```
function validateName(req, res, next) {
  const { name } = req.body;
  if (!name || name === '') return res.status(400).json({ message: 'Invalid
data!'});

  next();
};

app.post('/recipes', validateName, function (req, res) {
  const { id, name, price } = req.body;
  recipes.push({ id, name, price});
  res.status(201).json({ message: 'Recipe created successfully!'});
});

app.put('/recipes/:id', validateName, function (req, res) {
  const { id } = req.params;
  const { name, price } = req.body;
  const recipesIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipesIndex === -1)
    return res.status(404).json({ message: 'Recipe not found!' });

  recipes[recipesIndex] = { ...recipes[recipesIndex], name, price };

  res.status(204).end();
});
```

Você será capaz de:

- Aprender sobre o conceito de middlewares e encadeamento de middlewares usando a função `next` .
- Passar valores entre middlewares.
- Como organizar as rotas usando `Router` .
- Tratar erros com middleware de erros.

Conteúdos

ATENÇÃO: No conteúdo de hoje continuaremos a usar a API de receitas que fizemos no conteúdo anterior. Para ver o código desenvolvido vá para o tópico Conclusão do conteúdo do dia 26.4 - Node: HTTP com Express.

Middlewares

A primeira coisa que você precisa saber sobre middlewares é que, no Express qualquer função passada para uma rota é um middleware , seja de forma direta ou indireta. Como assim?
Para o Express, um middleware é uma função que realiza o tratamento de uma request e que pode encerrar essa request, ou chamar o próximo middleware.

Bom, para te contar um *segredo* : estamos usando middlewares desde o começo desse conteúdo, mas com outro nome! Até agora, nos referimos aos middlewares como `callback` ao falar sobre roteamento e definição de endpoints. Acontece que todos os callbacks que mostramos nessas rotas são middlewares.
Na prática, essas funções recebem três parâmetros: `req` , `res` e `next` , exatamente como as funções callback que usamos até agora para registrar rotas. Middlewares podem retornar *qualquer coisa* , incluindo Promises. O fato é que o Express ignora o retorno dos middlewares, visto que o importante é se aquele middleware chamou ou não um método que encerra a request, ou a função `next` .
Por exemplo, vamos considerar que temos o seguinte cenário onde na nossa API de CRUD de receitas precisamos validar se o nome não foi enviado vazio ao cadastrar uma nova receita.

Copiar

```
// ...
app.post('/recipes',
function (req, res, next) {
  const { name } = req.body;
  if (!name || name === '') return res.status(400).json({ message: 'Invalid
data!'}); // 1

  next(); // 2
},
function (req, res) { // 3
  const { id, name, price } = req.body;
  recipes.push({ id, name, price});
  res.status(201).json({ message: 'Recipe created successfully!'});
});
// ...
```

No exemplo acima, temos uma rota que utiliza dois middlewares, onde:

1. Fizemos uma validação que retorna uma resposta para requisição caso seja enviada no body da requisição um nome vazio. O middleware retorna uma resposta com status 400 e um json com uma mensagem dizendo que os dados enviados foram inválidos.
2. Caso não caia no `if` , este middleware endereça a requisição para o próximo middleware.
3. Esse middleware faz todo o processo de pegar os dados enviados, salvar em um array, e finalmente retornar uma mensagem de sucesso dizendo que o produto foi cadastrado.

Esse segundo middleware só é executado se o primeiro middleware chamar ele usando a função next. Experimente fazer a mesma requisição comentando a instrunção `next()` e você vai notar que a requisição nunca vai chegar na segunda rota.
Para fazer essa requisição use o comando abaixo ou uma das ferramentas para fazer requisições HTTP (Postman ou o Insomnia).

Copiar

```
# Essa requisição vai retornar { message: 'Invalid data!'}
http POST :3001/recipes price:=40
# Experimente chamar essa request com o código correto, e depois comentando o
next. A requisição não vai retornar uma resposta.
```

```
http POST :3001/recipes name=Macarronada price:=40
```

Uma das vantagens do Express suportar diversos middlewares é que podemos reaproveitar alguns deles para serem utilizados em diversas rotas. No nosso caso essa função que valida se o nome foi enviado poderia ser também aproveitada para a rota `PUT /recipes/:id` . Para isso vamos tirar a definição dessa função de dentro da rota `POST /recipes` e aplicá-la para ser usada nas duas rotas.

Copiar

```
// ...
function validateName(req, res, next) {
  const { name } = req.body;
  if (!name || name === '') return res.status(400).json({ message: 'Invalid data!'});

  next();
};

app.post('/recipes', validateName, function (req, res) {
  const { id, name, price } = req.body;
  recipes.push({ id, name, price});
  res.status(201).json({ message: 'Recipe created successfully!'});
});

app.put('/recipes/:id', validateName, function (req, res) {
  const { id } = req.params;
  const { name, price } = req.body;
  const recipesIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipesIndex === -1)
    return res.status(404).json({ message: 'Recipe not found!' });

  recipes[recipesIndex] = { ...recipes[recipesIndex], name, price };

  res.status(204).end();
});
// ...
```

Pronto. Agora o middleware que valida se o nome foi enviado foi isolado para uma função e conseguimos aplicá-la nas rotas para cadastrar e editar uma receita.
Para ficar nítido, todo middleware, pode receber o next como um terceiro parâmetro, mas geralmente no caso do último middleware de uma rota, que processa a resposta da requisição caso todos os middlewares anteriores não tenham encerrado o fluxo, não temos necessidade de usar o objeto `next` por isso podemos simplesmente receber apenas os objetos `req` e `res` .

Para Fixar

1. Crie uma função `validatePrice` para validar o preço foi enviado. O preço deve ser obrigatório, ser um número e não pode ser menor que 0. Aplique essa função como um middleware nas rotas `POST /recipes` e `PUT /recipes/:id` .
2.

Criando middlewares globais com app.use

Outra forma de utilizar middlewares é quando precisamos reaproveitar um middleware para todas as rotas da nossa aplicação (ou uma boa parte destas). Vamos criar uma forma de autenticar se um determinado usuário pode ter acesso a nossa API de receitas. Para isso, será necessário enviar as informações de nome de usuário e senha pelo Header da requisição (⚠️ Este é um exemplo didático, na prática vamos utilizar abordagens mais seguras de fazer esse tipo de autenticação, por exemplo utilizando JWT).
Vamos começar definindo nosso middleware em um arquivo separado: `auth-middleware.js` .

Copiar

```
/* auth-middleware.js */
const validUser = {
  username: 'MestreCuca',
  password: 'MinhaSenhaSuperSeguraSqn'
};

const authMiddleware = (req, res, next) => {
  const { username, password } = req.headers;

  if (!username && !password) {
    return res.status(401).json({ message: 'Username and password can`t be blank!'
});
  }

  if (username !== validUser.username || password !== validUser.password) {
    return res.status(401).json({ message: 'Invalid credentials!' });
  }

  next();
};

module.exports = authMiddleware;
```

No código acima temos um middleware que, ao receber uma requisição, verifica se ela possui no *header* as informações `username` e `password` . Se alguma das informações não foi enviada, esse middleware retorna uma mensagem dizendo que essas informações não podem ser vazias. Na sequência, é feita uma segunda verificação para checar se os valor de `username` e `password` são iguais aos valores pré-determinados no objeto `validUser` (Na prática, em uma aplicação de verdade, esse objeto validUser teria os valores vindo do banco de dados e não *hard-coded*).
Caso nenhuma dessas opções seja verdadeira, uma resposta é enviada ao client dizendo que não foi possível realizar a autenticação. Ao enviarmos uma resposta para o client, impedimos que qualquer outro middleware seja executado depois desse. Caso esteja tudo certo com o header, o middleware chama a função `next` que, basicamente, diz ao Express "ok, terminei aqui, pode chamar o próximo que disse que queria saber de requisições pra essa rota".
Para utilizarmos esse middleware de autenticação, vamos alterar o arquivo `index.js` .

Copiar

```
// const express = require('express');
// const bodyParser = require('body-parser');
const authMiddleware = require('./auth-middleware');

// const app = express();
// app.use(bodyParser.json());

// Esta rota não passa pelo middleware de autenticação!
app.get('/open', function (req, res) {
  res.send('open!')
});

app.use(authMiddleware);

// const recipes = [
//   { id: 1, name: 'Lasanha', preco: 40.0, tempoDePreparo: 30 },
//   { id: 2, name: 'Macarrão a Bolonhesa', preco: 35.0, tempoDePreparo: 25 },
//   { id: 3, name: 'Macarrão com molho branco', preco: 35.0, tempoDePreparo: 25 },
// ];
//
// function validateName(req, res, next) {
//   const { name } = req.body;
//   if (!name || name === '') return res.status(400).json({ message: 'Invalid
data!'});
//
//   next();
// };

// app.get('/recipes', function (req, res) {
//   res.status(200).json(recipes);
// });
//
// app.get('/recipes/pesquisar', function (req, res) {
//   const { name, maxPrice } = req.query;
//   const filteredRecipes = recipes.filter((r) => r.name.includes(name) && r.preco
< parseInt(maxPrice));
//   res.status(200).json(filteredRecipes);
// });
//
// app.get('/recipes/:id', function (req, res) {
//   const { id } = req.params;
//   const recipe = recipes.find((r) => r.id === parseInt(id));
//   if (!recipe) return res.status(404).json({ message: 'Recipe not found!'});
//
//   res.status(200).json(recipe);
// });
//
// app.post('/recipes', validateName, function (req, res) {
//   const { id, name, price } = req.body;
//   recipes.push({ id, name, price });
//   res.status(201).json({ message: 'Recipe created successfully!'});
```

```
// });
//
// app.put('/recipes/:id', validateName, function (req, res) {
//   const { id } = req.params;
//   const { name, price } = req.body;
//   const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));
//
//   if (recipeIndex === -1) return res.status(500).json({ message: 'Recipe not
found!' });
//
//   recipes[recipeIndex] = { ...recipes[recipeIndex], name, price };
//
//   res.status(204).end();
// });
//
// app.delete('/recipes/:id', function (req, res) {
//   const { id } = req.params;
//   const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));
//
//   if (recipeIndex === -1) return res.status(500).json({ message: 'Recipe not
found!' });
//
//   recipes.splice(recipeIndex, 1);
//
//   res.status(204).end();
// });
//
// app.all('*', function (req, res) {
//   return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
// });
//
// app.listen(3001);
```

Observe que adicionamos uma rota, antes do `app.use` . Aqui é importante destacar que o `app.use` só afeta as rotas que vem abaixo da sua definição. Ou seja, todas as rotas do nosso CRUD de receitas vão passar pelo middleware de autenticação, enquanto a rota `/aberto` não, por que foi definida antes da linha do `app.use` . Vamos testar: Tente fazer uma requisição para as rotas `GET /aberto` e `GET /recipes` .

Copiar

```
http GET :3001/aberto # execute apenas essa linha
> HTTP/1.1 200 OK
> Connection: keep-alive
> Content-Length: 55
> Content-Type: text/html; charset=utf-8
> Date: Sun, 22 Aug 2021 21:12:24 GMT
> ETag: W/"37-ZXNKqzv8YdcuUTIY0Egz9o2J97U"
> Keep-Alive: timeout=5
> X-Powered-By: Express
>
> Esta rota não passa pelo middleware de autenticação!
http GET :3001/recipes # execute apenas essa linha
> HTTP/1.1 401 Unauthorized
```

```
> Connection: keep-alive
> Content-Length: 60
> Content-Type: application/json; charset=utf-8
> Date: Sun, 22 Aug 2021 21:13:36 GMT
> ETag: W/"3c-p35mvWqky25aPCJVo0WioEMrIRQ"
> Keep-Alive: timeout=5
> X-Powered-By: Express
>
> {
>     "message": "Nome de usuário e senha não podem ser vazios"
> }
```

Para poder fazer a requisição para os nossos endpoints que começam com `/recipes` , precisamos mandar os dados de autenticação no body da requisição. Abaixo estão alguns exemplos.

Copiar

```
http GET :3001/recipes username:MestreCuca password:MinhaSenhaSuperSeguraSqn # listar receitas
http POST :3001/recipes username:MestreCuca password:MinhaSenhaSuperSeguraSqn nome=Churrasco id:=5 preco:=30 # cadastrar um novo receita
http POST :3001/recipes/2 username:MestreCuca password:MinhaSenhaSuperSeguraSqn nome=Lasanha preco:=45 # editar um receita
```

Para enviar parâmetros no header de uma requisição, utiliza-se o formato `<chave>:<valor>` enquanto no body da requisicão usa-se `<chave>=<valor>` ou `<chave>:=<valor>` como já vimos. No exemplo para request do tipo POST e PUT podemos ver como enviar informações no header e no body ao mesmo tempo.
Agora, entedemos como usar o `app.use` para criar middlewares genéricos, geralmente utilizados para operações de autenticação ou algum tipo de tratamento prévio dos dados recebidos na requisição. Agora que entendemos isso, vamos aprender como é possível enviar informações entre um middleware e outro.

Passando valores entre middlwares com objeto req

Middlewares também podem modificar o objeto `req` , e essas modificações serão recebidas pelos próximos middlewares, caso `next` seja chamado. Isso geralmente é utilizado para propagar informações de um middleware para o outro.
Por exemplo, vamos considerar que agora além de um único usuário válido para o nome de um restaurante temos vários usuários válidos, e ao cadastrar e editar queremos passar o objeto do usuário encontrado para os middlewares do CRUD terem acesso a esse usuário válido.

Copiar

```
/* auth-middleware.js */
const validUsers = [
  { username: 'MestreCuca', password: 'MinhaSenhaSuperSeguraSqn' },
  { username: 'McRonald', password: 'Senha123Mudar' },
  { username: 'Burger Queen', password: 'Senha123Mudar' },
  { username: 'UpWay', password: 'Senha123Mudar' },
];
```

```
const authMiddleware = (req, res, next) => {
  const { username, password } = req.headers;

  if (!username && !password) {
    return res.status(401).json({ message: 'Username and password can`t be blank!'
});
  }

  const foundUser = validUsers.find((user) => user.username === username);

  if (!foundUser) return res.status(401).json({ message: 'Invalid credentials!'
});

  if (!(username === foundUser.username  && password === foundUser.password)) {
    return res.status(401).json({ message: 'Invalid credentials!' });
  }

  req.user = foundUser; // Aqui estamos passando o usuário encontrado para o
próximo middleware.

  next();
};
```

```
module.exports = authMiddleware;
```

Vamos mudar na definição do nosso método de cadastrar uma receita para que ele tenha acesso ao objeto user que foi anexado ao objeto `req` para poder salvar o respectivo `username` desse usuário como um atributo do receita.

Copiar

```
// ...
app.use(authMiddleware);

// ...

app.post('/recipes', validateName, function (req, res) {
  const { id, name, price } = req.body;
  const { username } = req.user; // Aqui estamos acessando o usuário encontrado no
middleware de autenticação.
  recipes.push({ id, name, price, chef: username });
  res.status(201).json({ message: 'Recipe created successfully!'});
});
// ...
```

Observe que tivemos acesso ao objeto `req.user` que veio do nosso middleware `authMiddleware` . Dessa forma aproveitando o encadeamento entre middlewares conseguimos passar informações entre middleware sempre que for necessário. O objeto `req` praticamente aceita qualquer atributo que você quiser definir, só é preciso tomar cuidado para não sobrescrever nenhum dos atributos padrão (req.body, req.headers, req.params, req.query, etc).
Pacotes que são middlewares

Existem alguns pacotes que nos fornecem ferramentas necessárias para o desenvolvimento de nossas aplicações. Um exemplo disso é o módulo body-parser, que utilizamos ontem. Ele é um middleware que lê o corpo da request, cria nela uma propriedade body e coloca o conteúdo do corpo lá. Para utilizá-lo e ter acesso às informações do corpo da request, só precisamos instalá-lo com npm i body-parser e registrá-lo na nossa aplicação:
A função json() utilizada na linha `app.use(bodyParser.json());` diz ao body-parser que queremos um middleware que processe corpos de requisições escritos em JSON. Se executarmos nossa API script acima e fizermos uma requisição do tipo POST conseguimos ter acesso aos valores enviados no body da requisição. Porém se tirarmos o uso deste middleware, você irá perceber que as requisições do tipo POST não conseguem processar os dados enviados no body da requisição.
ℹ️ Faça o teste ℹ️ : Copie o script abaixo, cole-o em um arquivo chamado server.js e execute-o com o comando node server.js . Em seguida, abra o Postman ou o Insomnia e realize a request POST localhost:3000/hello , passando o JSON { "name": "<seu nome aqui">" } .

Copiar

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

app.use(bodyParser.json());

app.post('/hello', (req, res) => {
  // req.body agora está disponível
  res.status(200).json({ greeting: `Hello, ${req.body.name}!` });
});

app.listen(3000, () => { console.log('Ouvindo na porta 3000'); });
```

Experimente comentar a linha 5 do script, executar novamente o arquivo e realizar uma nova request para o endpoint POST /hello e perceba que, sem o body-parser , req.body é undefined.
Outro middleware bem comum de utilizarmos nas nossas aplicações back-end é o cors , que permite que nossa API receba requisições de outras aplicações, como por exemplo, uma aplicação front-end que consuma nossa API. O uso básico desse módulo é instalá-lo usando `npm i cors` e adicionando as seguintes linha no nosso código.

Copiar

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

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

Agora, qualquer requisição que você fizer de outra aplicação vai responder, pois temos o middleware `cors` . Caso não o tivéssemos, o navegador bloquearia as *requests* do nosso front-end para nossa API. O cors tem um conjunto de configurações que permitem criar regras específicas, de quem e como as requisições podem ser feitas. Por enquanto, não precisamos nos preocupar com isso já que estamos desenvolvendo aplicações apenas em ambiente de desenvolvimento. Porém é importante ter cuidado com essa configuração ao subir uma aplicação para ambiente de produção.
Para aprofundar-se em middlewares, assista a este vídeo.

Passando valores entre middlwares com objeto req

Middlewares também podem modificar o objeto `req` , e essas modificações serão recebidas pelos próximos middlewares, caso `next` seja chamado. Isso geralmente é utilizado para propagar informações de um middleware para o outro.
Por exemplo, vamos considerar que agora além de um único usuário válido para o nome de um restaurante temos vários usuários válidos, e ao cadastrar e editar queremos passar o objeto do usuário encontrado para os middlewares do CRUD terem acesso a esse usuário válido.

Copiar

```
/* auth-middleware.js */
const validUsers = [
  { username: 'MestreCuca', password: 'MinhaSenhaSuperSeguraSqn' },
  { username: 'McRonald', password: 'Senha123Mudar' },
  { username: 'Burger Queen', password: 'Senha123Mudar' },
  { username: 'UpWay', password: 'Senha123Mudar' },
];

const authMiddleware = (req, res, next) => {
  const { username, password } = req.headers;

  if (!username && !password) {
    return res.status(401).json({ message: 'Username and password can`t be blank!'
});
  }

  const foundUser = validUsers.find((user) => user.username === username);

  if (!foundUser) return res.status(401).json({ message: 'Invalid credentials!'
});

  if (!(username === foundUser.username  && password === foundUser.password)) {
    return res.status(401).json({ message: 'Invalid credentials!' });
  }

  req.user = foundUser; // Aqui estamos passando o usuário encontrado para o
próximo middleware.

  next();
};

module.exports = authMiddleware;
```

Vamos mudar na definição do nosso método de cadastrar uma receita para que ele tenha acesso ao objeto user que foi anexado ao objeto `req` para poder salvar o respectivo `username` desse usuário como um atributo do receita.

Copiar

```
// ...
app.use(authMiddleware);
```

```
// ...

app.post('/recipes', validateName, function (req, res) {
  const { id, name, price } = req.body;
  const { username } = req.user; // Aqui estamos acessando o usuário encontrado no middleware de autenticação.
  recipes.push({ id, name, price, chef: username });
  res.status(201).json({ message: 'Recipe created successfully!'});
});
// ...
```

Observe que tivemos acesso ao objeto `req.user` que veio do nosso middleware `authMiddleware` . Dessa forma aproveitando o encadeamento entre middlewares conseguimos passar informações entre middleware sempre que for necessário. O objeto `req` praticamente aceita qualquer atributo que você quiser definir, só é preciso tomar cuidado para não sobrescrever nenhum dos atributos padrão (req.body, req.headers, req.params, req.query, etc).

Pacotes que são middlewares

Existem alguns pacotes que nos fornecem ferramentas necessárias para o desenvolvimento de nossas aplicações. Um exemplo disso é o módulo body-parser, que utilizamos ontem. Ele é um middleware que lê o corpo da request, cria nela uma propriedade body e coloca o conteúdo do corpo lá. Para utilizá-lo e ter acesso às informações do corpo da request, só precisamos instalá-lo com npm i body-parser e registrá-lo na nossa aplicação:

A função json() utilizada na linha `app.use(bodyParser.json());` diz ao body-parser que queremos um middleware que processe corpos de requisições escritos em JSON. Se executarmos nossa API script acima e fizermos uma requisição do tipo POST conseguimos ter acesso aos valores enviados no body da requisição. Porém se tirarmos o uso deste middleware, você irá perceber que as requisições do tipo POST não conseguem processar os dados enviados no body da requisição.

ℹ️ Faça o teste ℹ️ : Copie o script abaixo, cole-o em um arquivo chamado server.js e execute-o com o comando node server.js . Em seguida, abra o Postman ou o Insomnia e realize a request POST localhost:3000/hello , passando o JSON { "name": "<seu nome aqui">" } .

Copiar

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');

app.use(bodyParser.json());

app.post('/hello', (req, res) => {
  // req.body agora está disponível
  res.status(200).json({ greeting: `Hello, ${req.body.name}!` });
});

app.listen(3000, () => { console.log('Ouvindo na porta 3000'); });
```

Experimente comentar a linha 5 do script, executar novamente o arquivo e realizar uma nova request para o endpoint POST /hello e perceba que, sem o body-parser , req.body é undefined.
Outro middleware bem comum de utilizarmos nas nossas aplicações back-end é o cors , que permite que nossa API receba requisições de outras aplicações, como por exemplo, uma aplicação front-end que consuma nossa API. O uso básico desse módulo é instalá-lo usando `npm i cors` e adicionando as seguintes linha no nosso código.

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

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

Agora, qualquer requisição que você fizer de outra aplicação vai responder, pois temos o middleware `cors` . Caso não o tivéssemos, o navegador bloquearia as *requests* do nosso front-end para nossa API. O cors tem um conjunto de configurações que permitem criar regras específicas, de quem e como as requisições podem ser feitas. Por enquanto, não precisamos nos preocupar com isso já que estamos desenvolvendo aplicações apenas em ambiente de desenvolvimento. Porém é importante ter cuidado com essa configuração ao subir uma aplicação para ambiente de produção.
Para aprofundar-se em middlewares, assista a este vídeo.

Lidando com erros

Até agora, falamos de middlewares comuns, que recebem `req` , `res` e `next` e tratam uma request caso tudo esteja correndo bem. Acontece que existe ainda um outro tipo de middleware: o *middleware de erro* .
A diferença de um middleware de erro para um middleware comum é que a assinatura dele recebe quatro parâmetros ao invés de três, ficando assim: `function (err, req, res, next) {}` .

Copiar
```
app.use(middleware1);
app.get('/', */ ... */);
app.use(function (err, req, res, next) {
  res.status(500).send(`Algo deu errado! Mensagem: ${err.message}`);
});
```

É importante notar que:

- Middlewares de erro sempre devem vir depois de rotas e outros middlewares ;
- Middlewares de erro sempre devem receber quatro parâmetros .

O Express utiliza a quantidade de parâmetros que uma função recebe para determinar se ela é um middleware de erro ou um middleware comum. Ou seja, mesmo que você não vá utilizar os parâmetros `req` , `res` ou `next` , seu middleware de erro precisa recebê-los .
Você pode adicionar um underline no começo do nome dos parâmetros que não vai usar. Isso é uma boa prática e sinaliza para quem está lendo o código que aquele parâmetro não é utilizado. Por exemplo: `function (err, _req, res, _next) {}` .
Também é possível encadear middlewares de erro, no mesmo esquema dos outros middlewares, simplesmente colocando-os na sequência em que devem ser executados.

Copiar

```
app.use(function logErrors(err, req, res, next) {
  console.error(err.stack);
  /* passa o erro para o próximo middleware */
  next(err);
});

app.use(function (err, req, res, next) {
  res.status(500);
  res.json({ error: err });
});
```

Repare que estamos fazendo `next(err)` na linha 4. Isso diz ao Express que ele não deve continuar executando nenhum middleware ou rota que não seja de erro. Ou seja, quando passamos qualquer parâmetro para o `next` , o `Express` entende que é um erro e deixa de executar middlewares comuns , passando a execução para o próximo middleware de erro registrado para aquela rota, router ou aplicação.
Esse detalhe é importante, pois se um erro acontece dentro de uma rota ou middleware e nós não o capturamos e o passamos para a função `next` , os middlewares de erro não serão chamados para tratar aquele erro. Isso quer dizer que nossa API ficará sem responder àquela requisição, ou até mesmo que o erro encerrará o processo do Node.
Por isso, lembre-se: Sempre realize tratamento de erros nas suas rotas e middlewares, passando o erro para a função `next` , caso necessário.
Um exemplo onde o erro fica "flutuando" e não existe resposta do servidor é quando utilizamos um middleware `async` . Como o Express não faz `.catch` na Promise retornada pelo middleware, ele não sabe que ocorreu um erro, a não ser que nós capturamos esse erro e o passemos para a função `next` .
Vamos usar como exemplo um método que lê um arquivo baseado em um parâmetro de rota enviado na requisição. Vamos fazer isso em um arquivo separado diferente dos exemplos anteriores que fizemos até agora.
⚠️ Atenção ⚠️: Jamais devemos realizar a leitura de um arquivo do sistema de arquivos dessa forma. Concatenar parâmetros recebidos do usuário diretamente na chamada para qualquer método representa uma falha gigantesca de segurança. Vamos fazer isso aqui nesse momento para fins didáticos. Repetindo: não tente isso em casa em produção!

Copiar

```
/* errorHandlerExample.js */
const express = require('express');
const fs = require('fs/promises');

const app = express();

app.get('/:fileName', async (req, res, next) => {
    try {
        const file = await fs.readFile(`./${req.params.fileName}`);
        res.send(file.toString('utf-8'));
    } catch (e) {
        next(e);
    }
});

app.use(function (err, req, res, next) {
```

```
  res.status(500).json({ error: `Erro: ${err.message}` });
});

app.listen(3002);
```

Nesse caso, tivemos que colocar as duas linhas que executam a leitura do arquivo dentro de uma estrutura `try/catch` , caso seja disparada alguma exceção, como no exemplo quando o arquivo não existe,o código cai dentro do catch, que por sua vez redireciona para o middleware de erro.
Para testar, execute essa nova API com o comando `node errorHandlerExample.js` e faça uma requisição para a URL `http://localhost:3002/abc` . A requisição vai retornar uma resposta similar a essa:

Copiar

```
{
  "error": "Erro: ENOENT: no such file or directory, open './abc'"
}
```

Agora, se você criar o arquivo e jogar o conteúdo, por exemplo, usando o comando `echo 'abc' > abc` e fizer a requisição de novo, a requisição vai retornar uma resposta com o conteúdo do arquivo.
⚠️ Atenção : O parâmetro passado para função next, é sempre um indicador que ele vai redirecionar para o middleware de erro, e não para passar um objeto qualquer entre dois middelwares, para fazer isso, como já vimos no conteúdo de hoje, usamos o objeto `req` .
Esse mesmo tipo de erro pode acontecer ao fazer uma query para um banco de dados, e ter várias possíveis falhas, como por exemplo: o banco não está respondendo a nosso pedido de conexão, temos uma query escrita errada, as credenciais de acesso ao banco estão erradas. Entre outras.
Para que não seja necessário ter que criar estruturas `try/catch` sempre que formos utilizar códigos que eventualmente podem disparar excessões podemos usar um pacote chamado `express-rescue` .

Pacote express-rescue

O pacote `express-rescue` está disponível no npm e nos ajuda com a tarefa de garantir que os erros sempre sejam tratados. Para utilizá-lo, primeiro faça a instalação usando o comando `npm i express-rescue`
Para adicionarmos os `express-rescue` , basta passarmos o nosso middleware como parâmetro para a função `rescue` que importamos do módulo. Essa função vai gerar um novo middleware que vai fazer o tratameto de erros da middleware sem precisarmos escrever o `try/catch` . Vamos refatorar o exemplo da seção anterior para usar o `express-rescue` .

Copiar

```
/* errorHandlerExample.js */
const express = require('express');
const rescue = require('express-rescue');
const fs = require('fs/promises');

const app = express();
```

```
app.get(
  '/:fileName',
  rescue(async (req, res) => {
    const file = await fs.readFile(`./${req.params.fileName}`);
    res.send(file.toString('utf-8'));
  })
);

app.use((err, req, res, next) => {
  res.status(500).json({ error: `Erro: ${err.message}` });
});

app.listen(3002);
```

O que o novo middleware faz é simplesmente executar nosso middleware original dentro de um bloco de `try ... catch` . Caso ocorra qualquer erro no nosso middleware, esse erro é capturado pelo `catch` e passado para o `next` , dando início ao fluxo de erro do Express. Faça os mesmos testes que fizemos no final da seção anterior e vai ver que o fluxo continua acontecendo da mesma forma, quando a excessão é disparada, a diferença é que nosso código ficou mais enxuto.
Através do uso correto de middlewares de erro, é possível centralizar o tratamento de erros da aplicação em partes específicas dela. Isso facilita a construção dos middlewares de rotas, pois você não precisa ficar tratando erros em todos esses middlewares. Se algo der errado em qualquer rota que estiver envelopada pelo `express-rescue` , esse erro vai ser tratado pelo middleware de erros mais próximo.
Por último, um padrão muito comum é ter um middleware de erro genérico, e outros middlewares que convertem erros para esse formato genérico. Por exemplo:

Copiar

```
/* errorMiddleware.js */

module.exports = (err, req, res, next) => {
  if (err.code && err.status) {
    return res.status(err.status).json({ message: err.message, code: err.code });
  }

  return res.status(500).json({ message: err.message });
}
```

O middleware acima verifica se o erro possui um código e um status HTTP . Caso possua, o código e a mensagem são devolvidas na response. Caso contrário um erro genérico de servidor é utilizado.

Copiar

```
/* index.js */
const express = require('express');
const rescue = require('express-rescue');
const errorMiddleware = require('./errorMiddleware');

const app = express();

app.get('/:fileName', [
  rescue(async (req, res) => {
```

```
    const file = await fs.readFile(`./${req.params.fileName}`);
    res.send(file.toString('utf-8'));
  })
  (err, req, res, next) => {
    if (err.code ==- 'ENOENT') {
      const newError = new Error(err.message);
      newError.code = 'file_not_found';
      newError.status = 404;
      return next(newError);
    }

    return next(err);
  },
]);
```

```
app.use(errorMiddleware);
```

Nesse trecho de código, convertemos um erro de leitura de arquivo para um erro que nosso middleware de erros conhece e sabe formatar. Dessa forma, nos middlewares comuns, precisamos nos preocupar apenas com o caminho feliz ao passo que, nos middlewares de erro, nos preocupamos apenas com o fluxo de erros.
Repare, também, que estamos utilizando um Array para passar mais de um middleware para uma mesma rota. Poderíamos passar cada middleware como um parâmetro, mas um Array deixa mais explícita a intenção de, realmente, utilizarmos vários middlewares numa mesma rota.

Conclusão

No conteúdo de hoje aprendemos o que são middlewares e diferente formas de associar um middleware com uma rota. Também entendemos como é possível passar valores entre middlewares e como organizar as rotas usando o recurso `Router` que permite quebrar uma aplicação express em partes menores, que ajuda bastante na organização do nosso código. Por fim, vimos como podemos tratar erros usando o middleware genérico de erro, e como escrever middlewares mais enxutos usando o `express-rescue` .
Todos esses conceitos vão ser essenciais para nossos próximos passos para desenvolver aplicações web, usando o NodeJS para construir APIs HTTP. Por isso é fundamental que você pratique bastante o desenvolvimento de APIs através dos exercícios e projeto desse bloco.

Exercícios

back-end

Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉

Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma `branch`, com arquivos e `commits` específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

```
$ git checkout main
```

6. `$ git pull`
7. A partir da main, crie uma `branch` com o nome `exercicios/26.5` *(bloco 26, dia 5)*
8. Copiar
9. `$ git checkout -b exercicios/26.5`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

```
$ mkdir back-end
```

12. `$ cd back-end`
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

```
$ mkdir bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
```

15. `$ cd bloco-26-introducao-ao-desenvolvimento-web-com-nodejs`
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-5-express-middlewares
```

18. `$ cd dia-5-express-middlewares`
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório `~/trybe-exercicios/back-end/block-26-introducao-ao-desenvolvimento-web-com-nodejs/dia-5-express-middlewares`. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/26.5
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`
26. Você pode visualizar o log de todos os commits já feitos naquela branch com `git log`.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.5)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e
coloquei pra quando clicar aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300
```

28. `    Resolvendo o exercício 1 usando eventListener`
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch `exercicios/26.5`

não vai existir no repositório remoto, então precisamos configurar o `remote` utilizando a opção `--set-upstream` (ou `-u`, que é a forma abreviada).

30. Copiar
31. `$ git push -u origin exercicios/26.5`
32. Após realizar o passo 9, podemos abrir a `Pull Request` a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com `-u` anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch `exercicios/26.5`, e disponíveis no repositório remoto do `GitHub`. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊🤜

Agora, a prática

Atividade 1

1. Crie uma rota `POST /user/register` que receba uma requisição que envie `username` , `email` e `password` no `body` da requisição, onde:
    1. `username` deve ter mais de 3 caracteres;
    2. `email` deve ter o formato email@mail.com;
    3. `password` deve conter no mínimo 4 caracteres e no máximo 8 (todos números);
    4. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "invalid data" }` ;
    5. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com uma mensagem de sucesso, ex: status `201` e `{ "message": "user created" }` ;

2. Crie uma rota `POST /user/login` que receba uma requisição que envie `email` / `password` no body da requisição e devolva um token como resposta, onde:
    1. O formato desse `token` retornado deve ser uma string aleatória com 12 caracteres;
    2. O `email` recebido deve ter o formato email@mail.com;
    3. O `password` deve conter no mínimo 4 caracteres e no máximo 8, todos números;
    4. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "email or password is incorrect" }`
    5. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com uma mensagem de sucesso, ex: status `200` e `{ "token": "86567349784e" }` ;

Dicas: separe suas rotas em arquivos e utilize middlewares para validar os campos recebidos nas requisições

---

Atividade 2:

3. Crie uma rota `GET /btc/price` que receba uma requisição com um token na chave `Authorization` do headers da requisição e verifique se ele está correto, onde:
    1. O `token` deve ser uma string de 12 caracteres contendo letras e/ou números.
    2. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `401` e `{ "message": "invalid token" }` ;
    3. Caso tenha sucesso deve ser feito um fetch em uma API externa de sua preferência e retorne os dados da API como resposta;

Dicas: - Sugestão de API (https://api.coindesk.com/v1/bpi/currentprice/BTC.json); - O token será passado pelo header da seguinte forma: authorization: 86567349784e; - Utilize middlewares para validar o token; - Para fazer uma requisição a uma API externa utilizer o FETCH ou AXIOS, parecido com que vimos em Front-end;

Atividade 3:

4. Crie uma rota `GET /posts/:id` que receba uma requisição com um id como `param route` , verifique existência do post com aquele id, onde:
    1. O `id` deve existir;
    2. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `404` e `{ "message": "post not found" }` ;

3. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com as informações do respectivo post;

5. Crie uma rota `GET /posts` que deve trazer todos os posts cadastrados, onde:
    1. Se não existir posts cadastrados retorne um array vazio e um status code, ex: status `200` e `{ "posts": [] }` ;
    2. Se existir posts cadastrados retorne um array com os posts e um status code;
6. Faça um middleware de erro. Caso tenha sido requisitada uma rota inexistente deve ser retornado o código de `status` e um `JSON` , ex: status `404` e `{ "message": "Opsss, route not found!" }`

Dicas: separe suas rotas em arquivos e utilize middleware de erro para capturar uma rota inexistente.

Atividade 4:

7. Crie uma rota `POST /teams` que receba uma requisição que envie `name` , `initials` , `country` e `league` no body da requisção, onde:
    1. `name` deve ter mais de 5 caracteres;
    2. `initials` deve conter no máximo 3 caracteres em caixa alta;
    3. `country` deve ter mais de 3 caracteres;
    4. `league` este campo é opcional;
    5. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "invalid data" }` ;
    6. Caso tenha sucesso deve ser gravado em um arquivo o dado recebido e retornado uma resposta com o código de `status` e um `JSON` com as informações do time criado;
8. Na rota `GET /teams` deve trazer todos os times cadastrados, onde:
    1. Se não existir times cadastrados retorne um array vazio e um status code, ex: status `200` e `{ "teams": [] }` ;
    2. Se existir times cadastrados retorne um array com os times e um status code;

Dicas: separe suas rotas em arquivos e para gravar/ler dados do arquivo, utilize o módulo FS do Node.js (Não esqueça de criar o arquivo teams.json na raiz do projeto)

Gabarito dos exercícios

A seguir encontra-se uma sugestão de solução para os exercícios propostos.

Atividade 1

1. Crie uma rota `POST /user/register` que receba uma requisição que envie `username` , `email` e `password` no body da requisição, onde:
    1. `username` deve ter mais de 3 caracteres;
    2. `email` deve ter o formato email@mail.com;
    3. `password` deve conter no mínimo 4 caracteres e no máximo 8 (todos números);
    4. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "invalid data" }` ;
    5. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com uma mensagem de sucesso, ex: status `201` e `{ "message": "user created" }` ;
2. Crie uma rota `POST /user/login` que receba uma requisição que envie `email` / `password` no body da requisição e devolva um token como resposta, onde:
    1. O formato desse `token` retornado deve ser uma string aleatória com 12 caracteres;
    2. O `email` recebido deve ter o formato email@mail.com;
    3. O `password` deve conter no mínimo 4 caracteres e no máximo 8, todos números;
    4. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "email or password is incorrect" }`
    5. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com uma mensagem de sucesso, ex: status `200` e `{ "token": "86567349784e" }` ;

Dicas: separe suas rotas em arquivos e utilize middlewares para validar os campos recebidos nas requisições

Resolução

./middlewares/validations.js

Copiar

```
// middlewares/validations.js
const isValidUsername = (req, res, next) => {
const { username } = req.body;

if(!username || username.length < 3)
  return res.status(400).json({ message: 'invalid data' });

  next();
};

const isValidEmail = (req, res, next) => {
  const { email } = req.body

  if(
```

```
    !email ||
    !email.includes('@') ||
    !email.includes('.com')
  )
    return res.status(400).json({ message: 'invalid data' });

  next();
};

const isValidPassword = (req, res, next) => {
  const { password } = req.body;
  const passwordRegex = /^[0-9]*$/;

  if(
    password.length < 3 ||
    password.length > 8 ||
    !password.match(passwordRegex)
  )
    return res.status(400).json({ message: 'invalid data' });

  next();
};

module.exports = {
  isValidUsername,
  isValidEmail,
  isValidPassword,
};
```

./routers/userRouter.js

Copiar

```
// routers/userRouter.js
const router = require('express').Router();
const {
  isValidUsername,
  isValidEmail,
  isValidPassword,
} = require('../middlewares/validations');

router.post(
  '/register',
  isValidUsername,
  isValidEmail,
  isValidPassword,
  (_req, res) => res.status(201).json({ message: 'user created' }),
);

router.post(
  '/login',
  isValidEmail,
  isValidPassword,
  (_req, res) => res.status(200).json({ token: '86567349784e' })
```

```
);

module.exports = router;
```

./index.js

Copiar

```
// index.js
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');

const userRouter = require('./routers/userRouter');
const PORT = 3000;

const app = express();
app.use(cors());
app.use(bodyParser.json());

app.use('/user', userRouter);

app.listen(PORT, () => console.log('Run server http://localhost:3000'));
```

Atividade 2:

3. Crie uma rota `GET /btc/price` que receba uma requisição com um token na chave `Authorization` do headers da requisição e verifique se ele está correto, onde:
    1. O `token` deve ser uma string de 12 caracteres contendo letras e/ou números.
    2. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `401` e `{ "message": "invalid token" }` ;
    3. Caso tenha sucesso deve ser feito um fetch em uma API externa de sua preferência e retorne os dados da API como resposta;

Dicas: - Sugestão de API (https://api.coindesk.com/v1/bpi/currentprice/BTC.json); - O token será passado pelo header da seguinte forma: authorization: 86567349784e; - Utilize middlewares para validar o token; - Para fazer uma requisição a uma API externa utilizer o FETCH ou AXIOS, parecido com que vimos em Front-end;

Resolução

./middlewares/validations.js

Copiar

```
// middlewares/validations.js

const isValidToken = (req, res, next) => {
const token = req.headers.authorization;
```

```
const tokenRegex = !/^[a-zA-Z0-9]{12}$/;

if (!token || tokenRegex.test(token))
  return res.status(401).json({ message: 'invalid token' });

  next();
};

module.exports = { isValidToken };
```

./index.js

Copiar

```
// index.js

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const axios = require('axios');
const { isValidToken } = require('./middleware/validation');
const PORT = 3000;
const ENDPOINTEXTERNALAPI =
'https://api.coindesk.com/v1/bpi/currentprice/BTC.json';

const app = express();
app.use(cors());
app.use(bodyParser.json());

app.get(
  '/btc',
  isValidToken,
  async (_req, res) => {
    const result = await axios.get(ENDPOINTEXTERNALAPI);

    res.status(200).json(result.data);
  }
);

app.listen(PORT, () => console.log('Run server http://localhost:3000'));
```

Atividade 3:

4. Crie uma rota `GET /posts/:id` que receba uma requisição com um id como `route param` , verifique existência do post com aquele id, onde:
    1. O `id` deve existir;
    2. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `404` e `{ "message": "post not found" }` ;

    3. Caso tenha sucesso deve ser retornado uma resposta com o código de `status` e um `JSON` com as informações do respectivo post;
5. Crie uma rota `GET /posts` que deve trazer todos os posts cadastrados, onde:
    1. Se não existir posts cadastrados retorne um array vazio e um status code, ex: status `200` e `{ "posts": [] }` ;
    2. Se existir posts cadastrados retorne um array com os posts e um status code;
6. Faça um middleware de erro. Caso tenha sido requisitada uma rota inexistente deve ser retornado o código de `status` e um `JSON` , ex: status `404` e `{ "message": "Opsss router not found" }`

Dicas: separe suas rotas em arquivos e utilize middleware de erro para capturar uma rota inexistente.

Resolução

./middlewares/routerNotFound.js

Copiar

```
// middlewares/routerNotFound.js

const routerNotFound = (err, _req, res, _next) =>
  res.status(err.statusCode).json({ message: err.message })

module.exports = { routerNotFound };
```

./routers/postRouter.js

Copiar

```
// routers/postRouter.js

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

const posts = [
  { id: 1, author: 'José Neto', comment: 'Mais um dia de Sol !' },
  { id: 2, author: 'Antonio Neto', comment: 'Hoje o dia está maneiro!' },
  { id: 3, author: 'Rodrigo Garga', comment: 'To aqui também' },
]

router.get('/', (_req, res) => res.status(200).json({ posts }));

router.get('/:id', (req, res, next) => {
  const { id } = req.params;
    const post = posts.find((item) => item.id === parseInt(id));

    if (!post)
    return next({ statusCode: 404, message: 'post not found' });

    res.status(200).json(post);
});

module.exports = router;
```

./index.js

Copiar

```
// index.js

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const postRouter = require('./routers/postRouter');
const errors = require('./middlewares/routerNotFound');
const PORT = 3000;

const app = express();
app.use(cors());
app.use(bodyParser.json());

app.use('/posts', postRouter);

app.use('*', (_req, _res, next) => next({ statusCode: 404, message: 'Opsss router not found' }));
app.use(errors.routerNotFound);

app.listen(PORT, () => console.log('Run server http://localhost:3000'));
```

Atividade 4:

7. Crie uma rota `POST /teams` que receba uma requisição que envie `name` , `initials` , `country` e `league` no body da requisição, onde:
    1. `name` deve ter mais de 5 caracteres;
    2. `initials` deve conter no máximo 3 caracteres em caixa alta;
    3. `country` deve ter mais de 3 caracteres;
    4. `league` este campo é opcional;
    5. Para todos os casos não atendidos acima deve ser retornado o código de `status` e um `JSON` com uma mensagem de erro, ex: status `400` e `{ "message": "invalid data" }` ;
    6. Caso tenha sucesso deve ser gravado em um arquivo o dado recebido e retornado uma resposta com o código de `status` e um `JSON` com as informações do time criado;
8. Na rota `GET /teams` deve trazer todos os times cadastrados, onde:
    1. Se não existir times cadastrados retorne um array vazio e um status code, ex: status `200` e `{ "teams": [] }` ;
    2. Se existir times cadastrados retorne um array com os times e um status code;

Dicas: separe suas rotas em arquivos e para gravar/ler dados do arquivo, utilize o módulo FS do Node.js (Não esqueça de criar o arquivo teams.json na raiz do projeto)

Resolução

./middlewares/validations.js

Copiar

```
// middlewares/validations.js
const isValid = (req, res, next) => {
  const { name, initials, country } = req.body;

  if(
    !name || name.length < 5 ||
    !initials || initials.length > 3 ||
    !country || country.length < 3
  )
    return res.status(400).json({ message: 'invalid data' });

  next();
};

module.exports = { isValid };
```

./routers/teamRouter.js

Copiar

```
// routers/teamRouter.js

const router = require('express').Router();
const {
    readContentFile,
  writeContentFile,
} = require('../helpers/readWriteFile');
const validations = require('../middlewares/validations');
const PATH_FILE = './teams.json';

router.get('/', async(_req, res) => {
    const teams = await readContentFile(PATH_FILE) || [];

    res.status(200).json({ teams });
});

router.post('/', validations.isValid, async(req, res,) => {
    const newTeam = {
        ...req.body,
        initials: req.body.initials.toUpperCase(),
    };
    const team = await writeContentFile(PATH_FILE, newTeam);

    res.status(200).json(team);
});

module.exports = router;
```

./helpers/readWriteFile.js

Copiar

```
// helpers/readWriteFile.js

const fs = require('fs').promises;

const readContentFile = async (path) => {
  try {
    const content = await fs.readFile(path, 'utf8');
    return JSON.parse(content);
  } catch (error) {
    return null;
  }
};

const writeContentFile = async (path, content) => {
  try {
    const arrContent = await readContentFile(path);

    arrContent.push(content);
    await fs.writeFile(path, JSON.stringify(arrContent));

    return content;
  } catch (error) {
    return null;
  }
};

module.exports = {
  readContentFile,
  writeContentFile,
};
```

./index.js

Copiar

```
// index.js

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const teamRouter = require('./routers/teamRouter');
const PORT = 3000;

const app = express();
app.use(cors());
app.use(bodyParser.json());

app.use('/teams', teamRouter);

app.listen(PORT, () => console.log('Run server http://localhost:3000'));
```