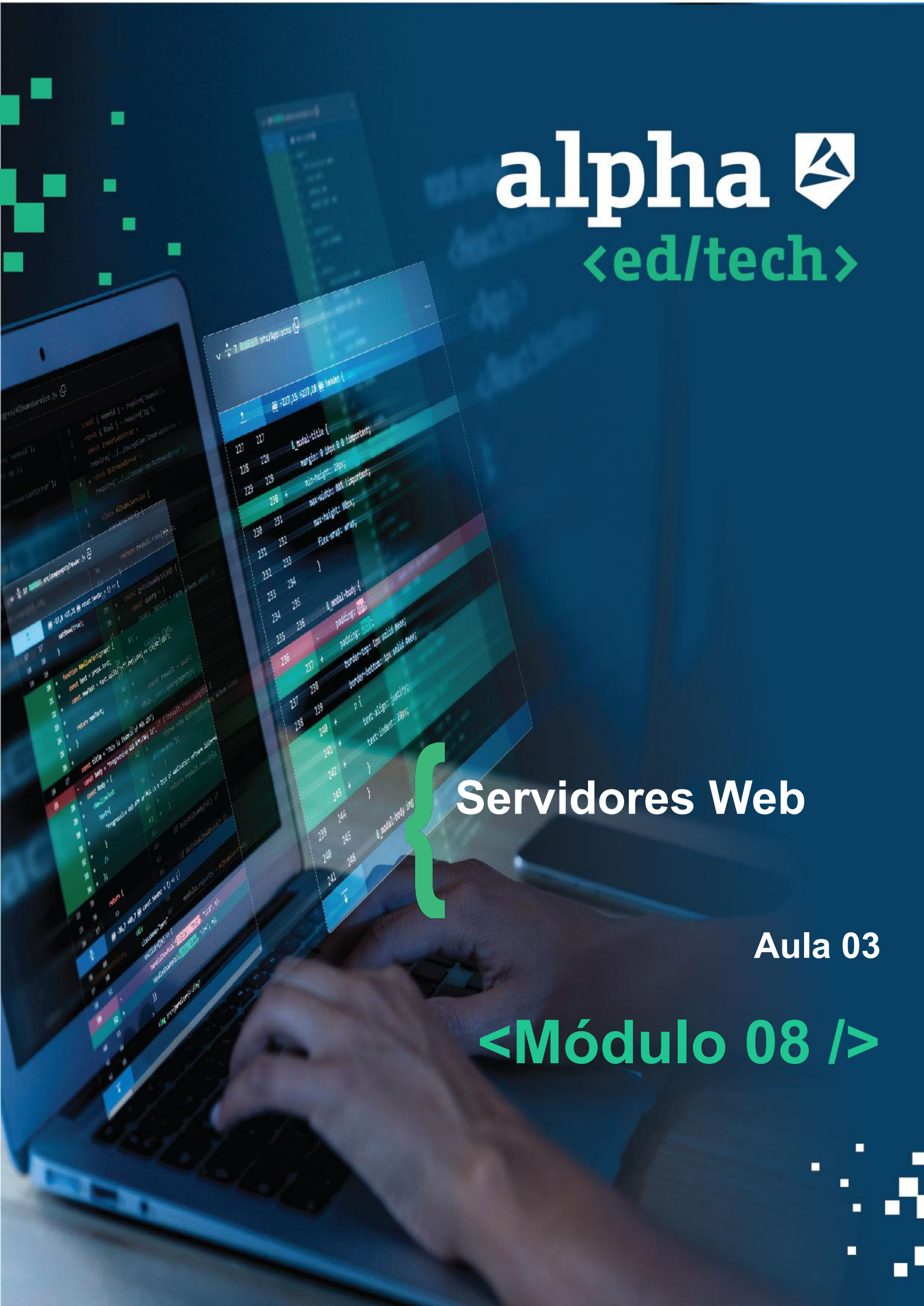

alpha
<ed/tech>
Servidores Web
Aula 03
<Módulo 08 />

Interagindo com APIs em Nodejs

Capturando parâmetros de URL com GET

Os parâmetros de URL são valores que são inseridos diretamente na URL, geralmente após um certo caminho.
Eles são frequentemente usados para identificar recursos específicos.
Por exemplo, em uma URL como **http://example.com/users/123**, 123 seria um parâmetro de URL que provavelmente identifica um usuário específico.

req.params

No Express.js, você pode capturar parâmetros de URL usando o objeto **req.params**.
Para fazer isso, você precisa definir um espaço reservado para o parâmetro em sua rota.
Os espaços reservados são definidos com dois pontos seguidos pelo nome do parâmetro, como **/users/:userId**

Aqui está um exemplo de código em Node.js:

```
app.get('/users/:userId', function(req, res) {
    let userId = req.params.userId;
    res.send(`User ID: ${userId}`);
});
```

Neste exemplo, se você acessar http://localhost:3000/users/123, o servidor responderá com User ID: 123.

Enviando dados GET

Para enviar dados usando req.params com a API Fetch, você precisa incluir os parâmetros na URL.
Aqui está um exemplo:

```
let userId = '123';
let url = `http://localhost:3000/users/${userId}`;

fetch(url, {
  method: 'GET'
})
.then(response => response.text())
.then(result => console.log(result))
.catch(error => console.error('Error:', error));
```

Interagindo com APIs em Nodejs

Capturando corpo de solicitação com POST e PUT

O corpo da solicitação é a parte da solicitação HTTP onde os dados adicionais são enviados ao servidor. Em uma solicitação POST ou PUT, o corpo geralmente contém os dados que você deseja enviar ao servidor. Esses dados podem estar em vários formatos, incluindo JSON, XML, texto simples, etc.

req.body

No Express.js, você pode acessar o corpo da solicitação usando o objeto **req.body**.
No entanto, o Express não analisa o corpo da solicitação por padrão.
Você precisa usar um middleware, como o **express.json()** ou **express.urlencoded()**, para analisar o corpo da solicitação.
Com uso do **express.json()**:

```
const express = require('express');
const app = express();

// Middleware para analisar o corpo da solicitação JSON
app.use(express.json());

app.post('/users', function(req, res) {
  let userName = req.body.name;
  let userEmail = req.body.email;
  res.send(`Name: ${userName}, Email: ${userEmail}`);
});

app.listen(3000, function() {
  console.log('App is listening on port 3000');
});
```

Com o uso do **express.urlencoded()**:

```
const express = require('express');
const app = express();

// Middleware para analisar o corpo da solicitação x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

app.post('/users', function(req, res) {
  let userName = req.body.name;
  let userEmail = req.body.email;
  res.send(`Name: ${userName}, Email: ${userEmail}`);
});

app.listen(3000, function() {
  console.log('App is listening on port 3000');
});
```

Interagindo com APIs em Nodejs

Enviando dados no body no formato JSON

Para enviar uma solicitação JSON usando a API Fetch, você precisa definir o cabeçalho Content-Type como **application/json** e formatar o corpo da solicitação como uma string JSON.

```
let url = 'http://localhost:3000/users';
let data = {
  name: 'John Doe',
  email: 'john@example.com'
};

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(response => response.text())
.then(result => console.log(result))
.catch(error => console.error('Error:', error));
```

Enviando dados no body no formato urlencoded

Para enviar uma solicitação x-www-form-urlencoded usando a API Fetch, você precisa definir o cabeçalho Content-Type como **application/x-www-form-urlencoded** e formatar o corpo da solicitação como uma string de consulta. Aqui está um exemplo:

```
let url = 'http://localhost:3000/users';
let data = {
  name: 'John Doe',
  email: 'john@example.com'
};

// Transforma o objeto data em uma string de consulta
let body = Object.keys(data).map(key => encodeURIComponent(key) + '=' + encodeURIComponent(data[key])).join('&');

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: body
})
.then(response => response.text())
.then(result => console.log(result))
.catch(error => console.error('Error:', error));
```

Interagindo com APIs em Nodejs

Query Strings

As **Query Strings** são uma parte fundamental das URLs que permitem passar informações adicionais para o servidor.
Elas começam com um ponto de interrogação **(?)** e são seguidas por pares de chave-valor, com cada par separado por um sinal de igual **(=)** e cada par separado por um e comercial **(&).**
Por exemplo, na URL **http://meusite.com?chave1=valor1&chave2=valor2**,
chave1 e chave2 são parâmetros da Query String.

As Query Strings são úteis para passar dados que não se encaixam naturalmente na hierarquia de uma URL, como parâmetros de pesquisa, dados de paginação, etc.

req.query

No Express.js, você pode acessar os parâmetros da Query String usando o objeto **req.query**.
Este objeto contém uma propriedade para cada parâmetro da Query String.

Aqui está um exemplo de como você pode usar req.query em uma rota Express:

```
app.get('/minharota', function(req, res) {
    let chave1 = req.query.chave1;
    let chave2 = req.query.chave2;
    res.send(`Chave1 é: ${chave1}, Chave2 é: ${chave2}`);
});
```

Neste exemplo, se você acessar
http://localhost:3000/minharota?chave1=valor1&chave2=valor2
O servidor responderá com Chave1 é: valor1, Chave2 é: valor2.

Um exemplo de envio de query strings utilizando fetch:

```
let url = 'http://localhost:3000/minharota';
let params = {
  chave1: 'valor1',
  chave2: 'valor2'
};

// Transforma o objeto params em uma string de consulta
let queryString = Object.keys(params).map(key => key + '=' + encodeURIComponent(params[key])).join('&');

fetch(url + '?' + queryString, {
  method: 'GET'
})
.then(response => response.text())
.then(result => console.log(result))
.catch(error => console.error('Error:', error));
```

Query Strings

Elas são visíveis para o usuário e podem ser facilmente manipuladas, portanto, nunca devem ser usadas para passar informações sensíveis ou confidenciais.

Interagindo com APIs em Nodejs

Validação de Dados

A validação de dados é um passo crucial no desenvolvimento de qualquer aplicação.
Ela garante que os dados recebidos sejam do tipo e formato esperados antes de serem processados ou armazenados.

Isso é importante por várias razões:

- **Segurança**: Dados inválidos podem levar a vulnerabilidades de segurança, como ataques de injeção de SQL.
- **Integridade dos dados**: Garante que os dados armazenados em seu banco de dados sejam precisos e consistentes.
- **Experiência do usuário**: Fornece feedback útil ao usuário se eles inserirem dados incorretos.

Exemplo de validação

Você pode criar suas próprias funções de validação em JavaScript puro.

Aqui está um exemplo de como você pode validar um objeto representando um usuário:

```
function validarUsuario(usuario) {
    if (typeof usuario.nome !== 'string' || usuario.nome.length < 3 || usuario.nome.length > 30) {
      return 'Nome deve ser uma string entre 3 e 30 caracteres';
    }

    if (typeof usuario.email !== 'string' || !/^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/.test(usuario.email)) {
      return 'Email deve ser um endereço de email válido';
    }

    if (typeof usuario.senha !== 'string' || !/^[a-zA-Z0-9]{3,30}$/.test(usuario.senha)) {
      return 'Senha deve ser uma string alfanumérica entre 3 e 30 caracteres';
    }

    return null;
}

let usuario = {
    nome: 'Kenji Taniguchi',
    email: 'kenji@alphaedtech.org.br',
    senha: 'minhasenha'
};

let erro = validarUsuario(usuario);

if (erro) {
    console.log('Erro:', erro);
} else {
    console.log('Os dados do usuário são válidos');
}
```

Bibliotecas de Validação

Você pode usar bibliotecas de validação como **Joi** ou **express-validator**, ou escrever suas próprias funções de validação.

Interagindo com APIs em Nodejs

Melhores práticas ao lidar com parâmetros de solicitação

- **Validação de entrada**: Sempre valide os dados de entrada. Isso pode ajudar a prevenir muitos problemas, incluindo ataques de injeção de SQL e XSS.
- **Não confie nos dados do cliente**: Nunca confie nos dados enviados pelo cliente. Mesmo se você tiver validação no lado do cliente, sempre valide novamente no lado do servidor.
- **Use parâmetros de consulta para filtragem, classificação e paginação**: É uma prática comum usar parâmetros de consulta para tarefas como filtragem, classificação e paginação.
- **Evite expor informações sensíveis em URLs**: Informações sensíveis, como tokens de autenticação ou IDs de usuário, não devem ser incluídas na URL, pois podem ser expostas em logs de servidor ou históricos de navegador.
- **Use métodos HTTP apropriados**: Use GET para solicitações que não alteram o estado do servidor e POST, PUT, DELETE para solicitações que alteram o estado do servidor.
- **Limite o tamanho dos parâmetros de entrada**: Para evitar ataques de negação de serviço (DoS), limite o tamanho dos parâmetros de entrada.
- **Codifique os dados corretamente**: Ao enviar dados na URL, certifique-se de que eles estejam corretamente codificados para evitar problemas com caracteres especiais.
- **Use HTTPS**: Para proteger os dados do usuário durante a transmissão, use HTTPS em vez de HTTP.

Interagindo com APIs em Nodejs

Headers HTTP

Aqui estão alguns dos principais *headers* (cabeçalhos) HTTP que você deve conhecer ao trabalhar com APIs RESTful:

- **Content-Type**: Especifica o tipo de mídia do corpo da solicitação.
  Exemplos comuns incluem application/json, application/xml, e text/html.
- **Accept**: Informa ao servidor os tipos de mídia que o cliente pode entender.
- **Authorization**: Usado para autenticação, geralmente contém credenciais na forma de tokens Bearer ou Basic.
- **Cache-Control**: Direciona o cliente e todas as partes intermediárias sobre como armazenar em cache a resposta.
- **ETag**: Um identificador único para uma versão específica de um recurso. É usado para permitir o cache eficiente e a concorrência otimizada.
- **If-Modified-Since / If-None-Match**: Usado para fazer solicitações condicionais.
  Se o recurso não foi modificado desde a última solicitação, o servidor retornará um status 304 Not Modified.
- **Content-Encoding**: Usado para especificar o tipo de codificação de conteúdo usado para comprimir os dados da resposta.
- **Content-Length**: O tamanho, em bytes, do corpo da resposta.
- **Set-Cookie**: Envia cookies do servidor para o cliente. Muito útil para manter o estado do usuário entre solicitações.
- **X-Requested-With**: Usado para identificar solicitações Ajax. O valor mais comum é XMLHttpRequest.

Lembre-se de que o uso desses *headers* dependerá das necessidades específicas de sua API e do que você está tentando alcançar.

Interagindo com APIs em Nodejs

Header ‘Access-Control-Allow-Origin’

O header HTTP para lidar com o **CORS** (*Cross-Origin Resource Sharing*) é o **Access-Control-Allow-Origin**.

Este header é usado para indicar se os recursos de uma página podem ser carregados de um domínio diferente do da página em si.

Por exemplo, você pode usar este header para permitir que seu API seja acessada de um domínio diferente.

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

const app = express();

var opcoesCors = {
  origin: 'http://exemplo.com', // Substitua pelo seu domínio
  optionsSuccessStatus: 200, // Alguns navegadores antigos (IE11, várias SmartTVs) falham com 204
  methods: "GET,HEAD,PUT,PATCH,POST,DELETE",
  allowedHeaders: ['Content-Type', 'Authorization']
};

app.use(cors(opcoesCors));

app.get('/', function (req, res) {
  res.send('Olá Mundo!');
});

app.listen(3000, function () {
  console.log('App ouvindo na porta 3000!');
});
```

No código acima, estamos usando o pacote **cors** do npm para habilitar o CORS.
A função **cors()** é usada como um middleware para todas as rotas.

O objeto **opcoesCors** define as opções de CORS.

- **origin** especifica o domínio que tem permissão para acessar o recurso.
- **optionsSuccessStatus** define o status que é enviado para o navegador em caso de sucesso.
- **methods** define os métodos HTTP permitidos.
- **allowedHeaders** define os headers HTTP permitidos.

Se você quiser permitir vários domínios, você pode passar um array de strings.

Lembre-se de que você precisa instalar o pacote cors usando npm ou yarn antes de poder usá-lo.
Você pode fazer isso com o comando **npm install cors** ou **yarn add cors**.

Interagindo com APIs em Nodejs

Header ‘X-Frame-Options’

O header ***HTTP X-Frame-Options*** permite que uma página seja exibida em um **iframe**.

Este header pode ter um dos seguintes valores:

- **DENY**: A página não pode ser exibida em um frame, independentemente do site que tenta fazer isso.
- **SAMEORIGIN**: A página só pode ser exibida em um frame no mesmo site de origem que a própria página.
- **ALLOW-FROM uri**: A página pode ser exibida apenas em um frame no site especificado.

Se você quiser permitir que sua página seja exibida em um iframe em qualquer site, você pode remover completamente o header X-Frame-Options.
No entanto, isso pode levar a problemas de segurança, como ataques de **clickjacking**, então use com cuidado.

```
const express = require('express');
const app = express();

app.use(function(req, res, next) {
  res.header("X-Frame-Options", "ALLOW-FROM http://exemplo.com");
  next();
});

app.get('/', function (req, res) {
  res.send('Olá Mundo!');
});

app.listen(3000, function () {
  console.log('App ouvindo na porta 3000!');
});
```

O header **Content-Security-Policy** com a diretiva **frame-ancestors** é uma alternativa mais moderna e flexível ao **X-Frame-Options**.

Por exemplo, **Content-Security-Policy: frame-ancestors 'self' http://example.com**; permitiria que a página fosse exibida em um iframe no mesmo site e em http://example.com.

```
const express = require('express');
const app = express();

app.use(function(req, res, next) {
  res.header("Content-Security-Policy", "frame-ancestors 'self' http://exemplo.com");
  next();
});

app.get('/', function (req, res) {
  res.send('Olá Mundo!');
});

app.listen(3000, function () {
  console.log('App ouvindo na porta 3000!');
});
```

Interagindo com APIs em Nodejs

Códigos de status HTTP

Os códigos de status HTTP são uma parte importante da comunicação entre o servidor e o cliente. Eles informam ao cliente o resultado de sua solicitação. Aqui estão alguns dos códigos de status HTTP mais comuns que as APIs devem retornar:

- **200 OK**: A solicitação foi bem-sucedida. Este é geralmente o código de resposta para uma solicitação GET bem-sucedida.
- **201 Created**: A solicitação foi bem-sucedida e um novo recurso foi criado como resultado. Este é geralmente o código de resposta para uma solicitação POST que resulta na criação de um novo recurso.
- **204 No Content**: A solicitação foi bem-sucedida, mas não há representação para retornar (ou seja, o corpo da resposta está vazio).
- **400 Bad Request**: A solicitação não pôde ser entendida ou estava faltando parâmetros necessários.
- **401 Unauthorized**: A autenticação falhou ou ainda não foi fornecida.
- **403 Forbidden**: A autenticação foi bem-sucedida, mas o cliente autenticado não tem acesso ao recurso solicitado.
- **404 Not Found**: O recurso solicitado não pôde ser encontrado.
- **405 Method Not Allowed**: O método HTTP usado na solicitação não é permitido para o recurso específico.
- **500 Internal Server Error**: Um erro ocorreu no servidor.

Esses códigos de status ajudam o cliente a entender se a solicitação foi bem-sucedida ou não, e em caso de falha, o que deu errado. Eles são uma parte essencial de qualquer API RESTful e ajudam a manter a consistência e a compreensibilidade da API.

Códigos de status HTTP no express

Os códigos de status HTTP podem ser enviados com a função status() conforme exemplo a seguir:

```
const express = require('express');
const app = express();

app.get('/ok', function (req, res) {
  res.status(200).send('OK');
});

app.get('/created', function (req, res) {
  res.status(201).send('Criado');
});

app.get('/badrequest', function (req, res) {
  res.status(400).send('Pedido inválido');
});

app.get('/unauthorized', function (req, res) {
  res.status(401).send('Não autorizado');
});

app.listen(3000, function () {
  console.log('App ouvindo na porta 3000!');
});
```

Testando APIs

Introdução

Existem várias ferramentas disponíveis para testar APIs RESTful.
Neste módulo vamos aprender a utilizar o Postman que é uma das ferramentas mais populares para testar APIs. O Postman oferece uma interface de usuário amigável, suporte para vários tipos de solicitações HTTP, testes automatizados, documentação e colaboração.

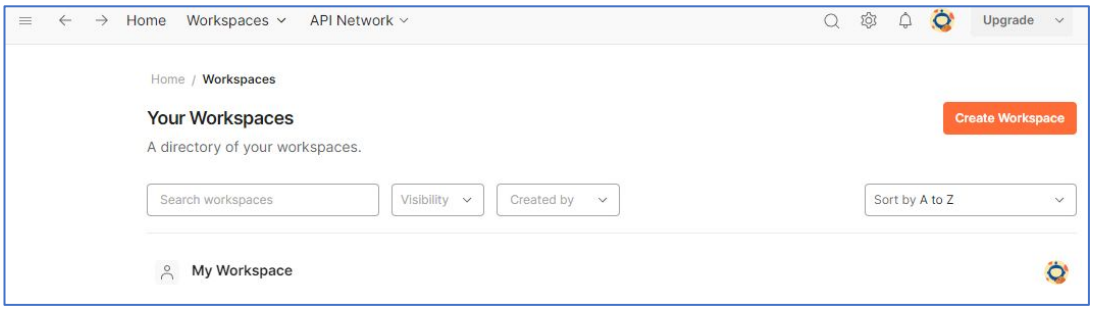
Postman

Acesse o site oficial do Postman
https://www.postman.com/downloads/

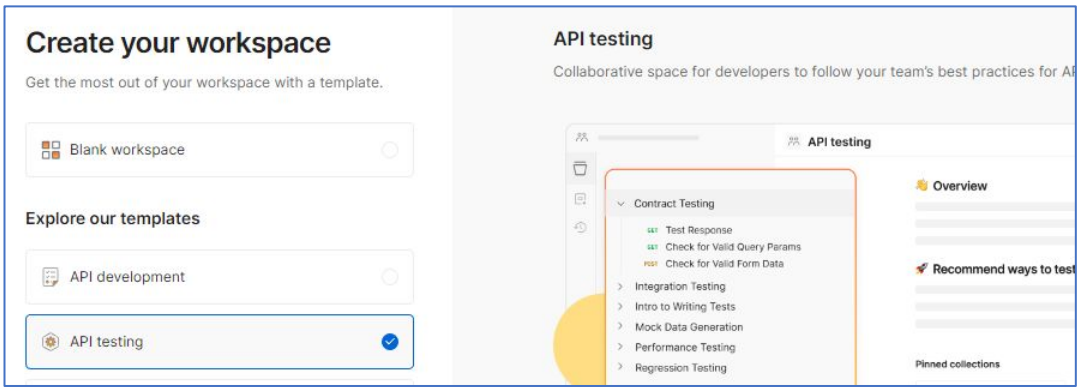
Clique no botão "Download" para baixar o instalador do Postman para o seu sistema operacional e siga as instruções na tela para instalar o Postman.
A versão utilizada é de Janeiro 2024 (v10.22)

- **Criando uma área de trabalho (workspace)**:

Incialmente, crie uma área de trabalho, selecionando "workspaces" e depois clique em "create workspace"



Como nosso intuito é testar API, selecione "API testing" e depois "Next"



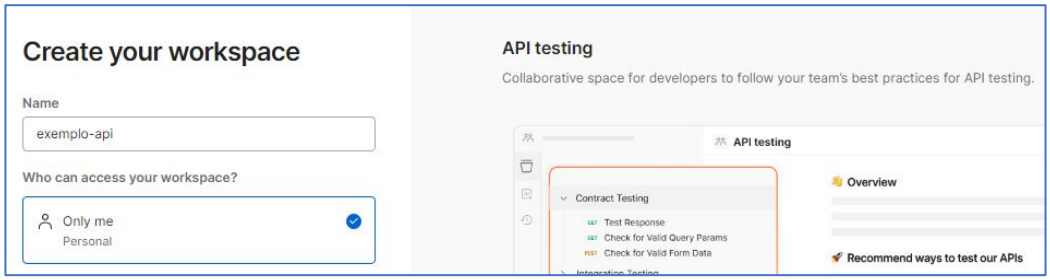
Outras ferramentas

- **Insomnia**: Uma alternativa ao Postman que também oferece uma interface de usuário amigável e suporte para vários tipos de solicitações HTTP. O Insomnia é conhecido por sua interface de usuário limpa e desempenho rápido.

- **Curl**: Uma ferramenta de linha de comando para enviar solicitações HTTP. O Curl é muito flexível e poderoso, mas tem uma curva de aprendizado mais íngreme do que as ferramentas com interface de usuário.

Testando APIs

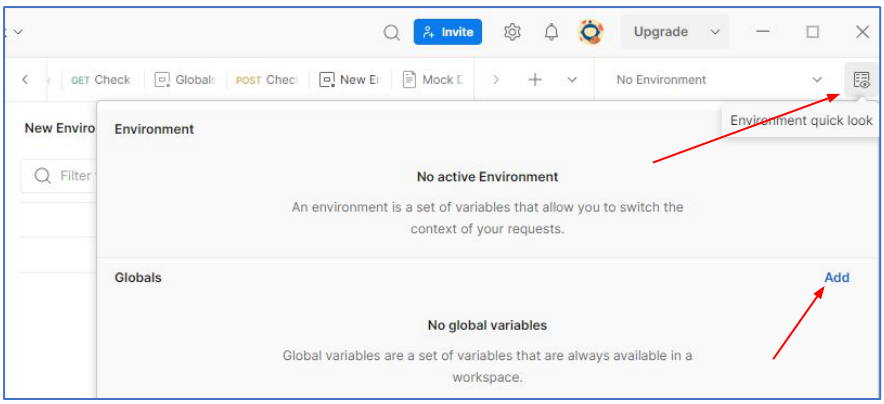
Postman



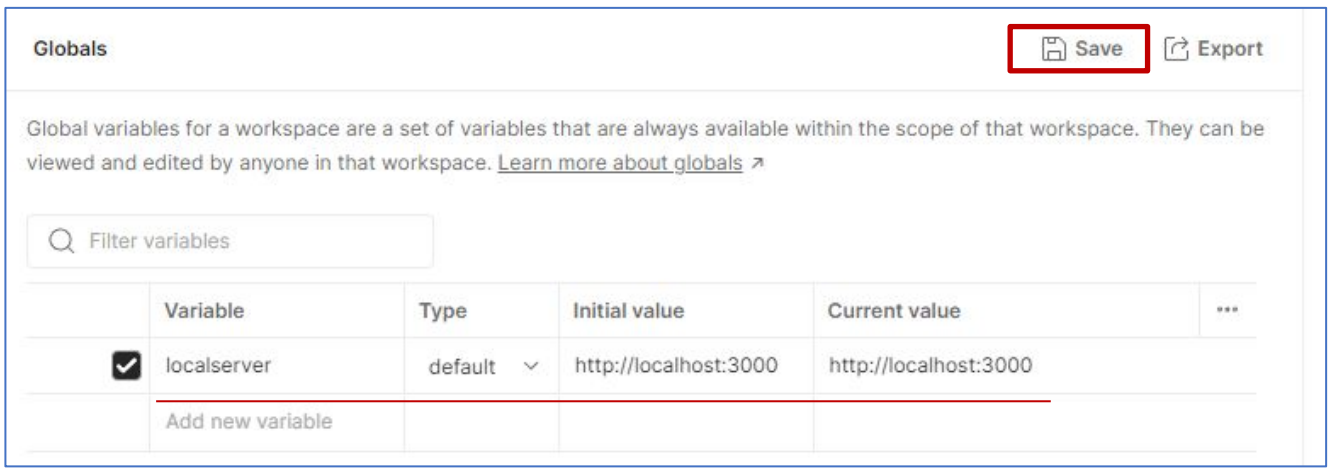
Dê um nome para sua área de trabalho (neste exemplo o nome dado foi "exemplo-api", selecione quem irá utilizar este workspace (neste exemplo, apenas você), e depois clique em "Create".
Pronto, já podemos começar a registrar nossas requisições.

- **Criando variáveis globais**:

Para facilitar a criação das requisições e o uso de valores que se repetem sem precisar alterar toda vez que há alguma mudança, é recomendável o registro de variáveis globais.
Para isso, basta clicar no canto superior direito no ícone com um 'olho' chamado 'Environment quick look' e no 'Globals' clique em 'Add'

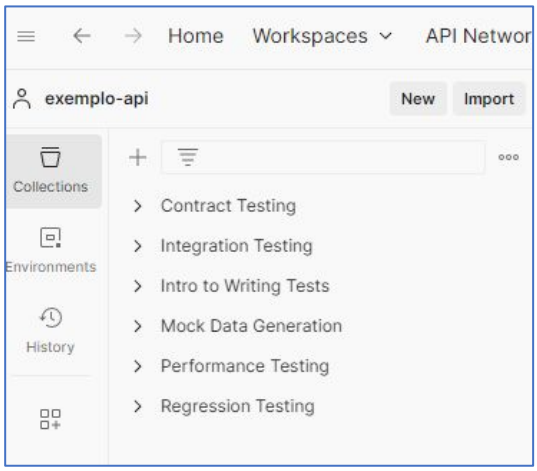


No nosso caso, para testar nossa aplicação loca, criamos a variável 'localserver' com o valor de 'http://localhost:3000' e depois clicando em 'Save'



Testando APIs

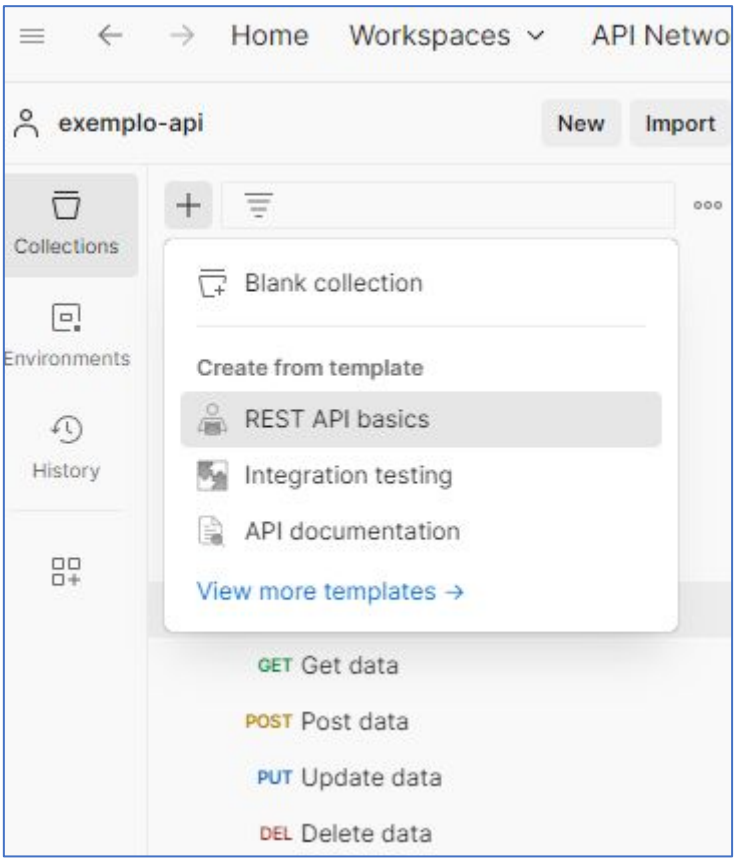
Postman



- **Collections**:

Recomenda-se testar as 'collections' pré-criadas pelo Postman para entender um pouco mais sobre as funcionalidades como 'Contract Testint', 'Integration Testint', 'Intro to Writing Tests', 'Mock Data Generation', 'Performance Testing' e 'Regression Testing

Para criar uma nova coleção, basta clicar no ícone de '+', identificado como 'Create New Collection', e depois clique em 'REST API Basics'

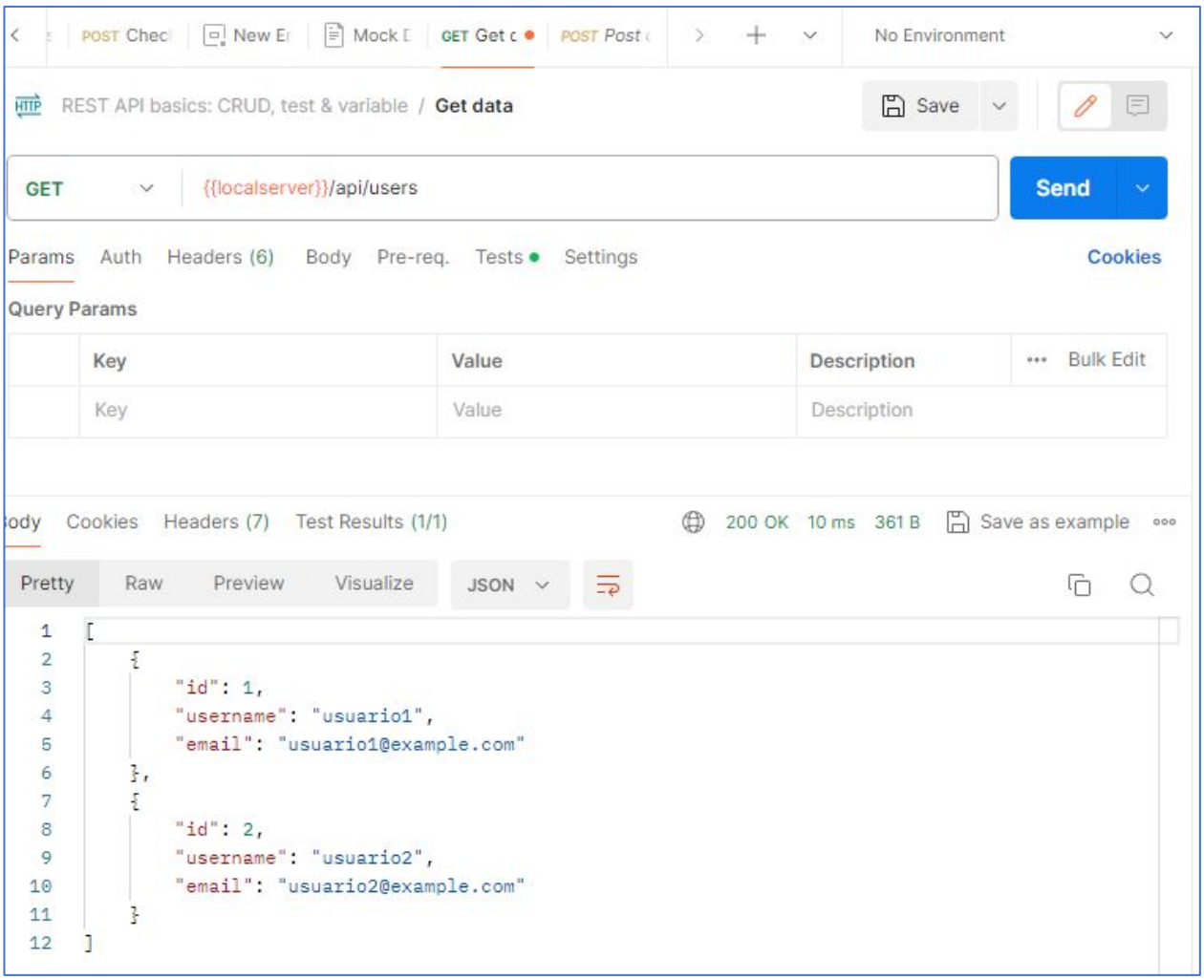


Note que já foram criadas 4 requisições que representam as requisições básicas de 'GET', 'POST', 'UPDATE' e 'DELETE'.

Testando APIs

Postman

- **GET**: Para criar uma nova requisição do tipo GET, substituir o endereço da variável 'base_url' para 'localserver' como anteriormente cadastramos e clicar em 'Send' para analisar a resposta.

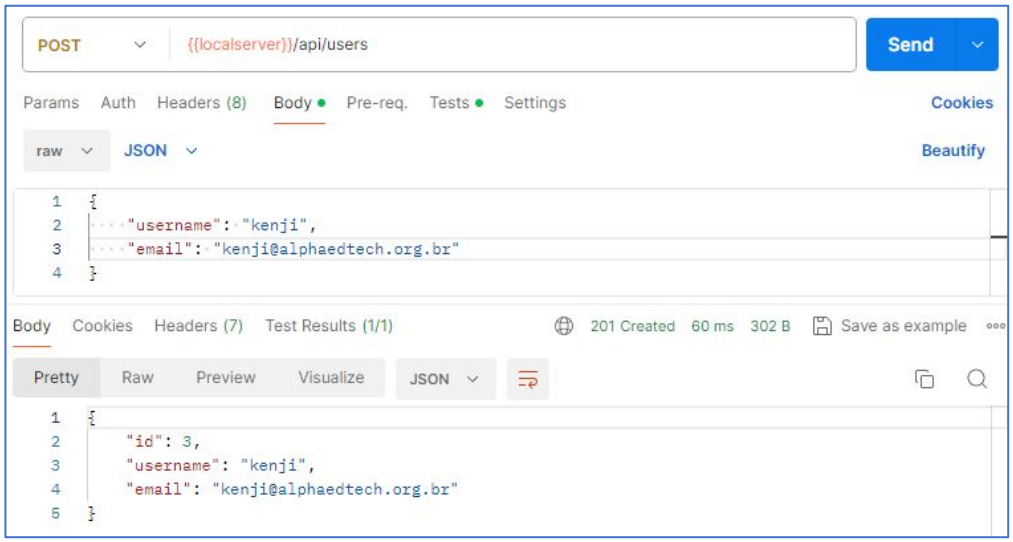


Veja também que há um script de test na aba 'Tests' que contém uma verificação se o status code retorna o valor '200':

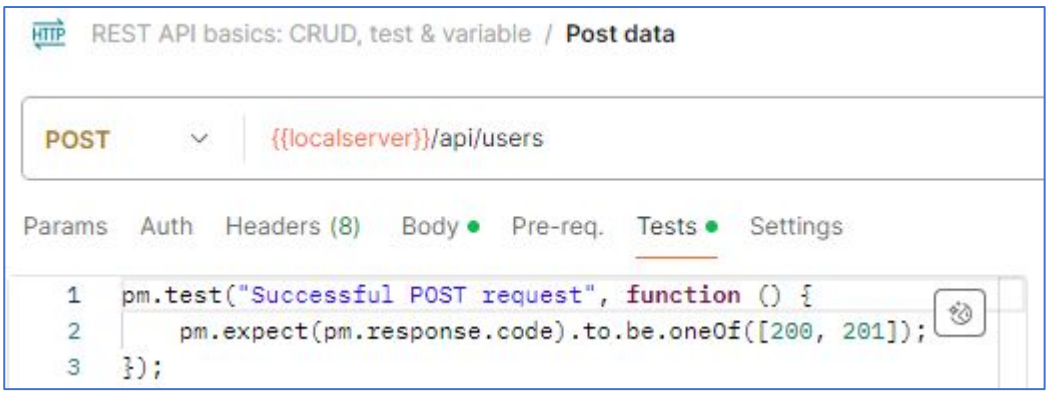
Testando APIs

Postman

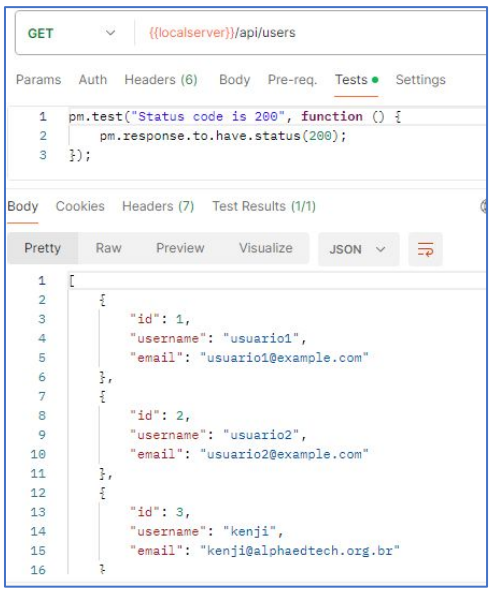
- **POST**: Para criar uma nova requisição do tipo POST, substituir o endereço da variável 'base_url' para 'localserver' como anteriormente cadastramos, colocar o dado a ser enviado na aba 'Body' selecionando 'raw' 'JSON' e clicar em 'Send' para analisar a resposta.



Veja também que há um script de test na aba 'Tests' que contém uma verificação se o status code retorna o valor '200':



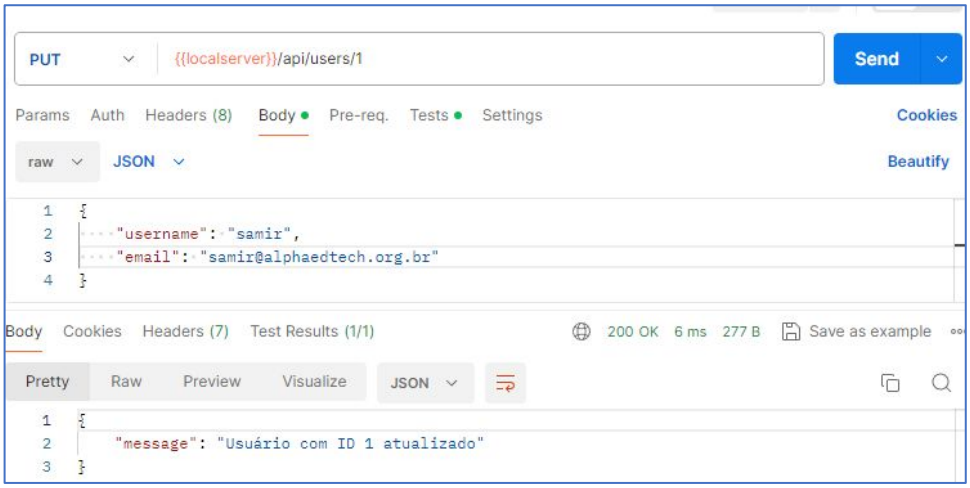
Em nova consulta GET notamos que o usuário enviado pela requisição POST foi incluído com sucesso:



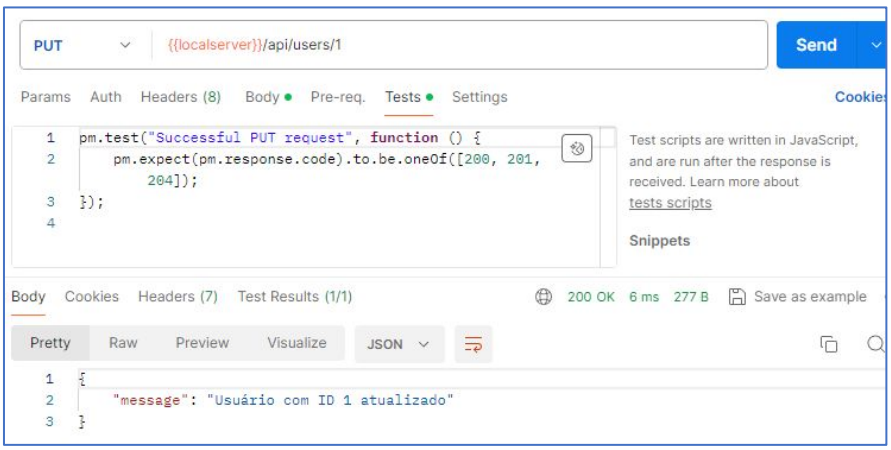
Testando APIs

Postman

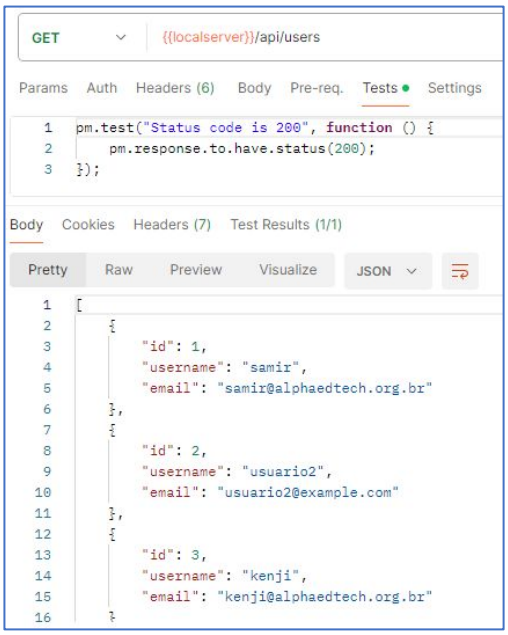
- **PUT**: Para criar uma nova requisição do tipo UPDATE, substituir o endereço da variável 'base_url' para 'localserver' como anteriormente cadastramos, colocar o dado a ser enviado na aba 'Body' selecionando 'raw' 'JSON' e clicar em 'Send' para analisar a resposta.



Veja também que há um script de test na aba 'Tests' que contém uma verificação se o status code retorna o valor '200':



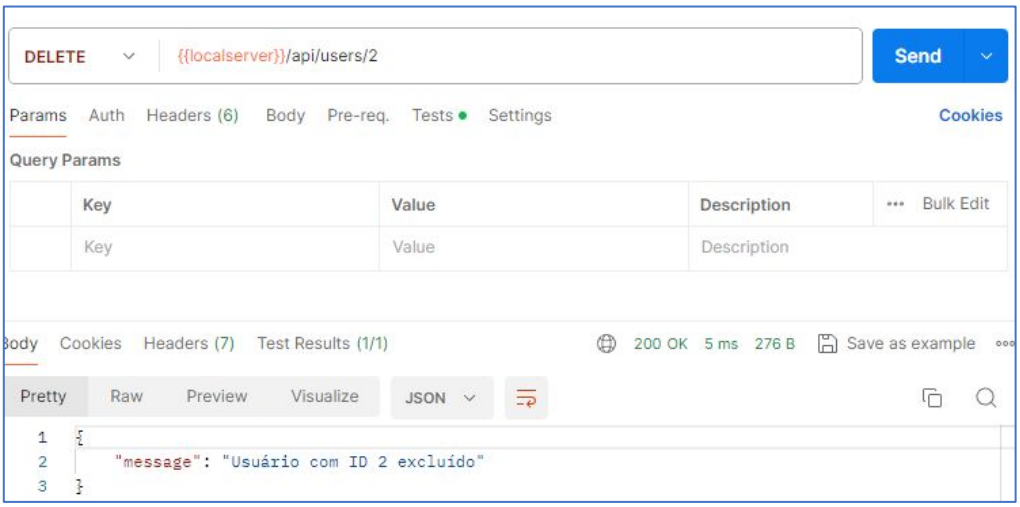
Em nova consulta GET notamos que o usuário enviado pela requisição PUT atualizou o usuário com 'id' igual a '1' com sucesso:



Testando APIs

Postman

- **DELETE**: Para criar uma nova requisição do tipo DELETE, substituir o endereço da variável 'base_url' para 'localserver' como anteriormente cadastramos, colocar o dado do 'id' do usuário como parâmetro no endereço e clicar em 'Send' para analisar a resposta.



Veja também que há um script de test na aba 'Tests' que contém uma verificação se o status code retorna o valor '200':

Em nova consulta GET notamos que o usuário enviado pela requisição DELETE atualizou o usuário com 'id' igual a '2' com sucesso: