O que vamos aprender?

Nesse bloco, vamos nos aprofundar em um dos padrões mais famosos ao se construir uma API: o REST !
E vamos também aprender como aplicar esse padrão nas nossas APIs.

Você será capaz de:

- Entender e aplicar os padrões REST;
- Escrever assinaturas para APIs intuitivas e facilmente entendíveis.

Por que isso é importante?

O padrão REST vai delimitar como sua API deve se comportar ao se comunicar com o mundo.
Pense nisso como a "etiqueta" da sua API, tipo aquele negócio de usar a faca certa pra comer o peixe, sabe?
Saber essas práticas vai fazer total diferença quando pessoas forem integrar seus endpoints nas aplicações que elas forem construir. Da mesma maneira, vai facilitar quando você estiver do outro lado, fazendo a integração. 🙂

REST

Antes de começarmos, vamos entender a diferença entre REST e RESTful:
RESTful é, basicamente, um web service que segue as regras definidas pelo padrão REST.
Pronto, agora vamos à definição formal:
Representational State Transfer (REST), em português Transferência de Estado Representacional, é um estilo de arquitetura de software, controlado pelo `W3C` , que define um conjunto de restrições a serem usadas para a criação de APIs.
Uma maneira interessante de explicar é a seguinte:
Imagine que você está em um jantar. Existem algumas coisas que você deveria fazer, como:

- Comer com a boca fechada;
- Não colocar os cotovelos na mesa;
- Usar os talheres corretamente;
- Não arrotar.

O REST deve ser visto da mesma forma: um conjunto de boas práticas. Quando você está construindo uma API, existem algumas normas que você deve seguir para ser "educado" (RESTful).
Para o REST, uma aplicação é um conjunto de recursos que podem ter seu estado representado de alguma forma. Ao consumir esses recursos, você está transferindo as informações sobre esse estado para o cliente (uma requisição `GET` , por exemplo) ou fazendo uma alteração nele (um `POST` , `PUT` ou `DELETE`). Daí o nome *Transferência de Estado Representacional* , ou seja, estamos transferindo uma representação do estado de algum recurso.

Mas o que é um Recurso, afinal?

Um recurso, em REST, é uma abstração da informação. Dito isso, qualquer coisa que possa ser nomeada pode ser um recurso. Por exemplo:

- Usuários;
- Produtos;
- Compras;
- Etc.

Todos os exemplos acima podem ser considerados recursos.
No universo de Star Wars existem vários planetas. Na SWAPI , podemos chamar um endpoint para listar todos os planetas ou apenas um. Nesse caso, um planeta é um recurso , e planetas é uma coleção de recursos .

As 6 restrições para ser RESTful

A arquitetura REST define seis restrições, chamadas constraints , que devem ser respeitadas para que sua API seja `RESTful` .

1 - Interface uniforme (*Uniform Interface*)

A interface de comunicação entre seu servidor e seu cliente deve ser definida e seguida à risca, através de um padrão, para que ela se mantenha consistente. Dessa forma, essa "constraint", se seguida à risca, simplifica e desacopla a sua arquitetura.
Essa interface inclui o endpoint , o tipo de retorno e o uso dos verbos HTTP .

Recursos e coleções

O recurso a ser acessado/alterado deve ser identificado pelo endpoint da requisição. Exemplo: `https://swapi.dev/api/planets/:id` . Nessa URL, vemos que o recurso que queremos acessar, `planet` , é facilmente identificado. Usar plural ou singular? Não importa. O importante é manter o padrão.

Tipo do retorno

Talvez você já tenha visto um header chamado `Content-type` nas respostas de requisições. Ele serve para dizer, para o nosso cliente, que tipo de conteúdo estamos retornando.
Se estamos retornando um JSON, enviamos o header como `Content-type: application/json` . Se fosse HTML, seria `Content-type: text/html` , e por aí vai.
Alguns formatos comuns são `JSON` , `XML` e `JavaScript` .
Esse tópico é, literalmente, sobre manter esses retornos consistentes. Se o cliente pede ou envia informação no formato JSON, você deve processar e retornar mantendo o formato JSON. Se um erro em um endpoint retorna os campos `code` , `error` e `message` , todos os erros devem retornar, pelo menos, esses campos. Se uma requisição ao endpoint de uma coleção (`GET /posts` , por exemplo), retorna um Array, todos os endpoints de coleção devem retornar Arrays. Se, por exemplo, quando realizamos uma requisição `GET /products` , recebemos um array de produtos, ao realizar a requisição `GET /sales` , não devemos receber um JSON no formato `{ "sales": [{ ... }] }` , já que esse comportamento é inconsistente com o do endpoint `GET /products` .
Dessa forma, ao consumir um endpoint da sua API, é possível até mesmo deduzir o comportamento dos demais endpoints, dispensando "tentativa e erro".

Ações/Verbos

A ação que vamos realizar no recurso deve ser identificada pelo verbo HTTP da requisição. Para o REST, os principais verbos HTTP são `POST` , `GET` ,

`PUT` e `DELETE` , e cada um realiza uma ação, dependendo se for enviado para o endpoint de um recurso ou de uma coleção .
As tabelas abaixo relacionam cada verbo com sua ação em caso de coleções ou recursos:

Na collection /products

Exemplo	CRUD	Ação	Status	Response Body
POST /products	Create	Cria um novo produto	201 Created	Dados do produto recém criado
GET /products	Read	Lista todos os produtos	200 OK	Lista de produtos
PUT /products	Update	-	405 Method Not Allowed	Vazio
DELTE /products	Delete	-	405 Method Not Allowed	Vazio

Verbos HTTP e ações em coleções REST

No recurso /products/1

Exemplo	CRUD	Ação	Status	Response Body
POST /products/1	Create	-	405 Method Not Allowed	Vazio
GET /products/1	Read	Obtém detalhes do produto	200 OK	Dados do produto1
PUT /products/1	Update	Atualiza o produto	200 OK	Dados atualizados do produto1
DELTE /products/1	Delete	Exclui o produto	204 No Content	Vazio

Verbos HTTP e ações em recursos REST

Respostas

Respostas são sempre obrigatórias. Nunca deixe seu cliente sem resposta, mesmo que ela não tenha um corpo.
Existem boas práticas em relação aos `status code` que nosso servidor envia como resposta. Temos uma variedade de códigos que devemos utilizar em situações específicas:

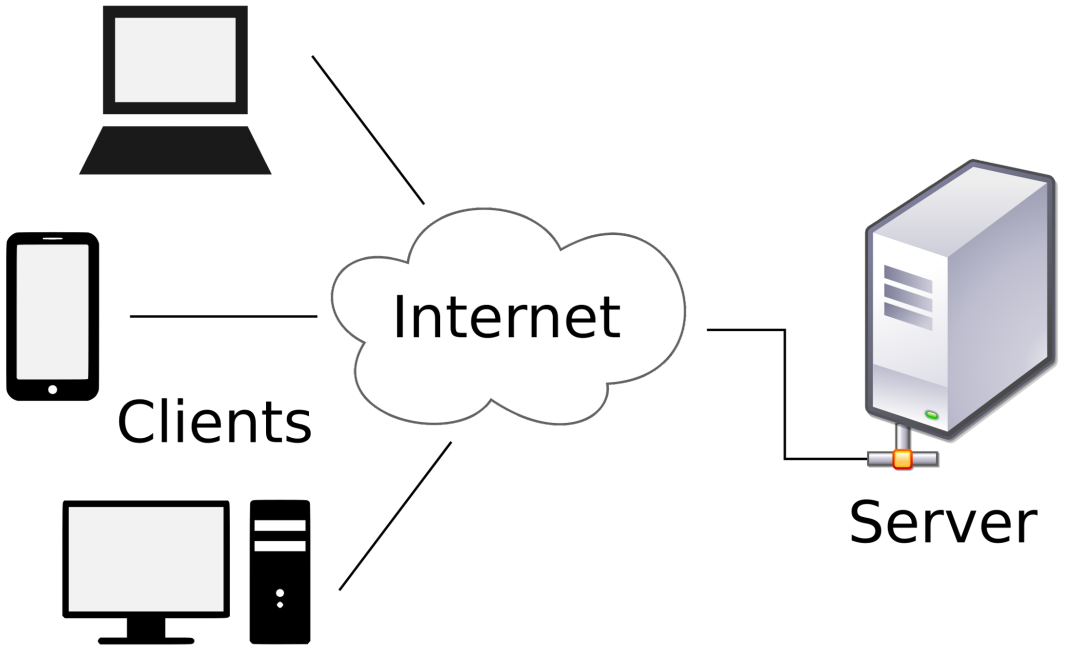
- 1xx: Informação;
- 2xx: Sucesso;
- 3xx: Redirecionamento;
- 4xx: Erro do cliente;
- 5xx: Erro no servidor.

Existe uma lista completa e detalhada sobre códigos de status HTTP disponibilizada pela Mozilla.

2 - Arquitetura cliente-servidor

Você já ouviu falar muito de arquitetura cliente-servidor, não é? De termos uma API e um cliente desacoplados? É exatamente o que o REST prega.

Não importa quem é nosso cliente, as nossas APIs têm que conseguir retornar dados para ele. O cliente pode ser um app mobile, web, tv, arduíno etc.
Lembra-se dos projetos que você fez no módulo de front-end, em que você consumia APIs externas, como a SWAPI ? Então, você poderia facilmente mudar as APIs com que esses projetos se comunicam para outras, desde que os contratos (os endpoints, os formatos das requisições, o que é retornado etc.) sejam mantidos.
Imagine como seria se o Mercado Livre não tivesse um padrão na API deles. Seria um caos para conseguir integrar no front-end que estávamos construindo.
O princípio básico aqui é a separação de responsabilidades em duas partes. O cliente se preocupa com a exibição dos dados, experiência da pessoa usuária, etc. O servidor se preocupa com armazenamento e acesso dos dados, cache, log e por aí vai.
Isso permite que as duas partes se desenvolvam de forma independente. Você pode trocar o cliente ou adicionar um novo sem mudar nada no servidor. Você pode mover o servidor para uma outra plataforma, escalá-lo etc. Pode até mesmo mudar completamente sua arquitetura interna, desde que a API que você fornece para seus clientes não mude (endpoint, resposta etc.).



Arquitetura cliente-servidor

3 - Sem estado (*Stateless*)

Essa constraint é um pouco mais complicada, mas nós temos um bom exemplo no React! Lembra-se quando criamos componentes que não tinham estado, e apenas recebiam props?
Esse é um dos conceitos mais importantes do REST. Ele que vai tornar possível nossa API responder a múltiplos clientes.
Não manter estado significa que toda requisição deve conter todas as informações necessárias (ser autossuficiente) para nossa API realizar uma ação . Desse jeito, não podemos reutilizar nenhum contexto que está armazenado no servidor (uma variável, por exemplo).
Exemplo :
Em um app em que você faz uma requisição para se logar, o servidor inicia sua sessão e te retorna um token.
Na próxima requisição, você precisa enviar o token novamente, pois o servidor "não se lembra" de você.
Esse ponto é importante, pois nos dá alguns benefícios:

- Transparência : facilita o trabalho do lado do servidor, pois todas as informações necessárias já estão na requisição;
- Escalabilidade : sem precisar manter estado, nosso servidor pode desalocar recursos que foram alocados para realizar uma ação específica e alocá-los apenas quando necessário.

Obs: No próximo bloco, vamos aprender sobre JWT, um padrão de autenticação sem estado, e veremos como o conceito *Stateless* funciona na prática.

4 - *Cacheable*

Primeiro, precisamos saber o que é `cache` . Cache é simplesmente um "depósito de informações". O exemplo mais próximo disso é quando seu navegador armazena imagens e arquivos para não precisar pedi-los para o servidor novamente toda vez que você revisitar uma página.
Esse cache é feito no lado do cliente, no browser.
O princípio aqui é que as respostas dadas pela nossa API devem dizer, explicitamente, se podem ou não ser cacheadas e por quanto tempo .
Com isso, evita-se que clientes forneçam respostas "velhas" ou inapropriadas.
Vale ressaltar que o cache deve ser usado sabiamente. Usá-lo demais faz sua API perder a confiabilidade, e usá-lo de menos pode sobrecarregar seu servidor desnecessariamente.

Uma camada de cache bem gerenciada pode reduzir ou eliminar iterações do lado do cliente, aumentando a escalabilidade e a performance da nossa API.
No HTTP, o cache é definido pelo header `Cache-Control: max-age=120` . Nesse caso, o cliente "cacheia" a resposta da requisição por 120 segundos. Durante esse tempo, o cliente fornecerá a resposta cacheada, sem precisar consultar o servidor.

5 - Sistema em camadas (*Layered System*)

No caso do REST, essa divisão em camadas não tem nada a ver com a organização do nosso código diretamente. Esse princípio é sobre abstrair do cliente as camadas necessárias para responder a uma requisição . Não importa se voce tem uma "API A" que se comunica com a "API B" que se comunica com uma fila ou um arquivo num "local C", ou se consulta um banco de dados, ou se esse banco de dados é local ou está armazenado em outro lugar. Quem consome a API não precisa saber de onde essas coisas estão vindo. Só precisa receber a resposta esperada.

6 - Código sob demanda (*Code on Demand*)

Esse princípio consiste em dar a possibilidade de o nosso servidor enviar código (JavaScript, por exemplo) ao nosso cliente, onde será executado. Assim, você consegue customizar o comportamento do cliente.
Um exemplo prático: enviar um "widget" para colocar na página um chat para que o cliente possa conversar com alguém.
Você não precisa implementar código sob demanda para ser RESTful, logo esse item é considerado opcional.

Ser ou não ser, eis a questão

Agora que vimos as restrições do REST, vale ressaltar: Você não precisa aplicar todas elas, a não ser que você realmente queira ser RESTful! Cenários diferentes exigem soluções diferentes. Em Software, nada é escrito em pedra. Os princípios podem ser quebrados, desde que a justificativa para tal seja plausível.

REST no Express

De maneira geral, usar o Express ou qualquer outro framework não deve fazer muita diferença. Os princípios devem ser seguidos independente da tecnologia que você usar na implementação da sua API. Ela pode ser escrita em Node.js, em Python, em Perl, tanto faz.
Uma das vantagens de se usar o Express para construção de APIs é a organização das rotas. Podemos definir N controllers (funções callback que lidam com as requisições) para a mesma rota, separando-as apenas pelo verbo HTTP da requisição.
Além disso, é simples retornar um formato específico solicitado pelo cliente, da mesma maneira que é simples retornar um status HTTP.

Copiar

```
app.route('/user')
  .get((req, res) => {
    // Realiza uma operação
    res.status(401).send({
      message: 'Usuário não autorizado'
    })
  })
  .post(...)
  .put(...)
  .delete(...)
```

Agora veremos na prática como trabalhar com REST no Express.

VIDEO: REST no Express - 16:283

Consumindo APIs

Até agora, você já consumiu APIs algumas vezes em diversos projetos como Carrinho de Compras , Front-end Online Store , Jogo de Trivia e por aí vai. No entanto, todas as vezes que precisou integrar seu front-end com alguma API, o único método utilizado foi o `GET` . Pois bem, chegou a hora de utilizar os demais métodos que aparecem no REST!
Para continuarmos fazendo requisições para APIs externas, vamos utilizar a biblioteca `axios` , que implementa no Node.JS, as mesmas funcionalidades que foram vistas nos módulos de Fundamentos e Front-end usando o `fetch` . Neste exemplo, vamos consumir a Postman Echo API , que é uma API mantida pelo *Postman* para fins de aprendizagem. Ela simplesmente retorna para nós tudo o que enviamos para ela.

Vamos começar criando nosso pacote Node.js e instalando o `axios` .
Faremos isso numa nova pasta chamada `hello-http-methods` .
Execute o seguinte comando no terminal

Copiar
```
mkdir hello-http-methods && cd hello-http-methods && npm init -y &&
npm i axios
```

Agora, crie o arquivo `index.js` , onde vamos escrever nosso código:

Copiar
```
// index.js

const axios = require('axios').default;

// Para aquecer, vamos começar com uma requisição do tipo `GET`
axios.get('https://postman-echo.com/get?param1=teste')
  .then((response) => {

    // Caso esteja tudo OK, retornamos os dados
    // As informações são retornas dentro da propriedade "data"
quando usamos axios
    return response.data;
  })
  .then((data) => {
    // Por fim, escrevemos o body no console
    console.log(data);
  })
  .catch((errorOrResponse) => {
    // Em caso de falha simples (a request completou com um status
diferente de 2xx)
    // simplesmente logamos o status no console
    if (errorOrResponse.status) {
      return console.error(`Request failed with status
${errorOrResponse.status}`);
    }

    // Caso tenha acontecido um erro de rede (não foi possível
completar a request)
    // logamos o erro todo
    console.error(errorOrResponse);
  });
```

Executando o código acima, temos o seguinte resultado no terminal:

Copiar

```
{
  args: { param1: 'teste' },
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a6d32-43dc9d2250b7f2a54effeae8',
    accept: 'application/json, text/plain, */*',
    'user-agent': 'axios/0.21.4'
  },
  url: 'https://postman-echo.com/get?param1=teste'
}
```

Repare que a resposta nos entrega os parâmetros que enviamos na *query string* através da propriedade `args` , e os headers que enviamos através da propriedade `headers` . Repare também que não existe uma propriedade `body` , nem mesmo como um objeto vazio, pois requests do tipo `GET` não possuem `body` .

Agora, vamos adicionar alguns headers próprios. Isso será útil quando você precisar, por exemplo, enviar um token de autenticação, que é exatamente a situação que vamos simular. Altere o código do `index.js` .

Copiar

```
// index.js

// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
const headers = { Authorization: API_TOKEN };

// Continuamos a fazer uma requisição do tipo `GET`, mas passando o token no header
axios.get('https://postman-echo.com/get?param1=teste', { headers })
//   .then((response) => {

//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data" quando usamos axios
//     return response.data;
```

```
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um
status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status
${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível
completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Agora, executando novamente o código, obtemos o seguinte resultado:

Copiar

```
{
  args: { param1: 'teste' },
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a6e45-5e5198396aaf81b13e7a1f04',
    accept: 'application/json, text/plain, */*',
    authorization: '2d635ea9b637ea0f27d58985cc161d64',
    'user-agent': 'axios/0.21.4'
  },
  url: 'https://postman-echo.com/get?param1=teste'
}
```

A resposta é muito parecida, a não ser pelo fato de que, agora, temos, na chave `headers` , a chave `authorization` que enviamos! Show de bola, nosso token está sendo enviado para a API! 🎊

Utilizando outros verbos HTTP

Agora que vimos como utilizar headers, vamos ver o que precisamos fazer para utilizar um verbo HTTP diferente.
Altere o arquivo `index.js` :

Copiar

```
// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
const headers = { Authorization: API_TOKEN };

// Agora, iremos fazer uma requisição do tipo `POST`
axios.post('https://postman-echo.com/post?param1=teste')
//   .then((response) => {

//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data" quando usamos axios
//     return response.data;
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status ${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Basta substituir, na URL, o endpoint da API que queremos chamar, e trocar o método `GET` pelo método `POST` .
Executando o código agora, temos o seguinte resultado:

Copiar

```
{
  args: { param1: 'teste' },
  data: '',
  files: {},
  form: {},
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a79ee-0b0915602108a393443446b6',
    'content-length': '0',
    accept: 'application/json, text/plain, */*',
    'content-type': 'application/x-www-form-urlencoded',
    'user-agent': 'axios/0.21.4'
  },
  json: null,
  url: 'https://postman-echo.com/post?param1=teste'
}
```

Note que, dessa vez, temos mais propriedades além de `args` : `data` , `files` , `form` e `json` também estão presentes. Estão todas vazias pois não enviamos nada do tipo para a API ainda. O que nos leva ao nosso próximo passo!

Enviando um Body

Você vai ver agora como enviar informações no corpo da requisição. Para isso, vamos criar um objeto chamado `body` com algumas propriedades, para assim, conseguirmos enviar alguma informação. Mãos ~à massa~ ao código! 💻

Copiar

```
// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
```

```
const headers = { Authorization: API_TOKEN };

const body = {
  name: 'Tryber',
  email: 'tryber@betrybe.com',
  password: 'Tr1b3r',
};

// Agora, iremos fazer uma requisição do tipo `POST` passando o body
axios.post('https://postman-echo.com/post?param1=teste', body, {
headers })
//   .then((response) => {

//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data"
quando usamos axios
//     return response.data;
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um
status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status
${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível
completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Agora, se executarmos o código, o resultado é o seguinte:

Copiar

```
{
  args: { param1: 'teste' },
  data: { name: 'Tryber', email: 'tryber@betrybe.com', password:
'Tr1b3r' },
```

```
  files: {},
  form: {},
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a7e60-34e10ba802dc0500153955b4',
    'content-length': '66',
    accept: 'application/json, text/plain, */*',
    'content-type': 'application/json',
    authorization: '2d635ea9b637ea0f27d58985cc161d64',
    'user-agent': 'axios/0.21.4'
  },
  json: { name: 'Tryber', email: 'tryber@betrybe.com', password:
'Tr1b3r' },
  url: 'https://postman-echo.com/post?param1=teste'
}
```

Dessa vez, a API do Postman nos envia de volta um objeto na propriedade `data` , e o mesmo objeto na propriedade `json` , o que quer dizer que o corpo da mensagem foi lido e interpretado com sucesso! 🥳
Para utilizar outros verbos HTTP, basta alterar para o método desejado, e pronto!

Para finalizar: Aqui utilizamos o `axios` e o Node.js para executar os exemplos no terminal. Podemos utilizar também o pacote `axios` na nossa aplicação front-end para realizar requesições. Assim, pode ser uma ótima alternativa para fazer requisições quando realizarmos integração entre front-end e back-end.

Consumindo APIs

Até agora, você já consumiu APIs algumas vezes em diversos projetos como Carrinho de Compras , Front-end Online Store , Jogo de Trivia e por aí vai. No entanto, todas as vezes que precisou integrar seu front-end com alguma API, o único método utilizado foi o `GET` . Pois bem, chegou a hora de utilizar os demais métodos que aparecem no REST!
Para continuarmos fazendo requisições para APIs externas, vamos utilizar a biblioteca `axios` , que implementa no Node.JS, as mesmas funcionalidades que foram vistas nos módulos de Fundamentos e

Front-end usando o `fetch` . Neste exemplo, vamos consumir a Postman Echo API , que é uma API mantida pelo *Postman* para fins de aprendizagem. Ela simplesmente retorna para nós tudo o que enviamos para ela.
Vamos começar criando nosso pacote Node.js e instalando o `axios` .
Faremos isso numa nova pasta chamada `hello-http-methods` .
Execute o seguinte comando no terminal

Copiar
```
mkdir hello-http-methods && cd hello-http-methods && npm init -y && npm i axios
```

Agora, crie o arquivo `index.js` , onde vamos escrever nosso código:

Copiar
```
// index.js

const axios = require('axios').default;

// Para aquecer, vamos começar com uma requisição do tipo `GET`
axios.get('https://postman-echo.com/get?param1=teste')
  .then((response) => {

    // Caso esteja tudo OK, retornamos os dados
    // As informações são retornas dentro da propriedade "data" quando usamos axios
    return response.data;
  })
  .then((data) => {
    // Por fim, escrevemos o body no console
    console.log(data);
  })
  .catch((errorOrResponse) => {
    // Em caso de falha simples (a request completou com um status diferente de 2xx)
    // simplesmente logamos o status no console
    if (errorOrResponse.status) {
      return console.error(`Request failed with status ${errorOrResponse.status}`);
    }

    // Caso tenha acontecido um erro de rede (não foi possível completar a request)
    // logamos o erro todo
    console.error(errorOrResponse);
```

```
  });
```

Executando o código acima, temos o seguinte resultado no terminal:

Copiar

```
{
  args: { param1: 'teste' },
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a6d32-43dc9d2250b7f2a54effeae8',
    accept: 'application/json, text/plain, */*',
    'user-agent': 'axios/0.21.4'
  },
  url: 'https://postman-echo.com/get?param1=teste'
}
```

Repare que a resposta nos entrega os parâmetros que enviamos na *query string* através da propriedade `args` , e os headers que enviamos através da propriedade `headers` . Repare também que não existe uma propriedade `body` , nem mesmo como um objeto vazio, pois requests do tipo `GET` não possuem `body` .

Agora, vamos adicionar alguns headers próprios. Isso será útil quando você precisar, por exemplo, enviar um token de autenticação, que é exatamente a situação que vamos simular. Altere o código do `index.js` .

Copiar

```
// index.js

// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
const headers = { Authorization: API_TOKEN };

// Continuamos a fazer uma requisição do tipo `GET`, mas passando o token no header
axios.get('https://postman-echo.com/get?param1=teste', { headers })
//   .then((response) => {
```

```
//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data"
quando usamos axios
//     return response.data;
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um
status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status
${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível
completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Agora, executando novamente o código, obtemos o seguinte resultado:

Copiar

```
{
  args: { param1: 'teste' },
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a6e45-5e5198396aaf81b13e7a1f04',
    accept: 'application/json, text/plain, */*',
    authorization: '2d635ea9b637ea0f27d58985cc161d64',
    'user-agent': 'axios/0.21.4'
  },
  url: 'https://postman-echo.com/get?param1=teste'
}
```

A resposta é muito parecida, a não ser pelo fato de que, agora, temos, na chave `headers` , a chave `authorization` que enviamos! Show de bola, nosso token está sendo enviado para a API! 🎊

Utilizando outros verbos HTTP

Agora que vimos como utilizar headers, vamos ver o que precisamos fazer para utilizar um verbo HTTP diferente.
Altere o arquivo `index.js` :

Copiar

```
// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
const headers = { Authorization: API_TOKEN };

// Agora, iremos fazer uma requisição do tipo `POST`
axios.post('https://postman-echo.com/post?param1=teste')
//   .then((response) => {

//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data" quando usamos axios
//     return response.data;
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status ${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Basta substituir, na URL, o endpoint da API que queremos chamar, e trocar o método `GET` pelo método `POST` .
Executando o código agora, temos o seguinte resultado:

Copiar

```
{
  args: { param1: 'teste' },
  data: '',
  files: {},
  form: {},
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a79ee-0b0915602108a393443446b6',
    'content-length': '0',
    accept: 'application/json, text/plain, */*',
    'content-type': 'application/x-www-form-urlencoded',
    'user-agent': 'axios/0.21.4'
  },
  json: null,
  url: 'https://postman-echo.com/post?param1=teste'
}
```

Note que, dessa vez, temos mais propriedades além de `args` : `data` , `files` , `form` e `json` também estão presentes. Estão todas vazias pois não enviamos nada do tipo para a API ainda. O que nos leva ao nosso próximo passo!

Enviando um Body

Você vai ver agora como enviar informações no corpo da requisição. Para isso, vamos criar um objeto chamado `body` com algumas propriedades, para assim, conseguirmos enviar alguma informação. Mãos ~à massa~ ao código! 💻

Copiar

```
// const axios = require('axios').default;

// Armazenamos o token numa variável.
// Num ambiente real, esse valor viria do Local Storage, ou de uma variável de ambiente
const API_TOKEN = '2d635ea9b637ea0f27d58985cc161d64';

// Criamos um novo objeto de Headers
```

```
const headers = { Authorization: API_TOKEN };

const body = {
  name: 'Tryber',
  email: 'tryber@betrybe.com',
  password: 'Tr1b3r',
};

// Agora, iremos fazer uma requisição do tipo `POST` passando o body
axios.post('https://postman-echo.com/post?param1=teste', body, {
headers })
//   .then((response) => {

//     // Caso esteja tudo OK, retornamos os dados
//     // As informações são retornas dentro da propriedade "data"
quando usamos axios
//     return response.data;
//   })
//   .then((data) => {
//     // Por fim, escrevemos o body no console
//     console.log(data);
//   })
//   .catch((errorOrResponse) => {
//     // Em caso de falha simples (a request completou com um
status diferente de 2xx)
//     // simplesmente logamos o status no console
//     if (errorOrResponse.status) {
//       return console.error(`Request failed with status
${errorOrResponse.status}`);
//     }

//     // Caso tenha acontecido um erro de rede (não foi possível
completar a request)
//     // logamos o erro todo
//     console.error(errorOrResponse);
//   });
```

Agora, se executarmos o código, o resultado é o seguinte:

Copiar

```
{
  args: { param1: 'teste' },
  data: { name: 'Tryber', email: 'tryber@betrybe.com', password:
'Tr1b3r' },
```

```
  files: {},
  form: {},
  headers: {
    'x-forwarded-proto': 'https',
    'x-forwarded-port': '443',
    host: 'postman-echo.com',
    'x-amzn-trace-id': 'Root=1-613a7e60-34e10ba802dc0500153955b4',
    'content-length': '66',
    accept: 'application/json, text/plain, */*',
    'content-type': 'application/json',
    authorization: '2d635ea9b637ea0f27d58985cc161d64',
    'user-agent': 'axios/0.21.4'
  },
  json: { name: 'Tryber', email: 'tryber@betrybe.com', password:
'Tr1b3r' },
  url: 'https://postman-echo.com/post?param1=teste'
}
```

Dessa vez, a API do Postman nos envia de volta um objeto na propriedade `data` , e o mesmo objeto na propriedade `json` , o que quer dizer que o corpo da mensagem foi lido e interpretado com sucesso! 🥳
Para utilizar outros verbos HTTP, basta alterar para o método desejado, e pronto!

Para finalizar: Aqui utilizamos o `axios` e o Node.js para executar os exemplos no terminal. Podemos utilizar também o pacote `axios` na nossa aplicação front-end para realizar requesições. Assim, pode ser uma ótima alternativa para fazer requisições quando realizarmos integração entre front-end e back-end.

Instruções para realização dos exercícios

Esse exercício vai ser um pouco diferente. Vamos trabalhar com uma técnica de desenvolvimento de software chamada de refactoring .
Essa técnica consiste em alterar um pedaço de código que já existe e deixá-lo melhor. É exatamente isso que vamos fazer!
Vamos refatorar uma API que faz gestão de produtos para deixá-la mais elegante e respeitando os padrões impostos pelo REST. Quem sabe até deixá-la RESTful ?

Detalhes do projeto

1. Primeiro, crie um novo diretório para nosso projeto;
2. Crie um novo projeto:

Copiar
```
npm init -y
```

3. Instale o pacote `express` :

Copiar
```
npm install express
```

4. Instale o pacote `body-parser` para parsear o corpo das requisições:

Copiar
```
npm install body-parser
```

5. Instale o pacote `nodemon` para criar um servidor que se atualize em cada alteração salva nos arquivos:

Copiar
```
npm install nodemon
```

Adicione a linha abaixo no seu package.json, no objeto `"scripts"` para rodar o nodemon com o comando `npm run debug` :

Copiar
```
"debug": "nodemon index.js"
```

6. Por fim, instale o pacote `mysql2` para conectar com o banco:

Copiar
```
npm install mysql2
```

Se preferir instalar todas as dependências de uma vez, use o comando abaixo:

Copiar
```
npm install mysql2 nodemon body-parser express
```

7. Na raiz do nosso projeto, crie o arquivo `index.js` para configurarmos o express:

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

const app = express();
app.use(express.json());
```

```
app.use(bodyParser.urlencoded({ extended: false }));

app.use('/products', require('./controllers/productController'));

app.listen(3000, () => {
  console.log("App listening on port 3000!");
});
```

8. Vamos criar uma conexão com o `mysql` , crie uma pasta `models` e um arquivo `connection.js` :

Copiar

```
const mysql = require('mysql2/promise');

const connection = mysql.createPool({
  host: 'localhost', // Se necessário, substitua pelo seu host, `localhost` é o comum
  user: 'root', // Se necessário, substitua pelo seu usuário para conectar ao banco na sua máquina
  password: 'senha123', // Se necessário, substitua pela sua senha para conectar ao banco na sua máquina
  database: 'rest_exercicios'});

module.exports = connection;
```

Ainda na pasta `models` , dentro dela, crie o arquivo `productModel.js` . Dentro desse arquivo, vamos ter um CRUD completo utilizando uma conexão com o MySQL:

Copiar

```
const connection = require('./connection');

const add = async (name, brand) => {
  try {
    const [
      result,
    ] = await connection.query(
      `INSERT INTO products (name, brand) VALUES (?, ?);`,
      [name, brand]
    );

    return { id: result.insertId, name, brand };
  } catch (err) {
```

```
    console.error(err);
    return process.exit(1);
  }
};

const getAll = async () => {
  try {
    const [rows] = await connection.query('SELECT * FROM products');
    return rows;
  } catch (err) {
    console.error(err);
    return process.exit(1);
  }
};

const getById = async (id) => {
  try {
    const [result] = await connection.query('SELECT * FROM products
WHERE id = ?', [id]);
    if (!result.length) return null
    return result[0];
  } catch (err) {
    console.error(err);
    return process.exit(1);
  }
};

const update = async (id, name, brand) => {
  try {
    await connection.query('UPDATE products SET name = ?, brand = ?
WHERE id = ?', [name, brand, id])
  } catch (err) {
    console.error(err);
    return process.exit(1);
  }
};

const exclude = async (id) => {
  try {
    const product = await getById(id);
    if (!product) return {};
    await connection.query('DELETE FROM products WHERE id = ?',
[id])
```

```
    return product;
  } catch (err) {
    console.error(err);
    return process.exit(1);
  }
};

module.exports = { add, getAll, getById, update, exclude };
```

7. Execute esse script para subir o banco do exercício:

Copiar

```
DROP DATABASE IF EXISTS rest_exercicios;
CREATE DATABASE IF NOT EXISTS rest_exercicios;
USE rest_exercicios;

CREATE TABLE IF NOT EXISTS products (
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    brand VARCHAR(100) NOT NULL
);

INSERT INTO products (name, brand)
VALUES ('Cerveja Skol', 'Ambev'),
       ('Monitor AGON', 'AOC'),
       ('MacBook Air', 'Apple');
SELECT * FROM products;
```

8. Por último, vamos criar uma pasta `controllers` e, dentro dela, o arquivo `productController.js` . Esse será o local onde utilizaremos a técnica de refactoring:

Copiar

```
const express = require('express');
const ProductModel = require('../models/productModel');

const router = express.Router();

router.get('/list-products', async (req, res, next) => {
  const products = await ProductModel.getAll();

  res.send(products);
});
```

```
router.get('/get-by-id/:id', async (req, res, next) => {
  const product = await ProductModel.getById(req.params.id);

  res.send(product);
});

router.post('/add-user', async (req, res) => {
  const { name, brand } = req.body;

  const newProduct = await ProductModel.add(name, brand);

  res.send(newProduct);
});

router.post('/delete-user/:id', async (req, res) => {
  const products = await ProductModel.exclude(req.params.id);

  res.send(products);
});

router.post('/update-user/:id', async (req, res) => {
  const { name, brand } = req.body;

  const products = await ProductModel.update(req.params.id, name,
brand);

  res.send(products);
});

module.exports = router;
```

Agora é sua vez!

Exercício 1 : Pense qual é o recurso que estamos trabalhando e altere os endpoints para que fiquem padronizados.
Exercício 2 : Padronize todos os retornos para JSON.
Exercício 3 : Utilize os verbos (POST, PUT, GET etc.) corretos para cada ação do CRUD.

Exercício 4 : Garanta que todos os endpoints tenham as respostas (status code) corretas, ou seja, código para quando der tudo certo, código pra quando ocorrer erro etc.
Dica : Para testar suas requisições você pode utilizar o Postman .

Bônus

1. Refatore a API para que utilize MongoDB como banco de dados.

Recursos Adicionais

- What is REST
- What are Resources
- Software Architecture Guide - Martin Fowler
- Arquitetura Multicamadas
- Lista de MIME Types no MDN
- Documentação do Content-Type no MDN

Gabarito dos exercícios

A seguir encontra-se uma sugestão de solução para o exercício proposto.

Solução

O código a seguir é a solução dos exercícios 1 ao 4. No qual criarmos corretamente o recurso, fizemos uso dos verbos HTTP, padronizamos os retornos para JSON e informamos os referentes status code de cada request.
Exercício 1 : Pense qual é o recurso que estamos trabalhando e altere os endpoints para que fiquem padronizados.
Exercício 2 : Padronize todos os retornos para JSON.
Exercício 3 : Utilize os verbos (POST, PUT, GET etc.) corretos para cada ação do CRUD.

Exercício 4 : Garanta que todos os endpoints tenham as respostas (status code) corretas, ou seja, código para quando der tudo certo, código pra quando ocorrer erro etc.
controllers/productController.js

Copiar

```
const express = require('express');
const ProductModel = require("../models/productModel");
const router = express.Router();

router.get('/', async (req, res, next) => {
  const products = await ProductModel.getAll();

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

router.get('/:id', async (req, res, next) => {
  const product = await ProductModel.getById(req.params.id);

  if (product === null) {
    res.status(404).send({ message: 'Produto não encontrado' });
  }

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

router.post('/', async (req, res) => {
  const { name, brand } = req.body;

  try {
    const newProduct = await ProductModel.add(name, brand);

    res.status(200).json(newProduct);
  } catch (e) {
    res.status(500).send({ message: 'Algo deu errado' });
  }
});

router.delete('/:id', async (req, res) => {
  try {
    const products = await ProductModel.exclude(req.params.id);

    res.status(200).json(products);
  } catch (e) {
```

```
    res.status(500).send({ message: 'Algo deu errado' });
  }
});

router.put('/:id', async (req, res) => {
  const { name, brand } = req.body;

  try {
    const products = await ProductModel.update(req.params.id, name,
brand);

    res.status(200).json(products);
  } catch (e) {
    res.status(500).send({ message: 'Algo deu errado' });
  }
});

module.exports = router;
```

Bônus

1. Refatore a API para que utilize MongoDB como banco de dados.

Para criar o banco de dados execute esse script.
OBS: Não se esqueça de instalar o mongodb na sua aplicação com o comando `npm install mongodb`

Copiar

```
use rest_exercicios
db.products.insertMany([
    { "name": "Cerveja Skol", "brand": "Ambev" },
    { "name": "Monitor AGON", "brand": "AOC" },
    { "name": "MacBook Air", "brand": "Apple" }
])
```

models/connections.js

Copiar

```
const { MongoClient } = require('mongodb');

const OPTIONS = {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}
```

```
const MONGO_DB_URL = 'mongodb://127.0.0.1:27017';

let db = null;

const connection = () => {
    return db
    ? Promise.resolve(db)
    : MongoClient.connect(MONGO_DB_URL, OPTIONS)
    .then((conn) => {
    db = conn.db('model_example');
    return db;
    })
};

module.exports = connection;
```

models/productModel.js

Copiar

```
const mongodb = require('mongodb');
const connection = require('./connection');
const { ObjectId } = require('mongodb');

async function getAll() {
    const db = await connection();
    const products = await
db.collection('products').find().toArray();
    return products;
  }

async function getById(id) {
  const db = await connection();
  if(!ObjectId.isValid(id)) return null;
  return db.collection('products').findOne(ObjectId(id));
}

async function add(name, brand) {
  const db = await connection();
  const addProduct = await
db.collection('products').insertOne({name, brand});
  return addProduct;
}

async function exclude (id) {
```

```
  const db = await connection();
  if(!ObjectId.isValid(id)) return null;
  const product = await getBytId(id);
  await db.collection('products').deleteOne({ _id: ObjectID(id) });
  return product;
}

async function update(id) {
  const db = await connection();
  if(!ObjectId.isValid(id)) return null;
  const product = await db.collection('products').updateOne({ _id:
ObjectID(id) }, { $set: name, brand });
  if(!product) return add(name, brand);
  return product;
}


module.exports = { add, getAll, getById, update, exclude };
```