

Você será capaz de:

- Fazer upload de arquivos em APIs REST;
 - Salvar arquivos no servidor através de uma API REST;
 - Consultar arquivos do servidor através de uma api REST.
-

Por que isso é importante?

Provavelmente você já precisou enviar um arquivo pelo Google Drive, WhatsApp ou Facebook? Uma foto, um vídeo, uma planilha do Excel, etc. O envio de arquivos é uma funcionalidade que pode servir a diversos tipos de aplicações. Sendo assim, é importante que uma pessoa desenvolvedora moderna saiba construir APIs que possam lidar com esse tipo de operação.

Upload de arquivos com **multer** e Express

multer

Você já utilizou, em outros momentos, o **body-parser** para tratar dados no corpo da request. Hoje você vai utilizar o **multer**. A funcionalidade dos dois é, em suma, a mesma: interpretar dados enviados através do body da requisição.

No entanto, enquanto o **body-parser** suporta requests nos formatos JSON (**Content-Type: application/json**) e URL Encoded (**Content-Type: application/x-www-form-urlencoded**), o **multer** suporta requests no formato conhecido como Form Data (**Content-Type: multipart/form-data**).

multipart/form-data

Este é um formato bem antigo, pensado para suportar todas as operações suportadas pela tag **<form>** do HTML. Sendo assim, pode transmitir dados comuns, como strings, booleans e números, mas também pode transmitir arquivos. Dessa forma, o body de uma request com formato Form Data

pode ter vários campos (assim como um JSON), e cada campo pode ter o tipo número, boolean, string, ou arquivo .

Já que suporta upload de arquivos, o `multer` nos fornece, além do `req.body` , com os campos comuns, uma propriedade `req.file` (ou `req.files` , caso sejam múltiplos arquivos na mesma request).

Assista ao vídeo abaixo para entender mais sobre o funcionamento do `multer` :

VÍDEO: 28.2 MULTER - Duração 15:37

Show me the code

Para começar, vamos criar um projeto chamado `io-multer` . Para isso, em sua pasta de exercícios, execute o comando `npm init @tryber/backend io-multer` .

Depois de criada a pasta do projeto, navegue até ela e instale o `multer` utilizando os seguintes comandos:

Copiar

```
cd io-multer
```

```
npm i multer
```

Agora, basta executar o comando `npm start` dentro da pasta `io-multer` para colocar nosso servidor de pé. Se preferir, utilize o comando `npm run debug` , para que o `nodemon` reinicie o servidor sempre que você realizar novas alterações.

Para testar nossa API, disponibilizamos um endpoint chamado `/ping` . Para fazer uma requisição para essa rota, podemos usar diretamente o browser, como mostrado na imagem abaixo:



pong!

Agora que já temos nosso servidor de pé e sabemos o que é o `multer` , vamos criar uma instância dele e configurá-lo para tornar a pasta `uploads` como pasta de destino dos uploads realizados. Além disso, vamos também tornar pública essa mesma pasta para que ela possa ser acessada através da nossa API. Assim, poderemos requisitar de volta os arquivos após fazer o upload deles:

io-multer/index.js

Copiar

```
// require('dotenv').config();
// const express = require('express');
// const cors = require('cors');
// const bodyParser = require('body-parser');
const multer = require('multer');

// const PORT = process.env.PORT;

// const controllers = require('./controllers');

// const app = express();

// app.use(
//   cors({
//     origin: `http://localhost:${PORT}`,
//     methods: ['GET', 'POST', 'PUT', 'DELETE'],
//     allowedHeaders: ['Authorization'],
//   })
// );

// app.use(bodyParser.json());
// app.use(bodyParser.urlencoded({ extended: true }));

/* Definindo nossa pasta pública */
/* `app.use` com apenas um parâmetro quer dizer que
queremos aplicar esse middleware a todas as rotas, com qualquer
método */
/* __dirname + '/uploads' é o caminho da pasta que queremos expor
publicamente */
/* Isso quer dizer que, sempre que receber uma request, o express
vai primeiro
verificar se o caminho da request é o nome de um arquivo que
existe em `uploads`.
Se for, o express envia o conteúdo desse arquivo e encerra a
response.
Caso contrário, ele chama `next` e permite que os demais
endpoints funcionem */
app.use(express.static(__dirname + '/uploads'));

/* Cria uma instância do `multer` configurada. O `multer` recebe um
objeto que,
```

```

    nesse caso, contém o destino do arquivo enviado. */
const upload = multer({ dest: 'uploads' });

// app.get('/ping', controllers.ping);

// app.listen(PORT, () => {
//   console.log(`App listening on port ${PORT}`);
// });

```

Com tudo configurado, vamos de fato criar uma rota que vai receber e salvar um único arquivo na pasta **uploads** :

io-multer/index.js

Copiar

```

// require('dotenv').config();
// const express = require('express');
// const cors = require('cors');
// const bodyParser = require('body-parser');
// const multer = require('multer');

// const PORT = process.env.PORT;

// const controllers = require('./controllers');

// const app = express();

// app.use(
//   cors({
//     origin: `http://localhost:${PORT}`,
//     methods: ['GET', 'POST', 'PUT', 'DELETE'],
//     allowedHeaders: ['Authorization'],
//   })
// );

// app.use(bodyParser.json());
// app.use(bodyParser.urlencoded({ extended: true }));

// /* Definindo nossa pasta pública */
// /* `app.use` com apenas um parâmetro quer dizer que
//    queremos aplicar esse middleware a todas as rotas, com
//    qualquer método */
// /* __dirname + '/uploads' é o caminho da pasta que queremos expor
//    publicamente */

```

```
// /* Isso quer dizer que, sempre que receber uma request, o express
vai primeiro
//   verificar se o caminho da request é o nome de um arquivo que
existe em `uploads`.
//   Se for, o express envia o conteúdo desse arquivo e encerra a
response.
//   Caso contrário, ele chama `next` e permite que os demais
endpoints funcionem */
// app.use(express.static(__dirname + '/uploads'));

// /* Cria uma instância do `multer` configurada. O `multer` recebe um
objeto que,
//   nesse caso, contém o destino do arquivo enviado. */
// const upload = multer({ dest: 'uploads' });

app.post('/files/upload', upload.single('file'), (req, res) =>
  res.status(200).json({ body: req.body, file: req.file })
);

// app.get('/ping', controllers.ping);

// app.listen(PORT, () => {
//   console.log(`App listening on port ${PORT}`);
// });
```

Note que, na rota `/files/upload`, passamos um middleware criado pelo `multer` como parâmetro, através da chamada `upload.single('file')` e depois passamos nosso próprio middleware, que recebe os parâmetros `req` e `res`. O `multer` adiciona um objeto `body` e um objeto `file` ao objeto `request` recebido na callback. Os objetos `body` e `file` contêm os valores dos campos de texto e o arquivo enviados pelo formulário, respectivamente. O parâmetro passado na chamada de `upload.single('file')` indica o nome do campo que conterá o arquivo. No caso desse exemplo, o nome é `file`, mas poderia ter outro nome em outros cenários. Por exemplo, se um formulário fosse construído desta forma:

Copiar

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

Seria necessário especificar o nome do input com `upload.single('post')`, pois o atributo `name` do input do tipo `file` está preenchido com `post`.

Além disso, estamos especificando, com o método `single` , porque queremos apenas um arquivo. Ou seja, qualquer pessoa que nos enviar uma requisição deverá informar uma propriedade chamada `file` , e só poderá enviar um arquivo por requisição.

Exercício de Fixação

Crie um arquivo que receba arquivos enviados para `http://localhost:3000/envios` e os armazene na pasta `envios` . Dica: você pode usar a pasta `io-multer` criando um novo arquivo, pois ela já tem os pacotes necessários, e o `io-multer/index.js` como exemplo.

Axios

Chegou a hora de testarmos nossa API. Para isso, vamos introduzir o `Axios` , uma biblioteca que nos ajudará a realizar requisições HTTP para APIs REST.

`Axios` é uma biblioteca que fornece um cliente HTTP que funciona tanto no browser quanto no NodeJS. Ela consegue interagir tanto com `XMLHttpRequest` quanto com a interface HTTP nativa do NodeJS. Por isso, uma das vantagens de se usar o `Axios` é que ele permite que o mesmo código utilizado para fazer requisições Ajax no browser também funcione no servidor. Além disso, as requisições feitas através da biblioteca retornam uma Promise compatível com a versão ES6 do JavaScript.

Nota : o `Axios` é parecido com o `fetch` , que você já aprendeu. Porém, é sempre interessante aprender a manipular diferentes bibliotecas, mesmo que elas tenham o mesmo objetivo!

Como um exemplo prático de sua utilização, vamos criar um script para saber se nossa API está de pé. Para isso, vamos criar outra pasta chamada `ping` , fora da nossa pasta `io-multer` , e, dentro dela, vamos criar um arquivo chamado `ping.js` .

Execute os comandos abaixo para prosseguir:

Copiar

```
> cd ..  
> mkdir ping  
> cd ping  
> npm init -y  
> npm install axios
```

Seu `package.json` deve se parecer com este:

Copiar

```
{
```

```

    "name": "ping",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
      "axios": "^0.19.2"
    }
  }
}

```

Dentro do arquivo `ping.js` , vamos usar o Axios para fazer uma requisição ao nosso servidor, que está rodando na porta 3000:

ping/ping.js

Copiar

```

const axios = require('axios');

/* Faz uma requisição do tipo GET */
axios
  .get('http://localhost:3000/ping/')
  .then((response) => {
    console.log(response.data);
    console.log(response.status);
  })
  .catch((error) => {
    console.log(error);
  });

```

Certifique que o `io-multer/index.js` esteja de pé para receber sua requisição `axios` e rode esse script, com `node ping.js` , na pasta `ping` , e veja a saída. Você deverá ver no console a mensagem `pong! 200` .

Explicando melhor o que aconteceu: o axios fez uma requisição HTTP, assim como as que o `Postman` faz, e assim como as que o `browser` faz.

Existem outras formas de se fazer requisições HTTP através do axios:

GET

Copiar

```

axios.get('/user', {
  params: {
    ID: 12345
  }
})

```

```

    })
    .then((response) => {
      console.log(response);
    })
    .catch((error) => {
      console.log(error);
    })
  })
}

```

```

// Você pode usar métodos async também
const getUser = async () => {
  try {
    const response = await axios.get('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}

```

POST

Copiar

```

const body = {
  firstName: 'Fred',
  lastName: 'Flintstone'
};

axios.post('/user', body)
  .then((response) => {
    console.log(response);
  })
  .catch((error) => {
    console.log(error);
  });

```

Você pode conferir mais exemplos na [documentação do axios](#).

Fazendo o upload de arquivos para uma API

Agora que já sabemos como utilizar o axios, vamos usá-lo para enviar um arquivo, lido localmente com o NodeJS, para a nossa API. Para isso, vamos criar mais uma pasta chamada `send-files`, fora das pastas criadas anteriormente. Lá dentro, criaremos dois arquivos: `send.js` e `meu-arquivo.txt`. Dentro de `meu-arquivo.txt`, coloque um texto qualquer. Lembre-se sempre de criar um projeto node com `npm init`.

Execute os seguintes comandos para prosseguir:

Copiar

```
cd ..  
mkdir send-files  
cd send-files  
npm init -y  
npm i axios form-data
```

`form-data` é uma biblioteca que nos ajudará a montar uma requisição do tipo `multipart/form-data`. Ela pode ser usada para submeter formulários e fazer upload de arquivos para outras aplicações web. Note que, no navegador, a classe `FormData`, fornecida por essa biblioteca, já existe por padrão, de forma que o uso do pacote de terceiros só se faz necessário no Node.js.

Dentro de `send.js`, colocamos o código abaixo:

`send-file/send.js`

Copiar

```
const FormData = require('form-data');  
const axios = require('axios');  
const fs = require('fs');  
  
/* Criamos um stream de um arquivo */  
const stream = fs.createReadStream('./meu-arquivo.txt');  
  
/* Aqui, criamos um formulário com um campo chamado 'file' que  
carregará */  
/* o stream do nosso arquivo */  
const form = new FormData();  
form.append('file', stream);  
  
/* Esse arquivo não será enviado no body da requisição como de  
costume. */  
/* Em ambientes NodeJS, é preciso setar o valor de boundary no  
header */  
/* 'Content-Type' chamando o método `getHeaders` */  
const formHeaders = form.getHeaders();  
  
axios  
  .post('http://localhost:3000/files/upload', form, {  
    headers: {  
      ...formHeaders,  
    },  
  })
```

```
.then((response) => {
  console.log(response.status);
})
.catch((error) => {
  console.error(error);
});
```

Em seguida, execute o arquivo `send.js` . Caso nenhum erro tenha ocorrido, verifique a pasta `/uploads` do nosso servidor que fica no projeto `io-multer` , lembra? Você verá que existe um arquivo com um nome como `f9556c41394ad1885b7f6e3d60b7d997` . Dentro dele, haverá o conteúdo do seu arquivo `meu-arquivo.txt` .

Dando nome aos "bois" arquivos com `multer` Storage

Como você percebeu, foi gerado um arquivo com um nome bizarro, não é mesmo? Como podemos fazer para dar um nome a esse arquivo?

Dentro no script do nosso servidor, vamos criar um `multer` Storage . Um storage nos permite ter um controle mais detalhado do upload de nossos arquivos. Podemos extrair o valor do nome original do arquivo enviado pelo formulário através da propriedade `originalname` :

`io-multer/index.js`

Copiar

```
// require('dotenv').config();
// const express = require('express');
// const cors = require('cors');
// const bodyParser = require('body-parser');
// const multer = require('multer');

// const PORT = process.env.PORT;

// const controllers = require('./controllers');

// const app = express();

// app.use(
//   cors({
//     origin: `http://localhost:${PORT}`,
//     methods: ['GET', 'POST', 'PUT', 'DELETE'],
//     allowedHeaders: ['Authorization'],
//   })
// );
```

```
// app.use(bodyParser.json());
// app.use(bodyParser.urlencoded({ extended: true }));

// /* Definindo nossa pasta pública */
// /* `app.use` com apenas um parâmetro quer dizer que
//    queremos aplicar esse middleware a todas as rotas, com
//    qualquer método */
// /* __dirname + '/uploads' é o caminho da pasta que queremos expor
//    publicamente */
// /* Isso quer dizer que, sempre que receber uma request, o express
//    vai primeiro
//    verificar se o caminho da request é o nome de um arquivo que
//    existe em `uploads`.
//    Se for, o express envia o conteúdo desse arquivo e encerra a
//    response.
//    Caso contrário, ele chama `next` e permite que os demais
//    endpoints funcionem */
// app.use(express.static(__dirname + '/uploads'));

/* destination: destino do nosso arquivo
   filename: nome do nosso arquivo.

No caso, vamos dar o nome que vem na
propriedade `originalname`, ou seja,
o mesmo nome que o arquivo tem no
computador da pessoa usuária */
const storage = multer.diskStorage({
  destination: (req, file, callback) => {
    callback(null, 'uploads');
  },
  filename: (req, file, callback) => {
    callback(null, file.originalname);
  }
});

const upload = multer({ storage });

// app.post('/files/upload', upload.single('file'), (req, res) =>
//   res.status(200).json({ body: req.body, file: req.file })
// );

// app.get('/ping', controllers.ping);

// app.listen(PORT, () => {
```

```
// console.log(`App listening on port ${PORT}`);  
// });
```

Reinicie novamente o servidor do projeto `io-multer` , com `node index.js` . Em seguida, execute o script `send.js` , com `node send.js` , várias vezes e confira sua pasta `uploads/` na pasta `io-multer` , no caso, seu servidor. Repare que agora foi gerado outro arquivo, porém com o nome `meu-arquivo.txt` . 😊 Você executou várias vezes, certo? Nada aconteceu desde que o arquivo `meu-arquivo.txt` foi gerado a primeira vez. Caso você altere o texto que está dentro do `meu-arquivo.txt` , e execute novamente, não será gerado um novo arquivo `meu-arquivo.txt` , ele será apenas atualizado com o novo valor do conteúdo!

i Faça o teste **i**: Seguindo o exemplo anterior, crie um arquivo que salve os arquivos enviados para `http://localhost:3000/uploads` , o formato dos arquivos salvos deve ser a seguinte:

`nome-do-arquivo-enviado${data-de-agora}` , sem a extensão do arquivo enviado.

Um ponto que merece ser comentado é o uso da callback para informar ao `multer` o nome do arquivo a ser armazenado. Isso significa duas coisas:

1. Podemos utilizar código assíncrono (como realizar uma busca no banco, por exemplo);
2. Podemos passar um erro no primeiro parâmetro caso não seja desejado prosseguir com o armazenamento do arquivo.

Acessando os arquivos enviados pela API

Como já tornamos pública a pasta `/uploads` , que é onde guardamos os arquivos enviados, não precisamos fazer mais nada para deixá-los disponíveis através da API.

Se você acessar `http://localhost:3000/meu-arquivo.txt` , deverá ver o conteúdo do seu arquivo no browser. Que tal testar com outros tipos de arquivo, como uma imagem?


Melhor ainda, você pode modificar esse script para pedir que a pessoa usuária digite na linha de comando o nome do arquivo que quer fazer upload. 😊

Agora, a prática

Vamos juntar tudo o que aprendemos até aqui e exercitar mais ainda nosso aprendizado!

Antes de começar, crie um projeto chamado `multer-exercises` utilizando o comando `npm init @tryber/backend multer-exercises`.

Depois de criar o projeto, instale o `multer` acessando a pasta e executando o comando `npm i multer` dentro dela.

Agora sim!  Tudo pronto para começar os exercícios!

1. Crie o endpoint `POST /upload`
2. O endpoint deve receber apenas um arquivo no campo `file` ;
3. O arquivo deve ser armazenado na pasta `uploads` ;
4. O arquivo armazenado deve ter o timestamp do upload (obtido com `Date.now()`) seguido do nome original do arquivo.
Exemplo, para o arquivo `profile.png` , o nome armazenado deve ser algo como `1616691266095-profile.png` , já que o timestamp será diferente a cada vez.
5. Retorne status `200 OK` se der tudo certo.
6. Altere o endpoint `POST /upload` para que atenda os seguintes critérios:
7. Apenas aceite arquivos cuja extensão seja `.png` ; Caso o arquivo tenha outro tipo de extensão, retorne o status `403 Forbidden` com o JSON a seguir:

Copiar

```
{
  "error": { "message": "Extension must be `png`" }
}
```

2. Não aceite um arquivo cujo nome (ignorando o timestamp) já exista na pasta `uploads` . Caso o arquivo já exista, retorne o status `409 Conflict` com o seguinte JSON:

Copiar

```
{
  "error": { "message": "File already exists" }
}
```

Dica: procure sobre `fileFilter` no multer, pode ajudar.

3. Torne a pasta `uploads` pública de forma que seja possível baixar os arquivos enviados anteriormente.

Bônus

1. Crie o endpoint `POST /multiple`

1. Permita o upload de vários arquivos através do campo `files` ;
2. Salve cada arquivo na pasta `/uploads` com um nome aleatório, que será gerado pelo `multer` ;
3. Retorne uma lista dos arquivos enviados juntamente com a URL pela qual cada um está acessível. Exemplo:

Copiar

```
[
  { "file": "meuArquivo.txt", "url":
"http://localhost:3000/c3f20f8a1a72729883b88a96f405bbd0" },
  { "file": "arquivo1.png", "url":
"http://localhost:3000/7c76b101fd872f7fc12705eeba2ddd1c" },
  { "file": "profile.jpg", "url":
"http://localhost:3000/0ec57a65a9522aa14f9405060089c6f5" },
  { "file": "tcc.docx", "url":
"http://localhost:3000/a7de65196a12ce1c53e8e76927099f12" },
  { "file": "CNH.jpg", "url":
"http://localhost:3000/78c948b7b737d9a80b13f52bc6968d75" }
]
```

2. Crie o endpoint `POST /profile`

1. Receba strings nos campos `name` , `email` , `password` e `bio` ;
2. Receba um arquivo no campo `profilePic` ;
3. Armazene o arquivo recebido na pasta `/profilePics` com o nome aleatório do `multer` ;
4. Utilize o nome gerado pelo `multer` como ID para o perfil criado;
5. Armazene as informações do perfil no arquivo `profiles.json`

3. Crie o endpoint `GET /profiles/:id`

1. Caso exista um perfil com o `id` informado, retorne as informações desse perfil, conforme salvo no arquivo `profiles.json`
2. Caso não exista um perfil com o `id` informado, retorne o status `404 Not Found` com o seguinte corpo:

Copiar

```
{
  "error": {
    "message": "Perfil não encontrado"
  }
}
```

Conteúdos

Show me the code

Exercício de Fixação

Crie um arquivo que receba arquivos enviados para <http://localhost:3000/envios> e os armazene na pasta `envios`. Dica: você pode usar a pasta `io-multer` criando um novo arquivo, pois ela já tem os pacotes necessários, e o `io-multer/index.js` como exemplo.

Solução

Copiar

```
// require('dotenv').config();
const express = require('express');
// const cors = require('cors');
// const bodyParser = require('body-parser');
const multer = require('multer');

// const { PORT } = process.env;

// const controllers = require('./controllers');
// const middlewares = require('./middlewares');

const app = express();

// app.use(
//   cors({
//     origin: `http://localhost:${PORT}`,
//     methods: ['GET', 'POST', 'PUT', 'DELETE'],
//     allowedHeaders: ['Authorization'],
//   })
// );

// app.use(bodyParser.json());
// app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static(__dirname + '/envios'));
```

```
const upload = multer({ dest: 'envios' });

app.post('/envios', upload.single('file'), (req, res) =>
  res.status(200).json({ body: req.body, file: req.file })
)

// app.get('/ping', controllers.ping);

// app.use(middlewares.error);

// app.listen(PORT, () => {
//   console.log(`App listening on port ${PORT}`);
// });
```

Exercícios

Agora, a prática

Exercício 1

Antes de começar, crie um projeto chamado `multer-exercises` utilizando o comando `npm init @tryber/backend multer-exercises`.

Depois de criar o projeto, instale o `multer` acessando a pasta e executando o comando `npm i multer` dentro dela.

Agora sim! ☒ Tudo pronto para começar os exercícios!

Crie o endpoint `POST /upload`

1. O endpoint deve receber apenas um arquivo no campo `file` ;
2. O arquivo deve ser armazenado na pasta `uploads` ;
3. O arquivo armazenado deve ter o timestamp do upload (obtido com `Date.now()`) seguido do nome original do arquivo. Exemplo, para o arquivo `profile.png` , o nome armazenado deve ser algo como `1616691266095-profile.png` , já que o timestamp será diferente a cada vez.
4. Retorne status `200 OK` se der tudo certo.

Solução

1. Vamos criar nosso controller `controllers/upload.js` .

Copiar

```
const upload = (req, res) => {  
}  
module.exports = upload;
```

2. Agora vamos adicionar o novo controller no `index.js` da pasta `controllers`.

Copiar

```
// const ping = require('./ping');  
const upload = require('./upload')  
  
// module.exports = {  
//   ping,  
//   upload  
// };
```

3. Adicione o endpoint `POST /upload` no seu `index.js`, passando o controller de upload.

Copiar

```
//app.get('/ping', controllers.ping);  
app.post('/upload', controllers.upload);
```

4. Importe o `multer` em seu arquivo `index.js` e logo após faça a configuração do `multer` e o storage de acordo com o que foi pedido.

Copiar

```
const multer = require('multer');  
//...  
  
const storage = multer.diskStorage({  
  destination: (req, file, callback) => { callback(null, 'uploads')  
},  
  filename: (req, file, callback) => { callback(null,  
    `${Date.now()}-${file.originalname}` )  
  })  
});  
  
const upload = multer({ storage });  
//app.get('/ping', controllers.ping);  
//app.post('/upload', controllers.upload);
```

O `destination` é a pasta onde vamos guardar os arquivos e o `filename` é a forma que usamos para alterar o nome do arquivo na hora de salvar. Não esqueça de criar a pasta `uploads` no seu projeto

5. Agora que configuramos o multer vamos adicionar o middleware dele em nosso endpoint de upload.

Copiar

```
//app.get('/ping', controllers.ping);  
app.post('/upload', upload.single('file'), controllers.upload);
```

Como é apenas um arquivo, vamos usar o single.

6. Configurar o retorno de nosso controller `controllers/upload.js` para retornar status code `200 OK`

Copiar

```
//const upload = (req, res) => {  
  return res.send()  
//}  
//module.exports = upload;
```

Exercício 2

Altere o endpoint `POST /upload` para que atenda o seguinte critério:

1. Apenas aceite arquivos cuja extensão seja `.png` ; Caso o arquivo tenha outro tipo de extensão, retorne o status `403 Forbidden` com o JSON a seguir:

Copiar

```
{  
  "error": { "message": "Extension must be `png`" }  
}
```

2. Não aceite um arquivo cujo nome (ignorando o timestamp) já exista na pasta `uploads` . Caso o arquivo já exista, retorne o status `409 Conflict` com o seguinte JSON:

Copiar

```
{  
  "error": { "message": "File already exists" }  
}
```

Solução

1. Para resolver o problema, usaremos o `fileFilter` do multer, então dentro do nosso `index.js` vamos criar uma função chamada `fileFilter` . Se quiser saber um pouco mais sobre, dá uma olhada na [documentação do multer](#) . Primeiro vamos resolver o problema da extensão do arquivo:

Copiar

```
const fileFilter = (req, file, cb) => {  
  
  if (file.mimetype !== 'image/png') {  
    //Colocar uma mensagem de erro na requisição  
    req.fileValidationError = true;  
  
    //Rejeitar o arquivo  
    return cb(null, false);  
  }  
  
  //Aceitar o arquivo  
  cb(null, true);  
}
```

Aqui vamos fazer a verificação se o arquivo tem a extensão `png` , se for a requisição segue normalmente, e o upload é feito, se o arquivo não for `png`, cancelamos o upload, e dentro do nosso request colocamos uma flag `fileValidationError= true` , e esse valor usaremos lá no controller.

2. Agora precisamos alterar o controller para retornar erro quando a extensão do arquivo for inválida.

Copiar

```
// const upload = (req, res) => {  
  if (req.fileValidationError)  
    return res.status(403).send({ error: { message: "Extension  
must be `png`" } });  
  // return res.send();  
  // }  
  
// module.exports = upload;
```

3. Para resolver o problema do arquivo duplicado primeiro vamos criar uma função responsável por fazer essa validação, nela teremos de entrada o nome do arquivo, e de saída um boolean informando se o arquivo já existe ou não. Chamarei essa função de `fileExists` , e

para facilitar a explicação deixarei ela no `index.js` , mas ela poderia ser um service.

Copiar

```
//Usaremos o 'fs' pois teremos que fazer a leitura de todos os arquivos do diretório.  
const fs = require('fs')  
  
const fileExists = (fileName) => {  
  //fs.readdirSync retorna uma lista com nome de todos os arquivos da pasta uploads.  
  const files = fs.readdirSync(`_${__dirname}/uploads`);  
  //Aqui usamos a função some, que retorna `true` se algum dos items do array passar no teste, no nosso caso o `file.includes`.  
  return files.some(file => file === fileName);  
}
```

4. Agora que criamos a verificação se o arquivo já existe, vamos para o `fileFilter` validar o nome do arquivo. Se ele já existir, cancelamos o upload, e colocamos uma outra flag no nosso `req` , chamaremos de `fileDuplicated` .

Copiar

```
// const fileFilter = (req, file, cb) => {  
  
  // if (file.mimetype !== 'image/png') {  
  //   //Colocar uma flag de erro na requisição  
  //   req.fileValidationError = true;  
  
  //   //Rejeitar o arquivo  
  //   return cb(null, false);  
  // }  
  
  if (fileExists(file.originalname)) {  
    //Colocar uma flag de erro na requisição  
    req.fileDuplicated = true;  
  
    //Rejeitar o arquivo  
    return cb(null, false);  
  }  
  
  // //Aceitar o arquivo  
  // cb(null, true);  
  // }
```

5. Agora para fechar a solução, vamos para o nosso controller, validar a nova possível flag `fileDuplicated` , e caso o arquivo seja duplicado vamos retornar `409 Conflict` .

Copiar

```
// const upload = (req, res) => {  
  if (req.fileDuplicated)  
    return res.status(409).send({ error: { message: "File already  
exists" } });  
  
  // if (req.fileValidationError)  
  //   return res.status(403).send({ error: { message:  
"Extension must be `png`" } });  
  // return res.send();  
// }  
  
// module.exports = upload;
```

Exercício 3

Torne a pasta `uploads` pública de forma que seja possível baixar os arquivos enviados anteriormente.

Solução

1. No seu `index.js` , use a configuração do `express.static` na pasta `uploads`

Copiar

```
//...  
app.use(express.static(`${__dirname}/uploads`));  
// app.use(middlewares.error);  
  
// app.listen(PORT, () => {  
//   console.log(`App listening on port ${PORT}`);  
// });
```

Bônus

Exercício 1

Crie o endpoint `POST /multiple`

1. Permita o upload de vários arquivos através do campo `files` ;
2. Salve cada arquivo na pasta `/uploads` com um nome aleatório, que será gerado pelo `multer` ;
3. Retorne uma lista dos arquivos enviados juntamente com a URL pela qual cada um está acessível. Exemplo:

Copiar

```
[
  { "file": "meuArquivo.txt", "url":
"http://localhost:3000/c3f20f8a1a72729883b88a96f405bbd0" },
  { "file": "arquivo1.png", "url":
"http://localhost:3000/7c76b101fd872f7fc12705eeba2ddd1c" },
  { "file": "profile.jpg", "url":
"http://localhost:3000/0ec57a65a9522aa14f9405060089c6f5" },
  { "file": "tcc.docx", "url":
"http://localhost:3000/a7de65196a12ce1c53e8e76927099f12" },
  { "file": "CNH.jpg", "url":
"http://localhost:3000/78c948b7b737d9a80b13f52bc6968d75" }
]
```

Solução

1. Vamos criar a base do nosso controller, chamaremos de `controllers/multiple.js`

Copiar

```
const multiple = (req, res) => {
}
module.exports = multiple;
```

2. Agora vamos adicionar o novo controller no `index.js` da pasta `controllers`.

Copiar

```
// const ping = require('./ping');
// const upload = require('./upload')
const multiple = require('./multiple');
// module.exports = {
//   ping,
//   upload,
//   multiple
// };
```

3. Adicione o endpoint `POST /upload` no seu `index.js`, passando o controller de upload.

Copiar

```
//app.get('/ping', controllers.ping);  
//app.post('/upload', controllers.upload);  
app.post('/multiple', controllers.multiple);
```

4. Agora que temos o controller pronto, vamos configurar um novo multer, agora ele não vai mudar o nome dos arquivos, e também não vai validar existência e extensão. Essa criação será feita no `index.js`.

Copiar

```
//...  
// const storage = multer.diskStorage({  
//   destination: (req, file, callback) => { callback(null,  
'uploads') },  
//   filename: (req, file, callback) => { callback(null,  
`${Date.now()}-${file.originalname}`) },  
// })  
  
// const upload = multer({ storage, fileFilter });  
  
const defaultStorage = multer.diskStorage({  
  destination: (req, file, callback) => { callback(null, 'uploads')  
})  
})  
  
const multiUpload = multer({ storage: defaultStorage })
```

5. Na nossa rota vamos adicionar o middleware do multer. E agora a forma de usar o multer muda um pouco, vamos usar o `multiUpload.array`

Copiar

```
// app.get('/ping', controllers.ping);  
// app.post('/upload', upload.single('file'), controllers.upload);  
app.post('/multiple', multiUpload.array('files'),  
controllers.multiple);
```

6. Voltando para o nosso controller, agora temos as informações sobre todos os arquivos no `req.files`, com essas informações nas mãos, vamos formatar de acordo com a saída pedida. Se quiser saber

todas as propriedades que tem em cada arquivo, coloca um `console.log(req.files);` no controller para ter mais detalhes.

Copiar

```
// const multiple = (req, res) => {  
  const uploadedFiles = req.files.map((file) => ({  
    file: file.originalname,  
    url: `http://localhost:3000/${file.path}`,  
  }));  
  
  return res.send(uploadedFiles);  
// };  
  
// module.exports = multiple;
```

Exercício 2

Crie o endpoint `POST /profile`

1. Receba strings nos campos `name` , `email` , `password` e `bio` ;
2. Receba um arquivo no campo `profilePic` ;
3. Armazene o arquivo recebido na pasta `/profilePics` com o nome aleatório do `multer` ;
4. Utilize o nome gerado pelo `multer` como ID para o perfil criado;
5. Armazene as informações do perfil no arquivo `profiles.json`

Solução

1. Crie e configura seu controller `POST controllers/profile.js` igual os exercícios passados.
2. Configure o middleware do `multer` em seu endpoint, importante, ele deve ler arquivo no campo `profilePic` e salvar no diretório `profilePics` . Aqui vamos montar de uma forma diferente, apenas para exemplo.

Copiar

```
//  
app.post('/profile', multer({ dest: 'profilePics' }  
)).single('profilePic'), controllers.profile);
```

3. Crie a pasta `profilePics` na raiz do seu projeto.
4. Crie o arquivo `profiles.json` no diretório do seu projeto, e dentro coloque apenas um array vazio.

Copiar



5. Com a base criada, vamos para o controller implementar a lógica de criar o profile. Primeiro passo é criar o objeto que iremos salvar no json. Para obter os outros dados que vem pelo form data, basta acessar pelo `req.body` e as informações do arquivo pelo `req.file` .

Copiar

```
const profile = (req, res) => {  
  const { name, email, passowrd, bio } = req.body;  
  
  const profileData = {  
    id: req.file.filename,  
    name,  
    email,  
    passowrd,  
    bio,  
  };  
};  
  
module.exports = profile;
```

6. Vamos ler as informações do arquivo `profile.json` , vamos adicionar o novo profile dentro, e logo após vamos salvar. Para facilitar a vida vou criar duas funções, `getProfileData` e `saveProfileData` . Não esqueça de importar o modulo `fs` para trabalhar com arquivos.

Copiar

```
const fs = require('fs');  
  
const FILE_PATH = `_${__dirname}/../profiles.json`;  
  
const getProfileData = () => {  
  const fileText = fs.readFileSync(FILE_PATH);  
  return JSON.parse(fileText);  
};  
  
const saveProfileData = (profiles) => {  
  fs.writeFileSync(FILE_PATH, JSON.stringify(profiles));  
};
```

7. Agora vamos montar o perfil e salvar no arquivo

Copiar

```
//const profile = (req, res) => {  
  const { name, email, passowrd, bio } = req.body;  
  
  const profileDate = {  
    id: req.file.filename,  
    name,  
    email,  
    passowrd,  
    bio,  
  };  
  
  const profiles = getProfileData();  
  profiles.push(profileDate);  
  saveProfileData(profiles);  
  return res.send({ profileDate });  
//};
```

Exercício 3

Crie o endpoint GET /profiles/:id

1. Caso exista um perfil com o **id** informado, retorne as informações desse perfil, conforme salvo no arquivo **profiles.json**
2. Caso não exista um perfil com o **id** informado, retorne o status **404 Not Found** com o seguinte corpo:

Copiar

```
{  
  "error": {  
    "message": "Perfil não encontrado"  
  }  
}
```

Solução

1. Primeiro passo é criar mais um endpoint **GET /profiles/:id** no nosso arquivo **controllers/profile.js**. Importante mudar a forma que exportamos o nosso controller, pois agora ele tem dois endpoints;

Copiar

```
//const profile = (req, res) => {...};  
const getProfile = (req, res) => {
```

```
};
```

```
module.exports = { profile, getProfile };
```

2. Agora vamos no nosso `index.js` do controller para ajustar o import do profile e adicionar o export do getProfile.

Copiar

```
// const ping = require('./ping');  
// const upload = require('./upload');  
// const multiple = require('./multiple');  
const { profile, getProfile } = require('./profile');
```

```
// module.exports = {  
//   ping,  
//   upload,  
//   multiple,  
//   profile,  
//   getProfile,  
// };
```

3. E agora vamos no nosso `index.js` configurar a rota

Copiar

```
//app.get('/ping', controllers.ping);  
// app.post('/upload', upload.single('file'), controllers.upload);  
// app.post('/multiple', multiUpload.array('files'),  
controllers.multiple);  
// app.post('/profile', multer({ dest: 'profilePics'  
}).single('profilePic'), controllers.profile);  
app.get('/profiles/:id', controllers.getProfile);
```

4. Voltamos para o nosso controller para obter o id do usuário e buscar na lista.

Copiar

```
//const getProfile = (req, res) => {  
  const profileId = req.params.id;  
  const profiles = getProfileData();  
  
  const profileResult = profiles.find((profile) => profile.id ===  
profileId);  
//};
```

5. Por fim teremos que configurar os retornos, se o perfil existir, retornamos 200 OK com o perfil, caso contrario, retornamos 404 Not Found

Copiar

```
//const getProfile = (req, res) => {  
//    const profileId = req.params.id;  
//    const profiles = getProfileData();  
//  
//    const profileResult = profiles.find((profile) => profile.id ===  
profileId);  
  
    if (profileResult) return res.send(profileResult);  
  
    return res.status(404).send({ error: { message: 'Perfil não  
encontrado' } });  
//};
```