays of integers, but they are not resizable and have a few methods meant specifically for binary data.

2. The Buffer.from() method creates a new buffer filled with the specified string, array or buffer.

3. **Uint8Array** typed array represents an array of 8-bit unsigned integers.

Poll 6 (15 Sec)

The _____ method replaces the specified file and content if it exists. If the file does not exist, a new file containing the specified content is created

1. fs.open()
2. fs.writeFile()
3. fs.appendFile()

Poll 6 (Answer)

The _____ method replaces the specified file and content if it exists. If the file does not exist, a new file containing the specified content is created

1. fs.open()

2. **fs.writeFile()**

3. fs.appendFile()

fs.unlink()

fs.unlink()

upGrad

- The fs.unlink() method asynchronously removes a file
- The following syntax is used for the fs.unlink() method:

```
fs.unlink(path, callback)
```

  - path <string> | <Buffer> | <URL>
  - callback <Function>
    - err <Error>

fs.unlink()

upGrad

Sample Code Snippets

```
import { unlink } from 'fs';

// Assuming that 'path/file.txt' is a regular file.
var fs = require('fs');
fs.unlink('path/file.txt', (err) => {
  if (err) throw err;
  console.log('path/file.txt was deleted');
});
```

fs.rename()

fs.rename()

upGrad

- The fs.rename() method asynchronously renames the file at oldPath to the pathname provided as newPath. If newPath already exists, it is overwritten. If there is a directory at newPath, an error is thrown instead

- The following syntax is used for the fs.rename() method:

```
fs.rename(oldPath, newPath, callback)
```

  - oldPath <string> | <Buffer> | <URL>
  - newPath <string> | <Buffer> | <URL>
  - callback <Function>
    - err <Error>

fs.rename()

upGrad

Sample Code Snippets

```
import { rename } from 'fs';

rename('oldFile.txt', 'newFile.txt', (err) => {
  if (err) throw err;
  console.log('Rename complete!');
});
```

Hands-On Exercise 2 (20 mins)

upGrad

- Create a Node project to upload a file

- Create a project ***upload_file***. Initialise it using ***npm init***
- Install the http-status-codes module
- Code a simple file upload form as the http response
- For parsing the uploaded file, i.e., reading the file and uploading it to a temporary location, you will require the formidable module

  Note: Remember to install the formidable module using ***npm i formidable --S,*** and then import it in the code ***const formidable = require('formidable');***

- The formidable module will store the uploaded file to the server but will be placed somewhere in a temporary folder with some the **.tmp** extension. So, it becomes difficult to trace it
- The path to this directory can be found in the 'files' object, which is passed as the third argument in the parse() method's callback function
- Use the fs module to move the file to your project's uploads folder. Use the fs module to rename the file (Hint: ***const fs = require('fs');***)

Refer: upload_file

Poll 7 (15 Sec)

Which of the following fs module methods is used to delete the file?

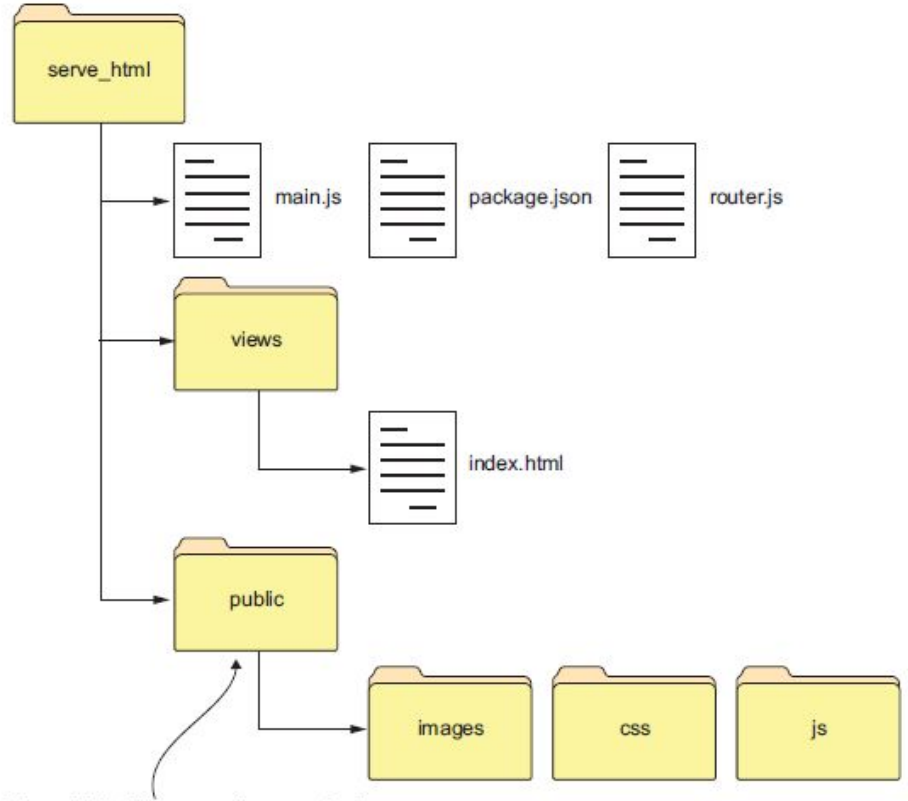
1. fs.delete()
2. fs.remove()
3. fs.unlink()
4. fs.undo()

Poll 7 (Answer)

Which of the following fs module methods is used to delete the file?

1. fs.delete()
2. fs.remove()
3. fs.unlink()
4. fs.undo()

Model-View-Controller (MVC)

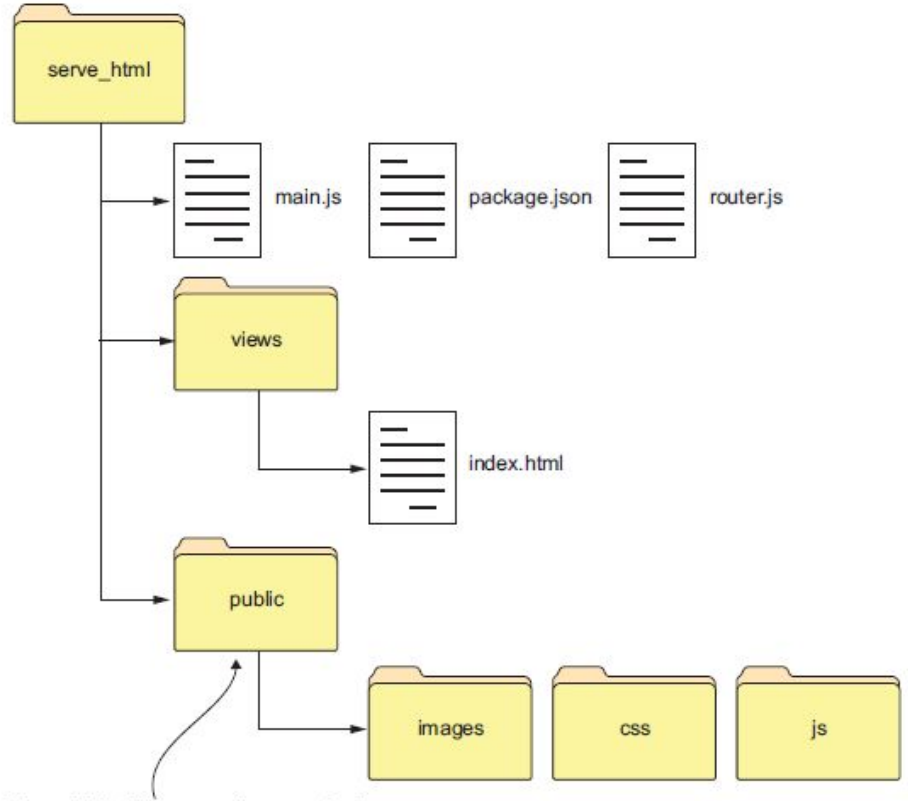


The public folder can be organized to separate your most common assets served to the client.

Our application's assets are the images, stylesheets and JavaScript that work alongside the views(HTML files) on the client side

To better organise the view part, you need to create a public folder at your project's root directory, and move all your assets there. Inside the public folder, you need to create a folder each for images, css and js, and move each asset into its respective folder

-- see the diagram.



The public folder can be organized to separate your most common assets served to the client.

Our application's assets are the images, stylesheets and JavaScript that work alongside the views(HTML files) on the client side

To better organise the view part, you need to create a public folder at your project's root directory, and move all your assets there. Inside the public folder, you need to create a folder each for images, css and js, and move each asset into its respective folder
-- see the diagram.

The view of the application resides in views and public folder.

As all the routing code is present in main.js, it forms the **CONTROLLER** part of this project

In the upcoming sessions, when we add database connectivity and DML and select queries in a separate set of .js files, which forms the **MODEL** part of the project

upGrad

Project Work: Checkpoint 2

Checkpoint 2: List of APIs

upGrad

Methods	Urls	Actions
GET	api/tutorials	Get all online courses
GET	api/tutorials/:id	Get online course by ID (Each online course has a unique ID)
POST	api/tutorials	Add a new online course (one at a time); **Only by ADMIN**
PUT	api/tutorials/:id	Update an online course by ID; **Only by ADMIN**
DELETE	api/tutorials/:id	Delete an online course by ID; **Only by ADMIN**
DELETE	api/tutorials	Remove all online course; **Only by ADMIN**
GET	api/tutorials/published	Find all published online courses
GET	api/tutorials?title=[kw]	Find all online courses whose titles contains 'kw'
GET	api/tutorials?category=[kw]	Find all online courses by category

Checkpoint 2: List of APIs

upGrad

Methods	Urls	Actions
POST	api/signup	Create a USER object and save it to the USER collection
POST	api/login	Verify login credentials (email_id and password). If verified, set ***IsLoggedIn*** to true
POST	api/logout	Log the user out, based on their **ID.** After logging them out, *IsLoggedIn* is set to false

The above APIs would enable for user management.

Checkpoint 2: Setting Up The Project

upGrad

You need to set up and create a proper directory structure and complete all the necessary npm installs

- Checklist: Is Node installed on your computer? **(Ans: YES)**
- Create a folder in the Node's path as follows:
  mkdir nodejs-express-mongodb
  $ cd nodejs-express-mongodb
- **npm init**
  name: (nodejs-express-mongodb)
  version: (1.0.0)
  description: Node.js Restful CRUD API with Node.js, Express and MongoDB
  entry point: (index.js) server.js
  test command:
  git repository:
  keywords: nodejs, express, mongodb, rest, api
  author: <selfName>
  licence: (ISC)

Checkpoint 2: Setting Up The Project

upGrad

- You need to install the following necessary modules: express, mongoose, body-parser and cors

- Run the following command:
  npm install express mongoose body-parser cors --S

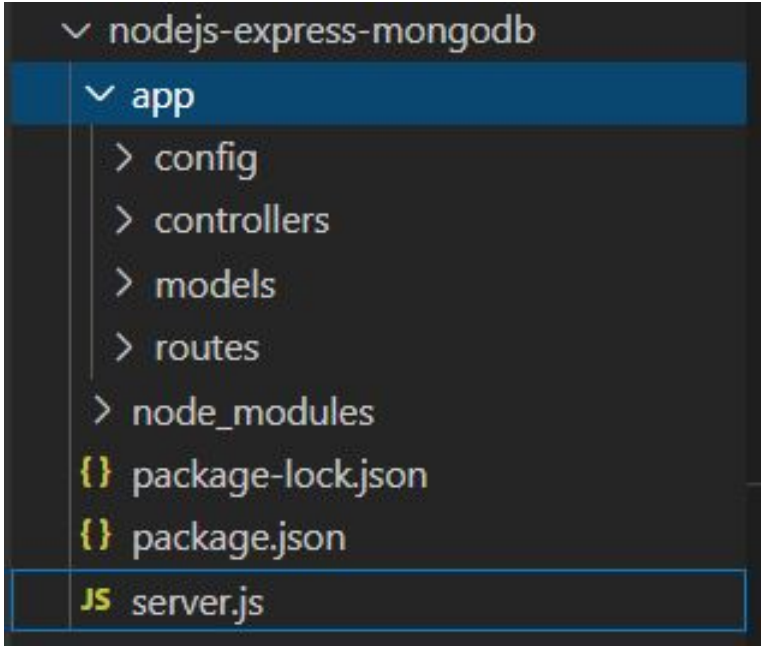
Here, *body-parser is used to extract the entire body portion of an incoming request stream and expose it on req.body. To read the HTTP POST data, you need to use the 'body-parser' node module. body-parser is a piece of express middleware that reads a form's input and stores it as a JavaScript object accessible through req.body*

CORS stands for Cross-Origin Resource Sharing. It allows you to make requests from one website to another website in the browser, which is normally prohibited by another browser policy called the Same-Origin Policy (SOP)

Project Work: Setting Up and Structuring the Project

upGrad

Project Structure



Note: Under the app (our application), we have four major folders

- config: This will hold the .js files that indicate configuration information such as connection parameters to MongoDB

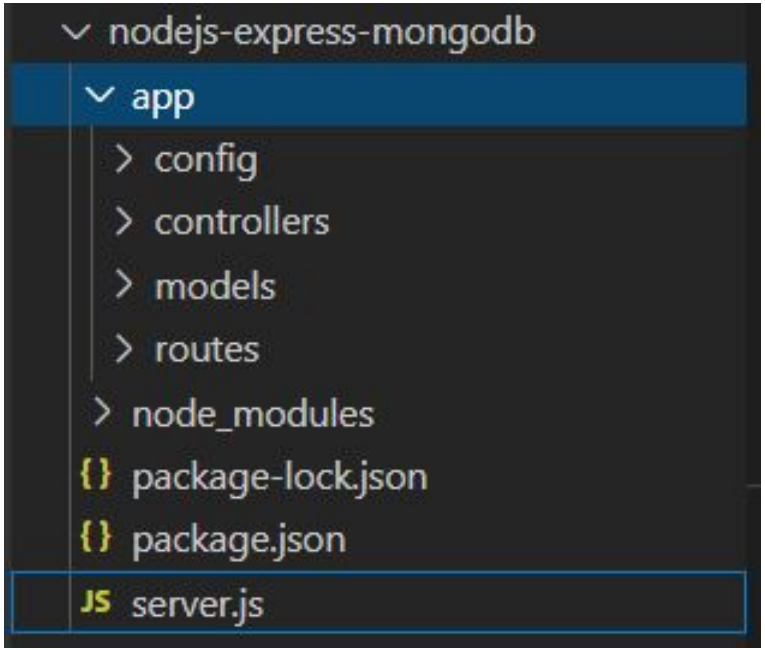
- controllers: This will hold the .js file that performs the core work including:
  - Creating and saving a new course: *create()*
  - Retrieving all courses from the database by title: *findAllByTitle()*
  - Retrieving all courses from the database by category: *findAllByCategory()*

Similarly, we will code other APIs, including findOne(), update(), delete(), deleteAll() and findAllPublished()

Also, we will code APIs for USER signup, login and logout, respectively

Project Work: Setting Up and Structuring the Project

upGrad

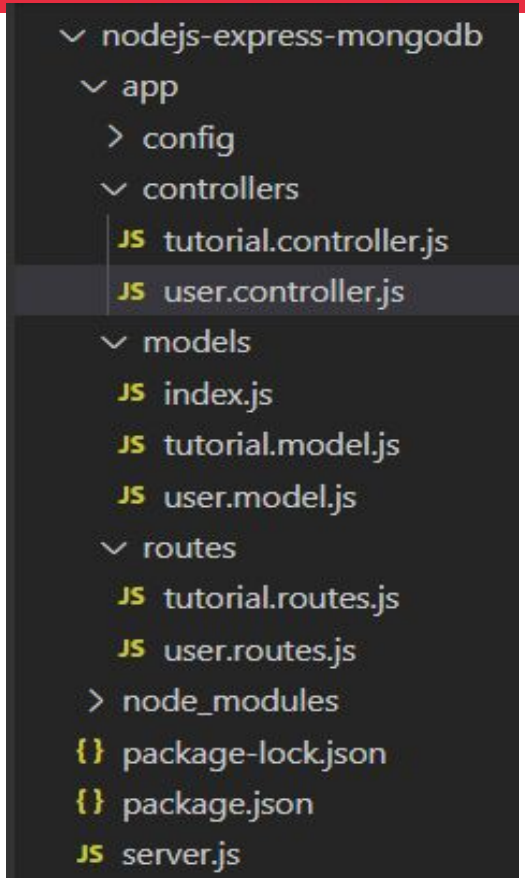


- models: This will hold the schema definition for MongoDB. Schema definition is done using Mongoose
- routes: This will hold the .js file for all the routing. When an http request comes in from the client (web app or mobile app, or a React application), then which file or API should be called is specified in the routes.

Project Work: Setting Up and Structuring the Project

upGrad

- Create the following .js files ***(no code to be added for now)***:
  - Under config:
    db.config.js
  - Under controllers:
    tutorial.controller.js
    user.controller.js
  - Under models:
    index.js
    tutorial.model.js
    user.model.js
  - Under routes:
    tutorial.routes.js
    user.routes.js
  - Under root folder:
    server.js



Homework

Create an app to serve static (only HTML without images) e-commerce web pages

- Your controller should dynamically read the URL file name from the request and accordingly create a file path.
- Use the file path from Step 1 and, using readFile() (from the fs module), read the respective html file and serve it.
- Your app should serve at least the following pages:
  - index.html: Welcomes the client to your e-commerce website
  - listofProducts.html: Indicates the list of products and their brand in a simple tabular format
  - PriceChart.html: Indicates the products, their price range and discounts in a simple tabular format
  - Contactus.html: Fetches basic contact information from the client

Refer: serve_ecommerce_html_pages

Doubt Clearance (5 mins)

upGrad

Key Takeaways

- The core modules in Node.js
- The fs core module and its application in routing the HTML files

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

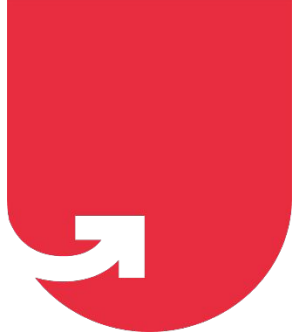
Homework
MCQs
Coding Questions

In the next class...

- You will learn how to route dynamic web applications

upGrad

#RahoAmbitious



Thank you!