

# Definição das rotas

Prof. Me. Ewerton J. Silva

# Relembrando...

- Abra o VS Code, caso algum outro projeto seja carregado feche o mesmo com a opção “fechar pasta”, em seguida carregue o projeto com a opção “abrir pasta”.
- Certifique-se que os arquivos que aparecem na área de navegação são todos os referentes ao seu projeto.
- Inicie o terminal no VS Code, e insira o comando “npm run dev”.
- Carregue a aplicação no navegador com o endereço “localhost:3333”

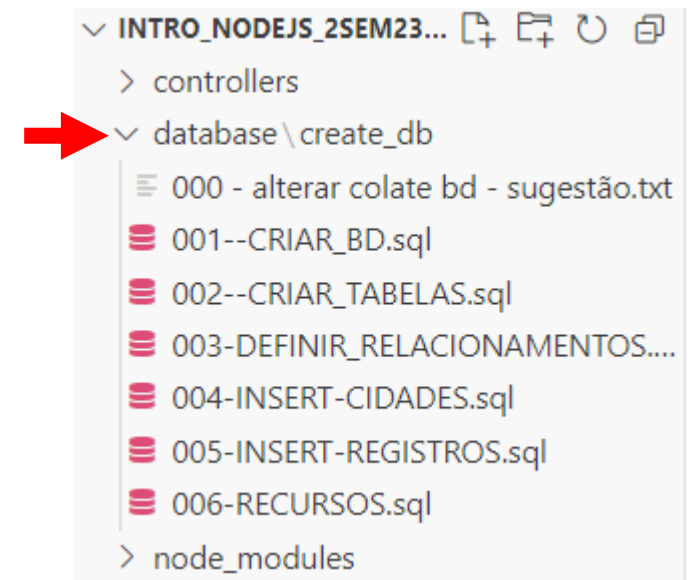
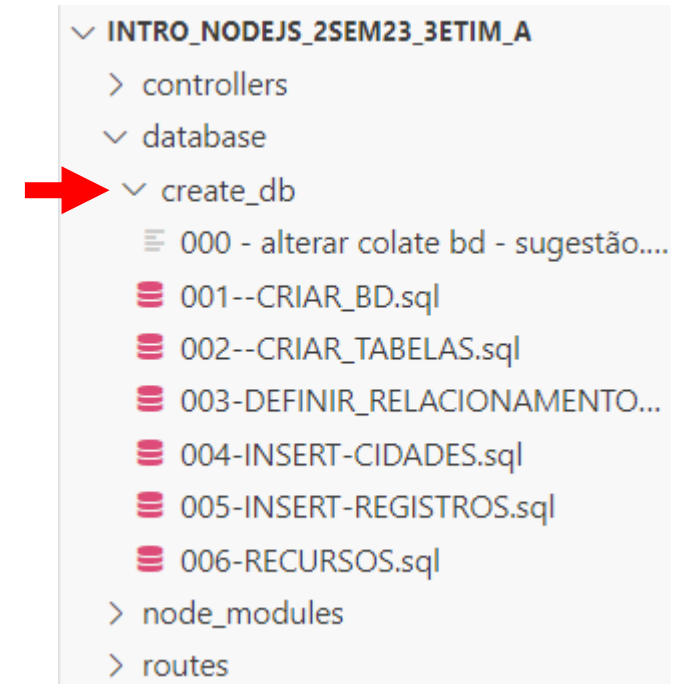
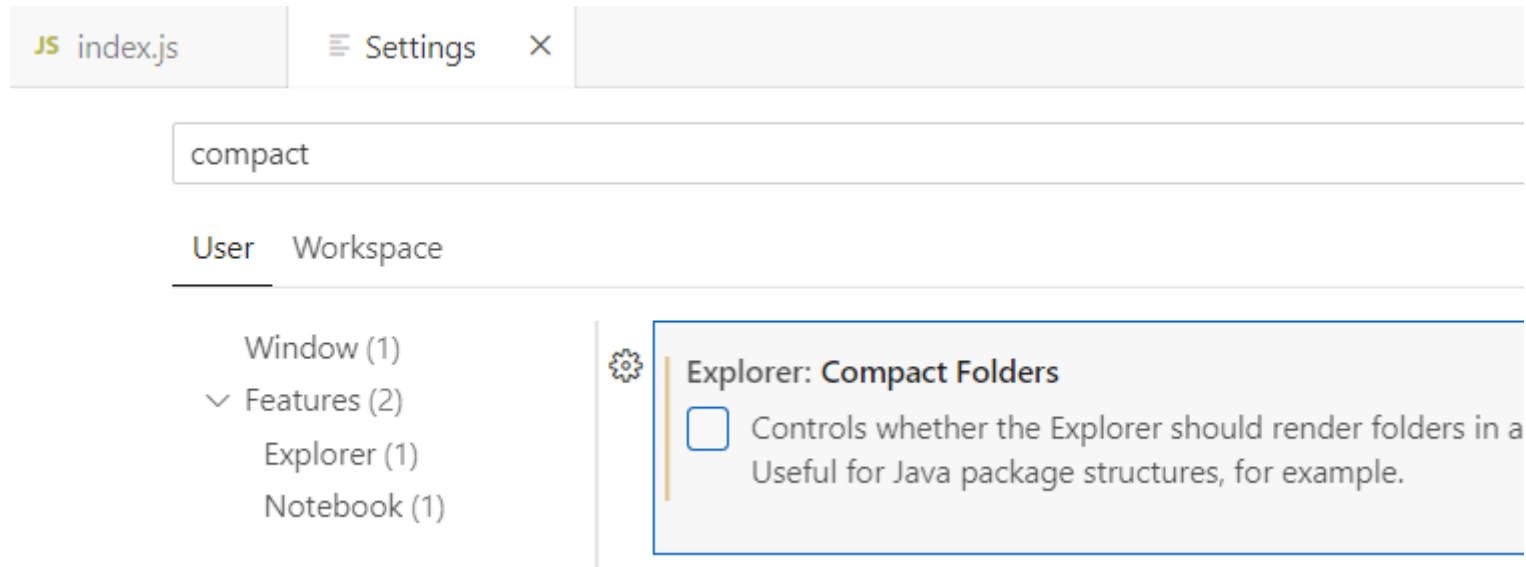
# Banco de dados

Insira na pasta “database” uma pasta chamada “create\_db” com todos os arquivos relacionados ao banco de dados.



# Obs. de uso do VSCode:

Existem duas maneiras de configurar a exibição das pastas, para altera-la basta pesquisar pela configuração “compact” e marcar ou desmarcar a opção “Compact Folders”, ao lado segue o exemplo de como cada uma é representada.

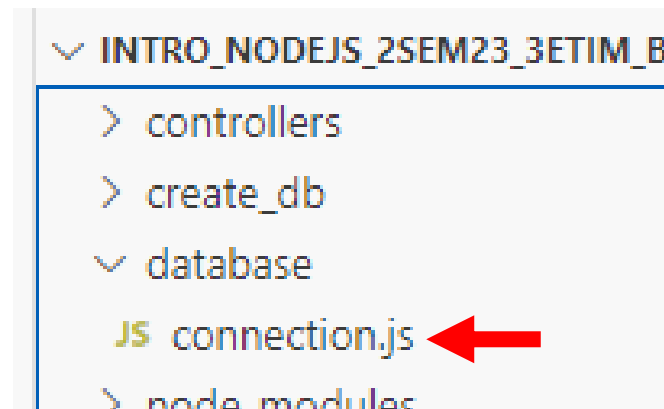


# Configurando a conexão com o banco

Tendo em vista que você já tem seu banco de dados criado e alimentado, iremos configurar o acesso a ele.

Dentro da pasta database crie um arquivo com o nome “connection.js”.

Adicione o código que vem a seguir alterando os valores de acesso para suas respectivas configurações de banco de dados.



O código ao lado configura a conexão com o banco de dados.

Para que isto funcione é necessário um arquivo .env em um projeto Node.js com Express e MySQL serve como um repositório para variáveis de ambiente, ou seja, informações de configuração que podem variar dependendo do ambiente em que sua aplicação está sendo executada (desenvolvimento, teste, produção).

```
src > database > JS connection.js > initializeDatabase
1  require('dotenv').config();
2  const mysql = require('mysql2/promise');
3
4  // Obtém as configurações do banco de dados a partir do arquivo .env
5  const config = {
6    host: process.env.BD_SERVIDOR,
7    port: process.env.BD_PORTA || 3306, // Porta padrão 3306 se não definida
8    user: process.env.BD_USUARIO,
9    password: process.env.BD_SENHA,
10   database: process.env.BD_BANCO,
11   waitForConnections: true,
12   connectionLimit: 10, // Pode ajustar conforme a necessidade
13   queueLimit: 0,
14 };
15
16 let pool;
17
18 const initializeDatabase = async () => {
19   try {
20     // Cria a pool de conexões
21     pool = mysql.createPool(config);
22
23     // Testa a conectividade com uma conexão simples
24     const connection = await pool.getConnection();
25     console.log('Conexão MySQL estabelecida com sucesso!');
26     connection.release(); // Libera a conexão de volta para a pool
27   } catch (error) {
28     console.error('Erro ao conectar ao banco de dados:', error.message);
29     process.exit(1); // Encerra o processo se a conexão falhar
30   }
31 };
32
33 // Inicializa o banco de dados ao carregar o módulo
34 initializeDatabase();
35
36 module.exports = pool;
```

## **Carregamento de variáveis de ambiente:**

```
require('dotenv').config();
```

Esta linha usa o pacote dotenv para carregar as variáveis de ambiente do arquivo .env para o process.env. Isso permite que você acesse as configurações do banco de dados (e outras configurações) como process.env.BD\_SERVIDOR, process.env.BD\_USUARIO, etc.

## **Importação do módulo mysql2/promise:**

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

O pacote mysql2/promise é uma versão do mysql2 que suporta Promises, facilitando o uso de async/await para operações de banco de dados assíncronas.

## Configuração da conexão com o banco de dados:

```
5  const config = {  
6      host: process.env.BD_SERVIDOR,  
7      port: process.env.BD_PORTA || 3306, // Porta padrão 3306 se não definida  
8      user: process.env.BD_USUARIO,  
9      password: process.env.BD_SENHA,  
10     database: process.env.BD_BANCO,  
11     waitForConnections: true,  
12     connectionLimit: 10, // Pode ajustar conforme a necessidade  
13     queueLimit: 0,  
14 };
```

- Este objeto config armazena as informações de conexão com o banco de dados MySQL.
- host, user, password e database são obtidos das variáveis de ambiente.
- port usa process.env.BD\_PORTA se definido, caso contrário, usa a porta padrão 3306.
- waitForConnections: true faz com que as solicitações aguardem se todas as conexões estiverem em uso.
- connectionLimit: 10 define o número máximo de conexões na pool.
- queueLimit: 0 desabilita a fila de solicitações, o que significa que as solicitações serão rejeitadas se não houver conexões disponíveis.



## Criação da pool de conexões:

```
16  let pool;
17
18  const initializeDatabase = async () => {
19    try {
20      // Cria a pool de conexões
21      pool = mysql.createPool(config);
22
23      // Testa a conectividade com uma conexão simples
24      const connection = await pool.getConnection();
25      console.log('Conexão MySQL estabelecida com sucesso!');
26      connection.release(); // Libera a conexão de volta para a pool
27    } catch (error) {
28      console.error('Erro ao conectar ao banco de dados:', error.message);
29      process.exit(1); // Encerra o processo se a conexão falhar
30    }
31  };
```

- A função initializeDatabase cria uma pool de conexões usando mysql.createPool(config).
- Ela obtém uma conexão da pool usando pool.getConnection() e testa a conectividade. connection.release() libera a conexão de volta para a pool após o teste.
- Se houver um erro na conexão, ele será registrado no console e o processo será encerrado.

Um pool de conexões é uma técnica essencial para aplicações que precisam interagir com bancos de dados de forma eficiente e escalável. Ele melhora o desempenho, a escalabilidade e o gerenciamento de recursos, tornando sua aplicação mais robusta e confiável.

Em contraste, usar conexões comuns (ou seja, criar uma nova conexão a cada requisição) pode levar a:

- Desempenho lento: Cada requisição teria que esperar a conexão ser estabelecida.
- Sobrecarga do banco de dados: Muitas conexões simultâneas podem sobrecarregar o banco de dados.
- Consumo excessivo de recursos: A criação e destruição constante de conexões consome mais recursos.

## **Inicialização do banco de dados::**

```
initializeDatabase();
```

Esta linha chama a função `initializeDatabase` para criar a pool de conexões quando o módulo é carregado.

## **Exportação da pool de conexões:**

```
module.exports = pool;
```

Esta linha exporta a pool de conexões para que possa ser usada em outros módulos do seu aplicativo.

# waitForConnections:

No contexto da configuração de uma pool de conexões MySQL com a biblioteca `mysql2/promise` em Node.js determina o comportamento da pool quando todas as conexões disponíveis na pool estão ocupadas (ou seja, estão sendo usadas por consultas ou transações ativas) e uma nova solicitação de conexão é feita.

Se `waitForConnections` estiver definido como `true`, a pool de conexões aguardará até que uma conexão esteja disponível para atender a solicitação antes de retornar um erro. Isso significa que, se todas as conexões estiverem em uso e uma nova solicitação for feita, a pool irá "esperar" até que uma conexão fique livre e, em seguida, alocará essa conexão para atender a nova solicitação. Isso pode ser útil em situações em que você deseja evitar erros de conexão, mesmo que isso signifique que a solicitação precise aguardar um tempo.

Por outro lado, se `waitForConnections` estiver definido como `false`, a pool de conexões retornará imediatamente um erro se todas as conexões estiverem ocupadas. Nesse caso, você precisará lidar com a lógica de erro e possivelmente implementar uma tentativa de reconexão ou uma estratégia de espera personalizada se desejar lidar com a sobrecarga de conexões de forma mais granular.

# connectionLimit

No contexto da configuração de uma pool de conexões MySQL com a biblioteca `mysql2/promise` em Node.js determina o número máximo de conexões simultâneas que podem ser mantidas na pool. Em outras palavras, ele controla quantas conexões podem estar ativas ao mesmo tempo, aguardando solicitações de consulta ou transação.

A definição adequada do valor para `connectionLimit` depende das necessidades e do dimensionamento do seu aplicativo. A seguir serão apresentadas algumas considerações ao definir o valor para `connectionLimit`:

1. Requisitos de Escala: O valor de `connectionLimit` deve ser escolhido com base na quantidade de tráfego e na carga esperada no seu aplicativo. Se você espera um alto volume de tráfego e consultas simultâneas, pode aumentar o limite para acomodar mais conexões. Por outro lado, se o tráfego é relativamente baixo, um limite menor pode ser suficiente.
2. Recursos do Servidor MySQL: Leve em consideração a capacidade do servidor MySQL em lidar com conexões simultâneas. Se o servidor MySQL não puder acomodar muitas conexões simultâneas devido a recursos limitados, não faz sentido definir um valor muito alto para `connectionLimit`.
3. Disponibilidade de Recursos do Sistema: Certifique-se de que o servidor onde o seu aplicativo Node.js está sendo executado tenha recursos adequados (CPU, memória, etc.) para suportar o número de conexões especificado em `connectionLimit`. Conexões em excesso podem levar a problemas de desempenho.
4. Monitoramento e Ajuste: É importante monitorar o desempenho do seu aplicativo e ajustar o valor de `connectionLimit` conforme necessário. Se você perceber que o aplicativo está tendo problemas de desempenho devido a um número insuficiente ou excessivo de conexões, você pode ajustar esse valor dinamicamente.

A escolha do valor correto para `connectionLimit` é uma parte crítica do dimensionamento e otimização de aplicativos que usam bancos de dados MySQL. Deve-se encontrar um equilíbrio entre ter conexões suficientes para atender às demandas do aplicativo sem sobrecarregar o servidor MySQL ou os recursos do sistema. Experimentar diferentes valores e monitorar o desempenho é uma abordagem recomendada para determinar a configuração ideal para o seu caso específico.

# queueLimit

No contexto da configuração de uma pool de conexões MySQL com a biblioteca mysql2/promise em Node.js controla o número máximo de solicitações de conexão em espera (ou seja, solicitações de conexão que não podem ser atendidas imediatamente devido ao limite de conexões simultâneas definido por connectionLimit).

Aqui estão algumas informações importantes sobre o queueLimit:

**Definição do queueLimit:** O queueLimit define quantas solicitações de conexão podem ficar em fila aguardando disponibilidade de conexões na pool. Quando todas as conexões disponíveis estão em uso (atendendo a consultas ou transações), novas solicitações de conexão são colocadas em espera na fila.

**Comportamento padrão:** Por padrão, o queueLimit é definido como 0, o que significa que não há limite para o número de solicitações em fila. Isso significa que, se todas as conexões estiverem ocupadas, um número ilimitado de solicitações de conexão será aceito na fila e aguardará até que uma conexão esteja disponível.

**Limitando a Fila:** Definir um valor positivo para o queueLimit impõe um limite ao número de solicitações de conexão em espera. Quando o limite é atingido, as solicitações adicionais serão rejeitadas imediatamente e receberão um erro, em vez de entrar na fila.

Se por exemplo o `queueLimit` é definido como 5, significa que apenas 5 solicitações de conexão podem estar em espera na fila ao mesmo tempo. Se todas as 10 conexões simultâneas estiverem ocupadas e mais de 5 solicitações de conexão forem feitas, as solicitações excedentes serão rejeitadas imediatamente com um erro.

A configuração do `queueLimit` é útil para controlar a carga no servidor e evitar que ele seja sobrecarregado com um grande número de solicitações de conexão em espera. No entanto, é importante ajustar esse valor com base nas necessidades do seu aplicativo e no dimensionamento do seu sistema para garantir que ele não seja definido muito baixo, impedindo o acesso quando necessário, nem muito alto, consumindo muitos recursos do sistema para manter solicitações em fila.

O valor 0 neste parâmetro representa que não existe um limite para a fila de solicitações, cuidado com esse valor, pois pode gerar sobrecarga no servidor.



Agora que configuramos o banco de dados partiremos o desenvolvimento dos controllers e sua utilização em conjunto com as rotas. Existem diversas formas de se organizar projetos de software, em nosso exemplo é na pasta controllers que ficam os controladores da sua aplicação. Os controladores são responsáveis por receber as requisições HTTP, processá-las e enviar as respostas. Cada rota da sua API deve ser mapeada para um controlador específico.

Ao invés de ter uma pasta chamada models, inserimos os modelos de dados que representam a estrutura e a lógica de negócios dos objetos manipulados pela sua API e a definição de esquemas de banco de dados, se você estiver usando um banco de dados dentro da pasta database/createdb.

Veja a seguir um esquema sugestivo para organização de aplicações nodejs.

# Padrão MVC (Model-View-Controller)

Pasta src: Crie uma pasta chamada src (abreviação de "source") na raiz do seu projeto. Essa pasta conterá todo o código-fonte da sua API.

Divisão em Camadas:

- Pasta controllers: Nesta pasta, coloque os controladores da sua aplicação. Os controladores são responsáveis por receber as requisições HTTP, processá-las e enviar as respostas. Cada rota da sua API deve ser mapeada para um controlador específico.
- Pasta models: Aqui, coloque os modelos de dados que representam a estrutura e a lógica de negócios dos objetos manipulados pela sua API. Isso inclui a definição de esquemas de banco de dados, se você estiver usando um banco de dados.
- Pasta routes: Configure suas rotas na pasta routes. As rotas definem as URLs e os métodos HTTP que a sua API irá tratar e mapeiam essas rotas para os controladores apropriados.
- Pasta middlewares: Se você tiver middlewares personalizados (funções que são executadas antes ou depois das rotas), coloque-os nesta pasta.
- Pasta config: Coloque arquivos de configuração, como configurações de banco de dados, variáveis de ambiente e outras configurações gerais.
- Pasta utils (opcional): Se você tiver funções utilitárias ou módulos reutilizáveis em várias partes da aplicação, pode organizá-los em uma pasta chamada utils.
- Pasta tests (opcional): Se você escrever testes automatizados para sua API, pode criar uma pasta separada chamada tests para armazenar seus casos de teste.

# Padrão MVC (Model-View-Controller)

**Arquivo de Entrada Principal:** Geralmente, você terá um arquivo de entrada principal, como `app.js` ou `index.js`, que inicializa a aplicação, configura o servidor HTTP, carrega as rotas e faz a conexão com o banco de dados, se aplicável.

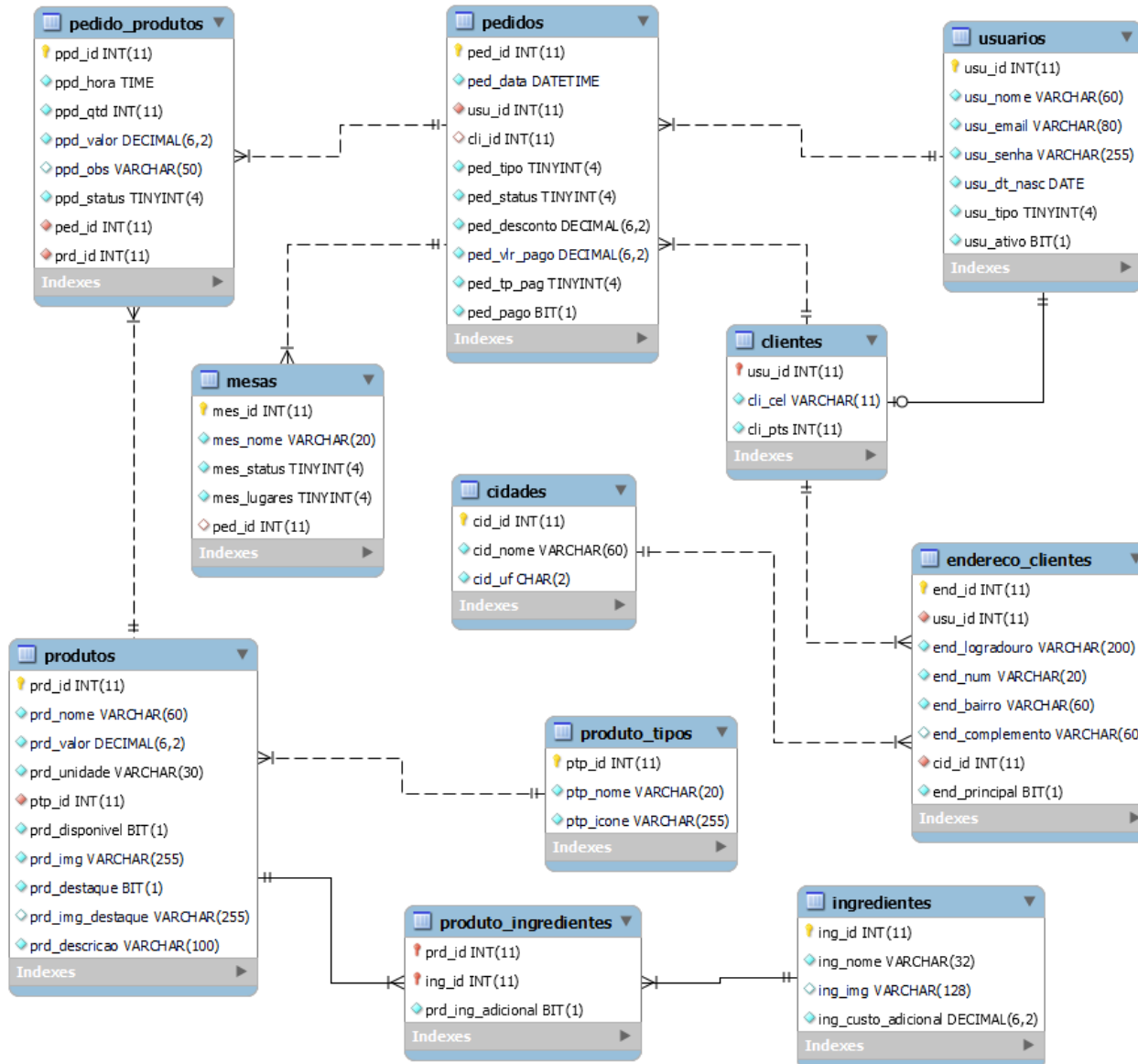
**Pasta public (opcional):** Se a sua API também servir recursos estáticos, como arquivos CSS, JavaScript ou imagens, você pode criar uma pasta `public` para armazenar esses recursos. Isso é comum em aplicações web que também fornecem uma interface do usuário.

**Gerenciador de Pacotes:** O arquivo `package.json` deve estar na raiz do projeto e conter informações sobre as dependências, scripts de inicialização e outros metadados relacionados ao seu projeto.

# O que é um controller?

Um controller como o nome já diz, é um controlador, ele é intermediário das requisições que chegam na nossa rota, após os dados chegarem na nossa rota, esses dados são enviados para o controller e desde então o controller pode tomar ações e decisões do que fazer com esses dados.

Na pasta chamada “controllers” crie e um arquivo .js para cada tabela do banco de dados.



controllers

- JS cidades.js
- JS clientes.js
- JS enderecoClientes.js
- JS ingredientes.js
- JS mesas.js
- JS pedidoProdutos.js
- JS pedidos.js
- JS produtoIngredientes.js
- JS produtos.js
- JS produtoTipos.js
- JS usuarios.js

No controller “usuários.js” insira o código abaixo para criar uma função para uma requisição de listagem de usuários:

```
controllers > JS usuarios.js > ...
1
2  const db = require('../database/connection');
3
4  module.exports = {
5      async listarUsuarios(request, response) {
6          try {
7              return response.status(200).json({
8                  sucesso: true,
9                  mensagem: 'Lista de usuários.',
10                 dados: null
11             });
12         } catch (error) {
13             return response.status(500).json({
14                 sucesso: false,
15                 mensagem: 'Erro na requisição.',
16                 dados: error.message
17             });
18         }
19     }
20 }
```





# routes

São responsáveis por determinar como o servidor deve responder a diferentes solicitações HTTP. Elas são um componente fundamental ao criar aplicativos web com Express, pois definem o comportamento da aplicação com base na URL e no método HTTP das solicitações recebidas.

Express oferece suporte a diferentes tipos de rotas para atender a várias necessidades.



# Rotas Básicas

As rotas básicas são definidas para manipular solicitações HTTP para URLs específicas. Você pode especificar a URL e o método HTTP (GET, POST, PUT, DELETE, etc.) que a rota deve tratar.

Exemplo de uma rota básica que lida com uma solicitação GET para a raiz ("/"):

```
app.get('/', (req, res) => {  
    res.send('Hello, world!');  
});
```





# Rotas com Parâmetros

Você pode definir rotas com parâmetros para capturar valores dinâmicos nas URLs. Isso é útil quando você deseja criar URLs com valores variáveis, como IDs de recursos.

Exemplo de uma rota com parâmetros para capturar um ID:

```
app.get('/user/:id', (req, res) => {  
  const userId = req.params.id;  
  // Faça algo com userId...  
});
```



# Rotas Aninhadas

Rotas aninhadas permitem agrupar rotas relacionadas sob um mesmo prefixo de URL. Isso ajuda a manter a organização do código.

Exemplo de rotas aninhadas:

```
const router = express.Router();

router.get('/profile', (req, res) => {
  // Rota /profile
});

router.get('/settings', (req, res) => {
  // Rota /settings
});

app.use('/user', router);
// Todas as rotas começando com /user serão tratadas por router
```



# Rotas com Manipuladores Múltiplos

Você pode associar vários manipuladores (funções) a uma única rota para executar tarefas diferentes, como autenticação, validação e processamento de dados.

Exemplo de uma rota com múltiplos manipuladores:

```
app.get('/protected',  
  (req, res, next) => {  
    // Middleware de autenticação  
    if (!req.isAuthenticated()) {  
      return res.status(401).send('Não autenticado');  
    }  
    next(); // Chama o próximo manipulador  
  },  
  (req, res) => {  
    // Rota protegida  
    res.send('Conteúdo protegido');  
  }  
);
```



# Rotas com Middleware Global

Você pode usar middleware globalmente para todas as rotas ou para grupos específicos de rotas. Middleware são funções que podem ser executadas antes ou depois do tratamento das solicitações, permitindo tarefas como autenticação, registro de logs e manipulação de erros.

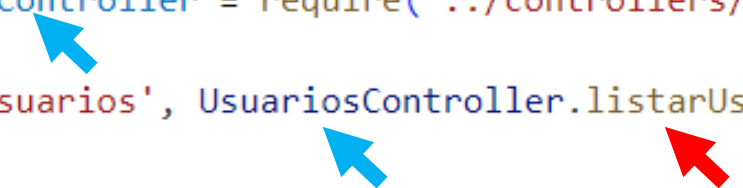
Exemplo de middleware global:

```
app.use((req, res, next) => {  
  // Middleware global para registro de logs  
  console.log(`${req.method} ${req.url}`);  
  next();  
});
```

# Configurando nosso arquivo de rotas

No projeto crie outra pasta com o nome “routes”, e dentro dela um arquivo chamado “routes.js”, é nele que devemos passar todas as rotas, o arquivo “routes.js” ficará igual ao da imagem ao lado:

```
routes > JS routes.js > ...
1  const express = require('express');
2  const router = express.Router();
3
4  const UsuariosController = require('../controllers/usuarios');
5
6  router.get('/usuarios', UsuariosController.listarUsuarios);
7
8  module.exports = router;
```



# Ajustando o arquivo index.js

Faça a referencia para os recursos apontados abaixo, sendo o “cors”, o responsável por garantir o acesso pelo front-end futuramente, e o router o arquivo com as rotas que definimos...

```
JS index.js > ...  
1  const express = require('express');  
➔ 2  const cors = require('cors');  
3  
➔ 4  const router = require('./routes/routes');  
5  
6  const app = express();  
7  
8  const porta = 3333;
```

... Ainda no “index.js” iremos adicionar como middlewares a serem executados na instância do “app” os recursos apontados na imagem abaixo, entre eles está o arquivo de rotas.

```
4   const router = require( './routes/routes' );
5
6   const app = express();
7   app.use(cors());
8   app.use(express.json());
9   app.use(router);
10
11  const porta = 3333;
12
13  app.listen(porta, () => {
```

# Teste de requisição no navegador

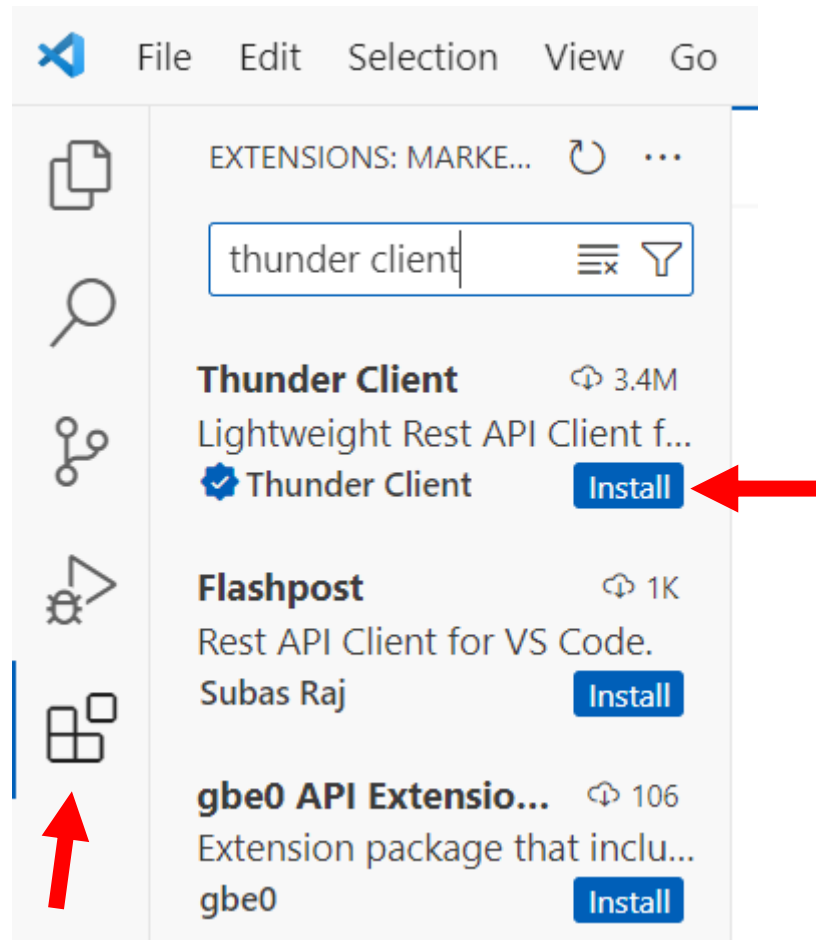


The image shows a web browser window with the address bar displaying `localhost:3333/usuarios`. Below the address bar, there are several bookmark icons: a star for 'Bookmarks', a pencil for 'AutoDraw', a colorful square for 'Centro Paula Souza...', the Gmail logo for 'Gmail', and a red 'CPS' logo for 'Moodle CPS'. The main content area of the browser displays a REST client interface with a list of lines numbered 1 to 8. The code is as follows:

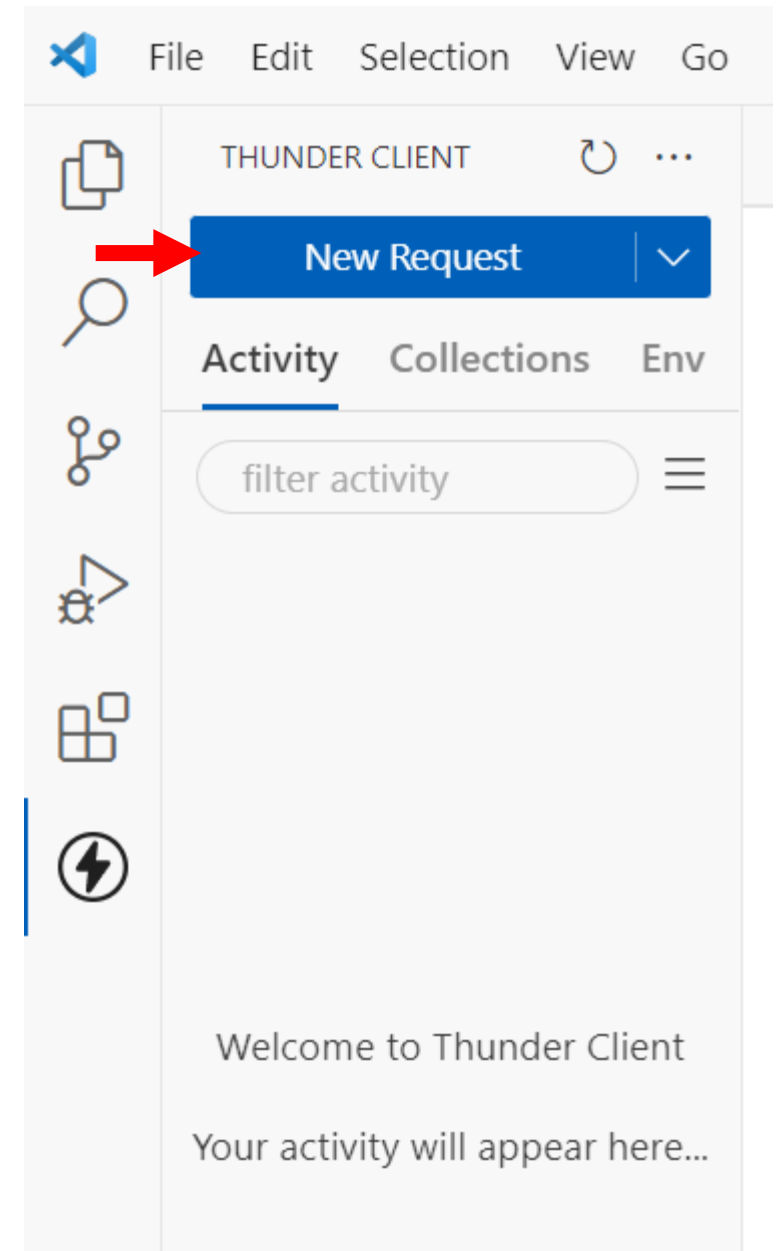
```
1 // 20240314203555
2 // http://localhost:3333/usuarios
3
4 {
5   "sucesso": true,
6   "mensagem": "Lista de usuários.",
7   "dados": null
8 }
```



# Extensão para teste de API do VSCode



Acesse o recurso e crie uma nova requisição



Insira o endereço na barra de navegação e clique em “Send” para testar a requisição criada.

The screenshot displays the Thunder Client application interface. At the top, a menu bar includes File, Edit, Selection, View, Go, Run, and a search icon. Below the menu, a toolbar shows icons for file operations and a search bar. The main workspace is divided into several panels. On the left, a sidebar contains a 'New Request' button and a list of activities, including a GET request to 'localhost:3333/usuarios' labeled 'just now'. The central panel shows the details of the selected request: a GET method to 'http://localhost:3333/usuarios'. Below this, there are tabs for Query, Headers, Auth, Body, Tests, and Pre Run. The 'Query' tab is active, showing a table for 'Query Parameters' with columns for 'parameter' and 'value'. On the right, a 'Send' button is visible. Below the 'Send' button, the response is displayed, showing a status of '200 OK', a size of '62 Bytes', and a time of '4 ms'. The response body is a JSON object: { "sucesso": true, "mensagem": "Lista de usuários.", "dados": null }. Red arrows highlight the URL in the address bar, the 'Send' button, and the response body.

THUNDER CLIENT

File Edit Selection View Go Run ...

api\_nodejs\_2des\_a\_2sem\_24

JS usuarios.js Extension: Thunder Client TC Release Notes TC New Request X JS routes.js JS index.js

New Request

Activity Collections Env

filter activity

GET localhost:3333/usuarios just now

GET http://localhost:3333/usuarios

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter	value

Send

Status: 200 OK Size: 62 Bytes Time: 4 ms

Response Headers 7 Cookies Results Docs

```
1 {
2   "sucesso": true,
3   "mensagem": "Lista de usuários.",
4   "dados": null
5 }
```

# Adicione as requisições de inserção, edição e exclusão nos controllers

controllers > JS usuarios.js > ...

```
1
2  const db = require('../database/connection');
3
4  module.exports = {
5    async listarUsuarios(request, response) {
6      try {
7        // throw new Error('Eu causei o erro!');
8        return response.status(200).json({
9          sucesso: true,
10         mensagem: 'Lista de usuários.',
11         dados: null
12       });
13     } catch (error) {
14       return response.status(500).json({
15         sucesso: false,
16         mensagem: 'Erro na requisição.',
17         dados: error.message
18       });
19     }
20   },
21   async cadastrarUsuarios(request, response) {
22     // ...
23   }
24 }
```


Observe que as novas requisições devem ser adicionadas uma após a outra, dentro do escopo do objeto exportado.

Cada nova requisição é adicionada após a “,” da anterior.

Observe os números das linhas para não se perder no código.

# Cadastrar

```
20     },
21     async cadastrarUsuarios(request, response) {
22         try {
23             return response.status(200).json({
24                 sucesso: true,
25                 mensagem: 'Cadastro de usuários.',
26                 dados: null
27             });
28         } catch (error) {
29             return response.status(500).json({
30                 sucesso: false,
31                 mensagem: 'Erro na requisição.',
32                 dados: error.message
33             });
34         }
35     },
```




# Editar

```
35     },
36     async editarUsuarios(request, response) {
37         try {
38             return response.status(200).json({
39                 sucesso: true,
40                 mensagem: 'Editar usuários.',
41                 dados: null
42             });
43         } catch (error) {
44             return response.status(500).json({
45                 sucesso: false,
46                 mensagem: 'Erro na requisição.',
47                 dados: error.message
48             });
49         }
50     },
```



# Apagar

```
50     },
51     async apagarUsuarios(request, response) {
52         try {
53             return response.status(200).json({
54                 sucesso: true,
55                 mensagem: 'Apagar usuários.',
56                 dados: null
57             });
58         } catch (error) {
59             return response.status(500).json({
60                 sucesso: false,
61                 mensagem: 'Erro na requisição.',
62                 dados: error.message
63             });
64         }
65     },
66 }
67
```



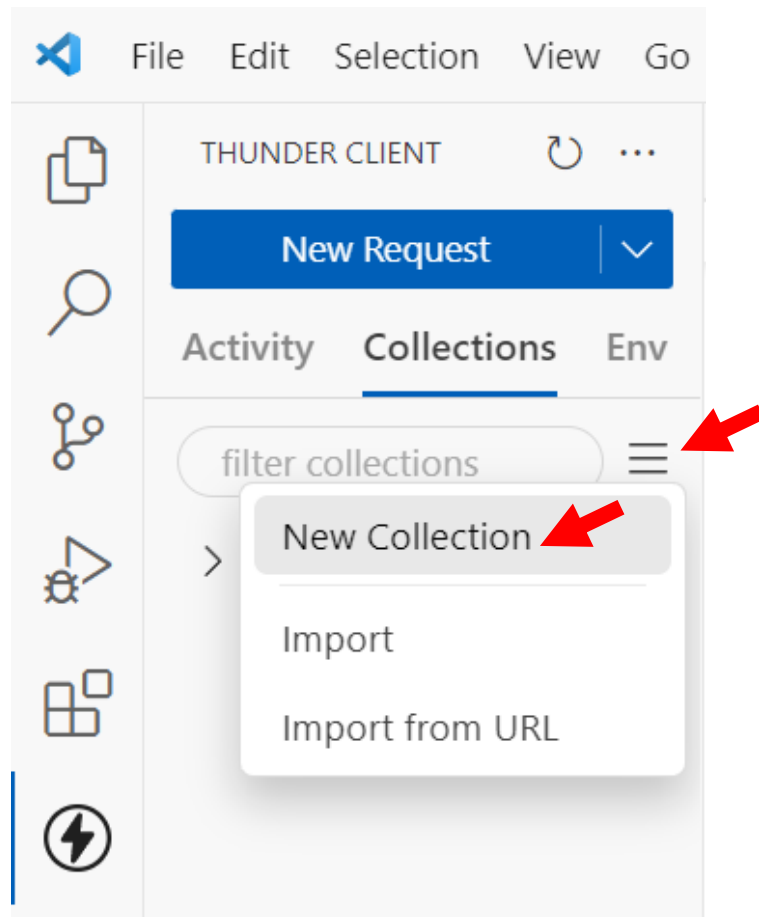
Em seguida, defina as rotas para os métodos adicionados no controller.

```
router.get('/usuarios', UsuariosController.listarUsuarios);  
router.post('/usuarios', UsuariosController.cadastrarUsuarios);  
router.patch('/usuarios', UsuariosController.editarUsuarios);  
router.delete('/usuarios', UsuariosController.apagarUsuarios);
```

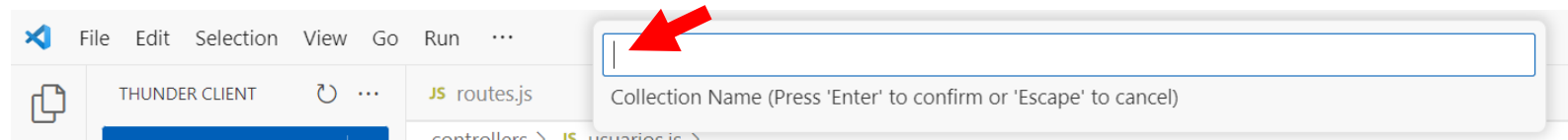




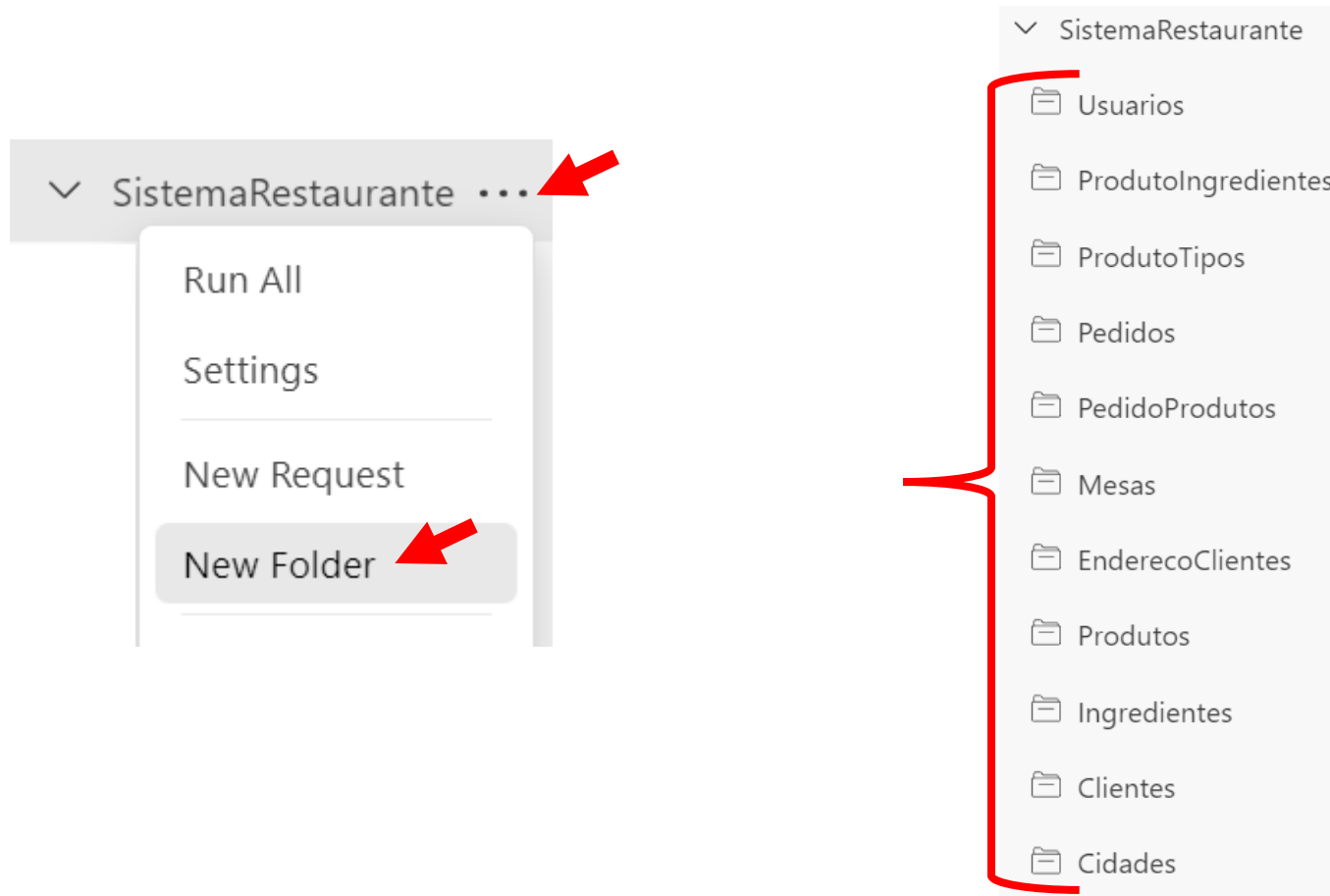
# Thunder Client



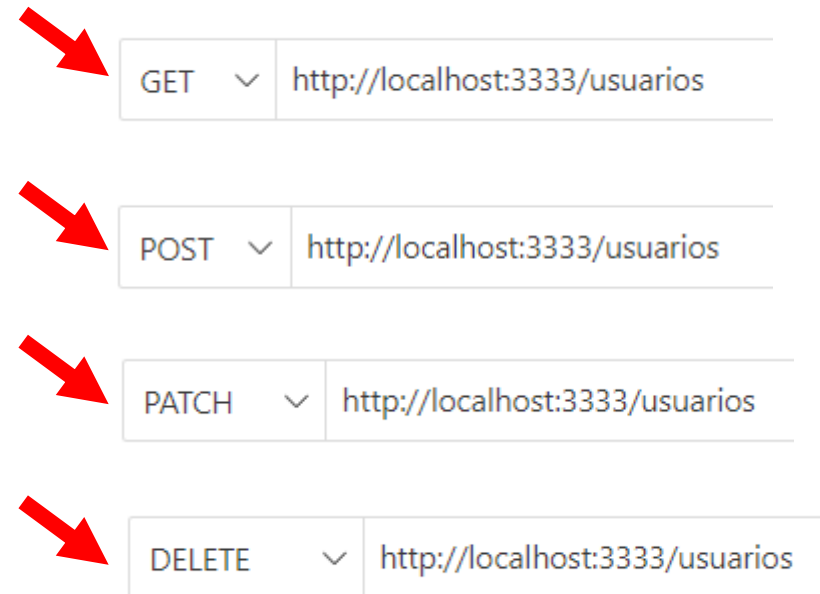
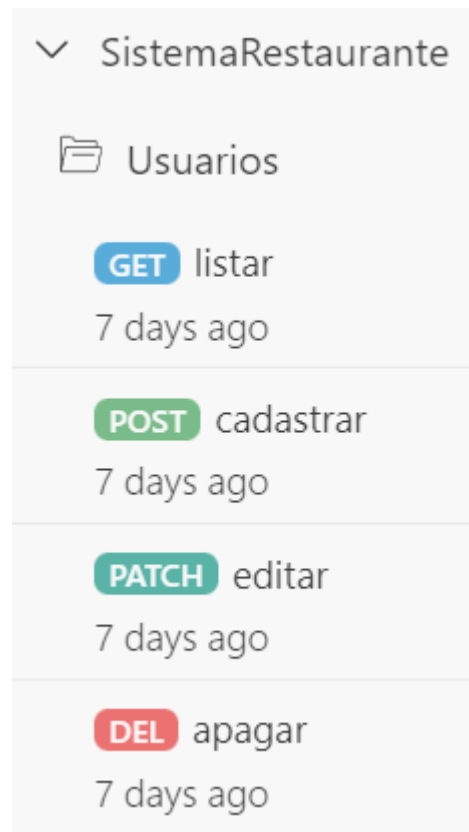
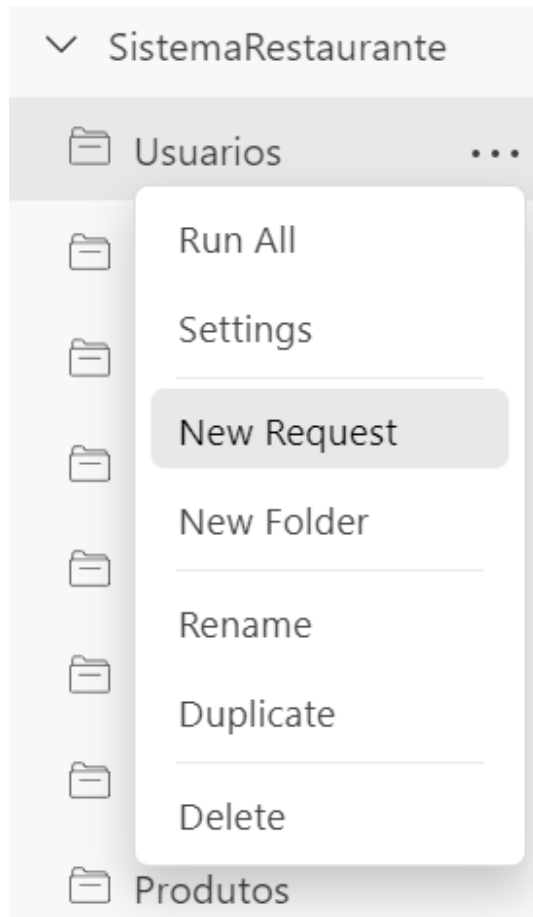
Agora crie uma collection para organizar melhor os testes para requisições.  
Clique no menu de opções para selecionar a opção “New Collection”, em seguida irá aparecer uma janela centralizada no topo do VSCode para dar um nome “Sistema Restaurante” para a Collection.



Agora nas opções da collection, selecione a opção “New Folder” para criar uma pasta para cada tabela do banco de dados.



# Em cada pasta deve ser adicionado um método http: get, post, patch e del



# Teste o método GET

GET ⌵ http://localhost:3333/usuarios Send

Query

Headers <sup>2</sup>

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

Status: 200 OK   Size: 62 Bytes   Time: 11 ms

Response

Headers <sup>7</sup>

Cookies

Results

Docs

```
1  {
2    "sucesso": true,
3    "mensagem": "Lista de usuários.",
4    "dados": null
5  }
```

# Teste o método POST

POST ▼ http://localhost:3333/usuarios Send

Query

Headers <sup>2</sup>

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

Status: 200 OK   Size: 65 Bytes   Time: 3 ms

Response

Headers <sup>7</sup>

Cookies

Results

Docs

```
1  {
2    "sucesso": true,
3    "mensagem": "Cadastro de usuários.",
4    "dados": null
5  }
```

# Teste o método PATCH

PATCH

⌵

http://localhost:3333/usuarios

Send

Query

Headers 2

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

Status: 200 OK    Size: 60 Bytes    Time: 2 ms

Response

Headers 7

Cookies

Results

Docs

```
1 {
2   "sucesso": true,
3   "mensagem": "Editar usuários.",
4   "dados": null
5 }
```

# Teste o método DELETE

DELETE

▼

http://localhost:3333/usuarios

Send

Query

Headers <sup>2</sup>

Auth

Body

Tests

Pre Run

Query Parameters

☐

parameter

value

Status: 200 OK

Size: 60 Bytes

Time: 3 ms

Response

Headers <sup>7</sup>

Cookies

Results

Docs

1

{

2

"sucesso": true,

3

"mensagem": "Apagar usuários.",

4

"dados": null

5

}

Em cada um dos controllers insira o código referente aos 4 métodos, lembrando de adaptar o nome da função e o response de acordo com o nome da tabela.

```
controllers > JS usuarios.js > ...
1
2  const db = require('../database/connection');
3
4  module.exports = {
5    async listarUsuarios(request, response) {
6      try {
7        return response.status(200).json({
8          sucesso: true,
9          mensagem: 'Lista de usuários.',
10         dados: null
11       });
12     } catch (error) {
13       return response.status(500).json({
14         sucesso: false,
15         mensagem: 'Erro na requisição.',
16         dados: error.message
17       });
18     }
19   }
20 }
```





controllers > JS usuarios.js > ...

```
1
2 const db = require('../database/connection');
```

3 controllers > JS productoTipos.js > ...

```
4 1
5 2 const db = require('../database/connection');
```

6 3
7 4 module controllers > JS productos.js > ...

```
8 5
9 6 2 const db = require('../database/connection');
```

10 7
11 8 module controllers > JS pedidos.js > ...

```
12 9
1 2 const db = require('../database/connection');
```

10 3
11 4 module controllers > JS pedidoProductos.js > ...

```
12 5
1 2 const db = require('../database/connection');
```

10 3
11 4 module controllers > JS enderecoClientes.js > ...

```
12 5
1 2 const db = require('../database/connection');
```

10 3
11 4 module controllers > JS clientes.js > ...

```
12 5
1 2 const db = require('../database/connection');
```

```
10 3
11 4 module.exports = {
12 5     async listarClientes(request, response) {
```

```
10 6
11 7
12 8
```

```
10 9
11 10
12 11
```

```
10 12
11 13
12 14
```

```
10 15
11 16
12 17
```

Em seguida importe todos os controllers.

routes > JS routes.js > ...

```
1  const express = require('express');
2  const router = express.Router();
3
4  // referência a controllers que serão utilizados nas rotas
5  const UsuariosController = require('../controllers/usuarios');
6  const ProdutosController = require('../controllers/produtos');
7  const IngredientesController = require('../controllers/ingredientes');
8  const ClientesController = require('../controllers/clientes');
9  const CidadesController = require('../controllers/cidades');
10 const EnderecoClientesController = require('../controllers/enderecoClientes');
11 const MesasController = require('../controllers/mesas');
12 const PedidoProdutosController = require('../controllers/pedidoProdutos');
13 const PedidosController = require('../controllers/pedidos');
14 const ProdutoIngredientesController = require('../controllers/produtoIngredientes');
15 const ProdutoTiposController = require('../controllers/produtoTipos');
16
17
18 router.get('/usuarios', UsuariosController.listarUsuarios);
19 router.post('/usuarios', UsuariosController.cadastrarUsuarios);
```

E defina as rotas para todos, mantendo inclusive o código comentado para cada um.

```
18 router.get('/usuarios', UsuariosController.listarUsuarios);
19 router.post('/usuarios', UsuariosController.cadastrarUsuarios);
20 router.patch('/usuarios', UsuariosController.editarUsuarios);
21 router.delete('/usuarios', UsuariosController.apagarUsuarios);
22
23 router.get('/produtos', ProdutosController.listarProdutos);
24 router.post('/produtos', ProdutosController.cadastrarProdutos);
25 router.patch('/produtos', ProdutosController.editarProdutos);
26 router.delete('/produtos', ProdutosController.apagarProdutos);
27
28 router.get('/ingredientes', IngredientesController.listarIngredientes);
29 router.post('/ingredientes', IngredientesController.cadastrarIngredientes);
30 router.patch('/ingredientes', IngredientesController.editarIngredientes);
31 router.delete('/ingredientes', IngredientesController.apagarIngredientes);
32
33 router.get('/clientes', ClientesController.listarClientes);
34 router.post('/clientes', ClientesController.cadastrarClientes);
35 router.patch('/clientes', ClientesController.editarClientes);
36
37 router.get('/cidades', CidadesController.listarCidades);
38
```

```
39 router.get('/enderecoclientes', EnderecoClientesController.listarEnderecoClientes);
40 router.post('/enderecoclientes', EnderecoClientesController.cadastrarEnderecoClientes);
41 router.patch('/enderecoclientes', EnderecoClientesController.editarEnderecoClientes);
42 router.delete('/enderecoclientes', EnderecoClientesController.apagarEnderecoClientes);
43
44 router.get('/mesas', MesasController.listarMesas);
45 router.post('/mesas', MesasController.cadastrarMesas);
46 router.patch('/mesas', MesasController.editarMesas);
47 router.delete('/mesas', MesasController.apagarMesas);
48
49 router.get('/pedidoprodutos', PedidoProdutosController.listarPedidoProdutos);
50 router.post('/pedidoprodutos', PedidoProdutosController.cadastrarPedidoProdutos);
51 router.patch('/pedidoprodutos', PedidoProdutosController.editarPedidoProdutos);
52 router.delete('/pedidoprodutos', PedidoProdutosController.apagarPedidoProdutos);
53
54 router.get('/pedidos', PedidosController.listarPedidos);
55 router.post('/pedidos', PedidosController.cadastrarPedidos);
56 router.patch('/pedidos', PedidosController.editarPedidos);
57 router.delete('/pedidos', PedidosController.apagarPedidos);
58
59 router.get('/produtoingredientes', ProdutoIngredientesController.listarProdutoIngredientes);
60 router.post('/produtoingredientes', ProdutoIngredientesController.cadastrarProdutoIngredientes);
61 router.patch('/produtoingredientes', ProdutoIngredientesController.editarProdutoIngredientes);
62 router.delete('/produtoingredientes', ProdutoIngredientesController.apagarProdutoIngredientes);
63
64 router.get('/produtotipos', ProdutoTiposController.listarProdutoTipos);
65 router.post('/produtotipos', ProdutoTiposController.cadastrarProdutoTipos);
66 router.patch('/produtotipos', ProdutoTiposController.editarProdutoTipos);
67 router.delete('/produtotipos', ProdutoTiposController.apagarProdutoTipos);
68
69 module.exports = router;
70
```

# Salve o arquivo e teste todas as rotas no ThunderClient.

Method	URL	Status	Size	Time				
GET	http://localhost:3333/pedidos	200 OK	60 Bytes	8 ms				
<p>Query Parameters</p> <table border="1"><thead><tr><th>parameter</th><th>value</th></tr></thead><tbody><tr><td><input type="checkbox"/></td><td></td></tr></tbody></table>		parameter	value	<input type="checkbox"/>		<p>Response</p> <pre>1 { 2   "sucesso": true, 3   "mensagem": "Lista de Pedidos.", 4   "dados": null 5 }</pre>		
parameter	value							
<input type="checkbox"/>								
POST	http://localhost:3333/produtos	200 OK	64 Bytes	3 ms				
<p>Query Parameters</p> <table border="1"><thead><tr><th>parameter</th><th>value</th></tr></thead><tbody><tr><td><input type="checkbox"/></td><td></td></tr></tbody></table>		parameter	value	<input type="checkbox"/>		<p>Response</p> <pre>1 { 2   "sucesso": true, 3   "mensagem": "Cadastro de produtos.", 4   "dados": null 5 }</pre>		
parameter	value							
<input type="checkbox"/>								
PATCH	http://localhost:3333/produtos	200 OK	59 Bytes	2 ms				
<p>Query Parameters</p> <table border="1"><thead><tr><th>parameter</th><th>value</th></tr></thead><tbody><tr><td><input type="checkbox"/></td><td></td></tr></tbody></table>		parameter	value	<input type="checkbox"/>		<p>Response</p> <pre>1 { 2   "sucesso": true, 3   "mensagem": "Editar produtos.", 4   "dados": null 5 }</pre>		
parameter	value							
<input type="checkbox"/>								