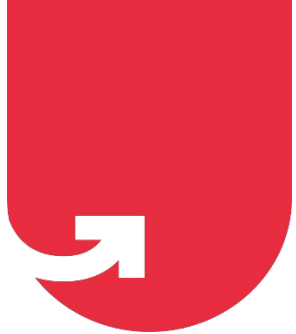
upGrad

#RahoAmbitious



Full Stack Software Development

upGrad

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

Lecture on: End-to-End App Development

Instructor: Rocky Jagtiani

upGrad

In the previous class, we covered...

- We learnt about different Express.js modules

Today’s Agenda

- Multer

End-to-End Development

upGrad

- So far you understood that **node.js** is a backend framework. This allows the javascript codes to run on backend servers.

- To store data in a NoSQL database, you used **MongoDB**.

- As MongoDB does not have its own schema definition language, you used **Mongoose**. In fact, using mongoose, you can also build associations between different schemas, namely one-to-one, one-to-many and many-to-many.

- To make routing codes easier and shorter, you learned a framework called **express.js**. Here, you clearly understood routing, arranging the entire code blocks in MVC and handling the four important HTTP request calls, namely GET, POST, PUT and DELETE.

Case Study:

Code an application to upload or store files in MongoDB using Node.js, Express and Multer

- You would code this case study in two versions :
  - Version 1: Code to only upload a single image file at a time.
  - Version 2: Enhance the application to allow multiple image file selections and many uploads simultaneously.

***Note 1**: Multer is a node.js middleware used to handle multipart/form-data, which is primarily used for uploading files (https://www.npmjs.com/package/multer).*

***Note 2:** multer-gridfs-storage is a GridFS storage engine for Multer to store uploaded files directly to MongoDb (https://www.npmjs.com/package/multer-gridfs-storage).*

Multer

Multer

upGrad

- Multer is a node.js middleware used to handle multipart/form-data, which is primarily used for uploading files.

  NOTE: *Multer will not process any form that is not multipart (multipart/form-data).*

- Installation

```
$ npm install multer --S
```

```
<form action="/profile" method="post" enctype="multipart/form-data">
  <input type="file" name="avatar" />
</form>
```

Multer

upGrad

```
var express = require('express')
var multer = require('multer')
var upload = multer({ dest: 'uploads/' })

var app = express()

app.post('/profile', upload.single('upgradLogo'), function (req, res, next) {
  // req.file is the `upgradLogo ` file
  // req.body will hold the text fields, if there were any
})

app.post('/photos/upload', upload.array('photos', 12), function (req, res, next)
{
  // req.files is array of `photos` files. Max 12 pics allowed
  // req.body will contain the text fields, if there were any
})
```

- Multer adds a body object and a file or files object to the request object. The body object contains the values of the text fields of the form. The file or files object contains the files uploaded via the form.

Multer

upGrad

- **Multer** stores information for each file as attributes.

fieldname	Field name specified in the form
Original-name	Name of the file on the user's computer
encoding	Encoding type of the file
Mime-type	Mime type of the file
size	Size of the file in bytes
destination	The folder to which the file has been saved
filename	The name of the file within the destination
path	The full path to the uploaded file

Multer

upGrad

- Multer accepts an **options object**, the most basic of which is the ***dest*** property, which tells Multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to the disk.

- By default, Multer will rename the files to avoid name-related conflicts. The renaming function can be customised according to your needs.

- The following are the options that can be passed to Multer.
  - dest or storage: Where to store the files
  - fileFilter: Function to control which files are accepted
  - limits: Limits of the uploaded data
  - preservePath: Keep the full path of the files instead of just the base name

Hands-on Exercise 1 (45 mins)

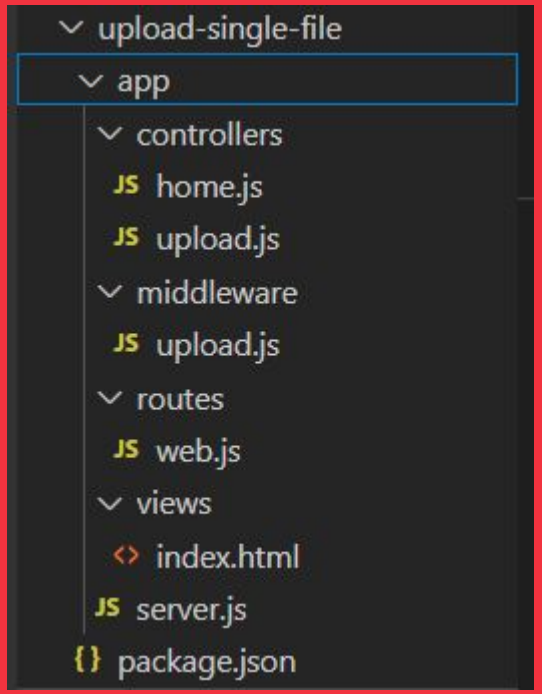
upGrad

- Create a project folder upload-single-file ***(to upload image files, one at a time)***

- Create the following project structure.
  - views/index.html contains HTML form for user to upload images.
  - routes/web.js defines routes for endpoints that are called from views and use controllers to handle requests.
  - Controllers:
    - home.js returns views/index.html
    - upload.js handles upload and storing images with middleware function
  - middleware/upload.js initialises Multer GridFS storage engine (including MongoDB) and defines middleware function.
  - server.js initialises routes and runs Express application.

Hands on Exercise 1 (45 mins)

upGrad



Hands on Exercise 1 (45 mins)

upGrad

- Initialise the project
  npm init
- Install Express, Multer and Multer GridFS storage using the following command:
  npm install express multer multer-gridfs-storage –S
- In the ‘views’ folder, create index.html file with the HTML and Javascript or Jquery code.
- For HTML part, you create a form using the following elements:

```
action="/upload"
method="POST"
enctype="multipart/form-data"
```

Hands on Exercise 1 (45 mins)

upGrad

- Create middleware to upload and store the image.
- Create Controller for the view -> **controllers/home.js**.
- Create Controller for uploading Images -> **controllers/upload.js**.
- In the routes folder, define routes in web.js with Express Router.
- Create Express app server.
- Test the app.

Note: Two databases would be created in MongoDB databaseName.photos.files and databaseName.photos.chunks.

Refer To: upload-single-file

photos.chunks and photo.files

upGrad

- If you look at the output of the first hands-on exercise in the database, you would notice that 2 collections were created by Multer and multer-gridfs-storage node.js modules.
- Let's understand **the purpose and association between the two collections**.

A document in the **photos.files collection:**


```
{
"_id":{"$oid":"607699f1312b6a15a041e1ab"},
"length":44128,
"chunkSize":261120,
"uploadDate":{"$date":"2021-04-14T07:29:55.522Z"},
"filename":"1618385393919-rocky-AJAX workflow.png",
"md5":"e1632a92d15203372f75a25ea1696500",
"contentType":"image/png"
}
```


A document in the **photos.chunks collection:**


```
{
  "_id":{"$oid":"607699f3312b6a15a041e1ac"},
  "files_id":{"$oid":"607699f1312b6a15a041e1ab"},
  "n":0,
   "data":{
     "$binary":"iVBORw0KGgoAAAANSUhEUgAABR.....",
      "$type":"0"
   }
}
```

photos.chunks and photo.files

upGrad

Instead of storing a file in a single document, GridFS divides the file into parts, or chunks, and stores each chunk as a separate document. By default, GridFS uses a default chunk of size 255 kB, that is, GridFS divides a file into chunks of 255 kB each, with the exception of the last chunk. The last chunk is only as large as necessary. Similarly, files that are not larger than the chunk size are included in a single chunk, using only as much space as needed, along with some additional metadata.

GridFS uses two collections to store files. One collection stores the file chunks and the other stores file metadata.

photos.chunks and photo.files

upGrad

Syntax of the Chunks Collection:
Each document in the chunks collection represents a distinct chunk of a file as represented in GridFS. Documents in this collection have the following form:

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary>
}
```

Syntax of the Files Collection:
Each document in the files collection represents a file in GridFS.

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>,
  "uploadDate" : <timestamp>,
  "md5" : <hash>,
  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <any>,
}
```

Poll 1 (15 sec)

What is the default chunk size in bytes when storing files in parts in the MongoDB?

1. 1024
2. 65536
3. 261120
4. Any multiple of 1024, depending on the version of MongoDB

Poll 1 (Answer)

What is the default chunk size in bytes when storing files in parts in the MongoDB?

1. 1024
2. 65536
3. **261120**
4. Any multiple of 1024, depending on the version of MongoDB

How to Upload Multiple Image Files ?

upGrad

- In the previous End-to-End project, you used Node.js modules, Express.js for routing, MongoDB for storing files data and Mongoose for defining the schema.
- What changes should be done in the project code, such that you can upload multiple image files?
- We would need to do the following changes:

In views/index.html
Change form for uploading multiple images. In the form, update the input tag name and add multiple attributes, as depicted below:

```
<input
  type="file"
  name="multi-files"
  multiple
  id="input-files"
  class="form-control-file
border"
/>
```

How to Upload Multiple Image Files ?

upGrad

- Modify middleware to upload and store image. For multiple images, you use function: **array()** instead of single().

In middleware/upload.js

```
// var uploadFile = multer({ storage: storage }).single("file");

var uploadFiles = multer({ storage: storage }).array("multi-files", 10);

var uploadFilesMiddleware = util.promisify(uploadFiles);
module.exports = uploadFilesMiddleware;
```

array() function limits the number of files that are uploaded each time. The first parameter is the name of the input tag (in html view: <input type="file" name="multi-files">), the second parameter is the maximum number of files (10) that can be selected for uploading.

How to Upload Multiple Image Files ?

upGrad

- In case if more than the permissible number of files are selected, then LIMIT_UNEXPECTED_FILE error would be raised.

- Modify the **Controller** for uploading multiple images. Here, the following updates are expected:
  - Change the controller function name from uploadFile to uploadFiles.
  - Check if atleast one file has been selected or not.
  - Handle the error code **LIMIT_UNEXPECTED_FILE**. This error is raised when the user selects more than max_allowed number of files to upload.

Refer To: **upload-multiple-file**

How to Upload Multiple Image Files ?

upGrad

```
onst upload = require("../middleware/upload");
const uploadFiles = async (req, res) => {
  try {
    await upload(req, res);
    console.log(req.files);

    if (req.files.length <= 0) {
      return res.send(`You must select at least 1 file.`);
    }

    return res.send(`Files have been uploaded.`);
  } catch (error) {
    console.log(error);

    if (error.code === "LIMIT_UNEXPECTED_FILE") {
      return res.send("Too many files to upload.");
    }
    return res.send(`Error when trying upload many files: ${error}`);
  }
};

module.exports = {
  uploadFiles: uploadFiles
};
```

Homework

- Create a mini-project that would provide the following three functionalities or **APIs**

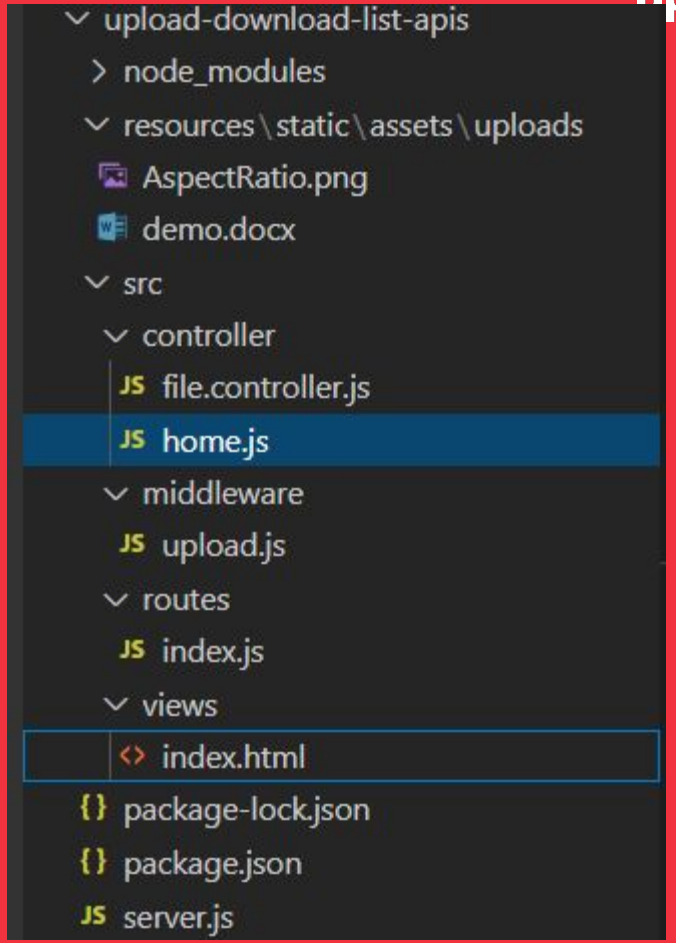
Methods	**Urls**	**Actions**
POST	/upload	upload a File
GET	/files	get List of Files (name and url)
GET	/files/[filename]	download a File

- The files would be uploaded and downloaded to/from a static folder using Multer (with file size limit of 1 MB).

- node.js modules required for the project include:
  - express
  - multer
  - cors

upGrad

- Create the following project structure:
  - resources/static/assets/uploads: Folder for storing uploaded files
  - middleware/upload.js: Initialises Multer Storage engine and defines middleware function to save uploaded files in the ‘uploads’ folder
  - file.controller.js exports Rest APIs: POST a file, GET all files’ information and download a file with url
  - home.js: Displays the view (views/index.html)
  - routes/index.js: Defines routes for endpoints that are called from HTTP Client and uses a controller to handle requests.
  - server.js: Initialises routes and runs Express app



- Setup Node.js Express File Upload Project.
  npm install express multer cors --S
  npm init

- Create middleware for file upload.The middleware will use Multer for handling multipart/form-data and uploading files.

- Inside the middleware folder, create **upload.js** file.

- In the middleware/upload.js file, you do the following steps:
  - First, you import multer module.
  - Next, you configure multer to use the Disk Storage engine.

- Here, two options are used:
  - Destination determines the folder to store the uploaded files.
  - The filename determines the name of the file inside the destination folder.

- util.promisify() allows the exported middleware object to be used with async-await.

- Create Controller for file upload/download.
- In the ‘controller’ folder, create file.controller.js
- For File Upload method, you export upload() function that:
  - uses middleware function for file upload
  - catches Multer error (in middleware function)
  - returns response with message
- For File information and download:
  - getListFiles(): Reads all files in the ‘uploads’ folder, return a list of files’ information (name, url)
  - download(): Receives file name as input parameter, then uses Express res.download API to transfer the file on the path (directory + file name) as an 'attachment'.

- Define Route for uploading the file (routes/index.html).
- When a client sends HTTP requests, you need to determine how the server will respond after setting up the routes. There are four routes with corresponding controller methods:

```
GET     /                   getHome()
POST    /upload:            upload()
GET     /files:             getListFiles()
GET     /files/[fileName]:  download()
```

- Create Express app server (server.js).

- Test the following APIs:
  - **https://localhost:3000/**
    This would show the index.html file (under ‘views’ folder). Here, select one to upload at a time. On form submit, the following URL is called http://localhost:3000/upload.

  - **https://localhost:3000/files**
    This would return a json list of files (name and url) format.

  - **https://localhost:3000/files/abcd.png**
    This would download the file, provided the file is located in the resources/static/assets/uploads folder.

Doubt Clearance (5 mins)

upGrad

Key Takeaways

- We developed an application using the concepts learnt so far.

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

Homework
MCQs
Coding Questions
Course Project - Checkpoint 8

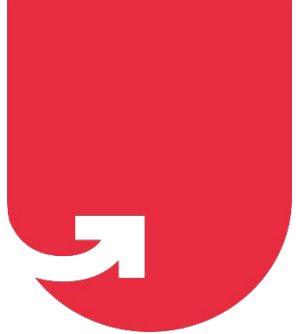
upGrad

In the next class, we will discuss...

- We learn to integrate the React frontend with our backend code

upGrad

#RahoAmbitious



Thank you!