Objetivo:

I. Estrutura de uma aplicação do lado servidor;
II. Node e Express;
III. Criar servidor Node com Express;
IV. Definição de rotas;
V. Rotas para arquivo estático;
VI. Hierarquia de rotas.

I. Estrutura de uma aplicação do lado servidor

Navegadores se comunicam com o servidor web usando o protocolo HTTP. O Apache Tomcat, Apache PHP e Node com Express são exemplos de servidores web para rodar aplicações do lado back-end.

A Figura 1 representa uma requisição HTTP, ela possui os objetos Request e Response. O objeto Request inclui a URL (Uniform Resource Locator), o método que define a ação da requisição (GET, POST, PUT e DELETE), informações adicionais da URL, assim como os parâmetros e o corpo da requisição.

Na URL

```
http://localhost:3000?nome=Ana&idade=21
```

nome e idade são parâmetros.

O objeto Response possui a mensagem de resposta com o código de status (200 OK, 401 Unauthorized, 404 Not Found etc.) e o corpo da resposta em caso de sucesso.

A representação da Figura 1 é de uma aplicação dinâmica, isto é, o objeto Response da requisição será de acordo com o resultado gerado pelo programa (Web Application). Existem também as aplicações estáticas, mas elas retornam sempre o mesmo conteúdo, isto é, elas não possuem a parte da Web Application.

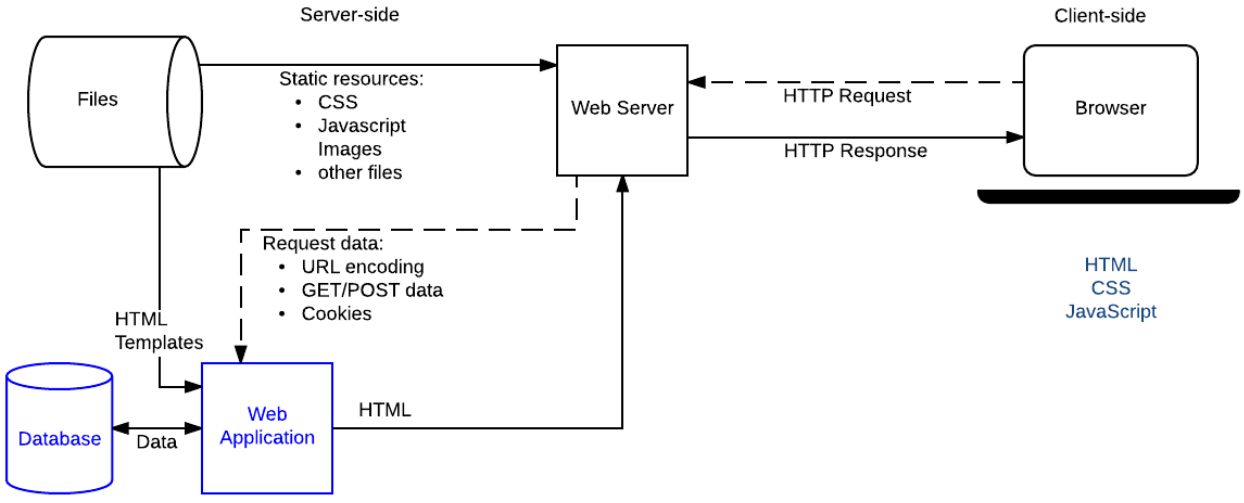


Figura 1 – Representação de uma requisição HTTP.

(Fonte: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction)

As respostas de requisição HTTP podem conter diferentes códigos de status que indicam o resultado da solicitação. Cada código de status é uma representação numérica de três dígitos que informa ao cliente o resultado da requisição. As respostas são agrupadas em cinco classes:

- 100 a 199 (respostas de informação): indica que a requisição foi recebida e o servidor continua processando-a;
- 200 a 299 (respostas de sucesso): indica que a requisição foi recebida, entendida e aceita com sucesso:
  - 200 OK: a requisição foi bem-sucedida;
  - 201 Created: a requisição foi bem-sucedida e resultou na criação de um novo recurso;
  - 204 No Content: a requisição foi bem-sucedida, mas não há conteúdo para ser retornado (por exemplo, em uma requisição DELETE).
- 300 a 399 (redirecionamentos): indica que a requisição precisa de ações adicionais para ser concluída:
  - 301 Moved Permanently: a URI do recurso solicitado foi alterada permanentemente e a nova URI é fornecida na resposta;
  - 302 Found / 303 See Other: a URI do recurso solicitado foi temporariamente alterada. O cliente deve redirecionar para a URI fornecida na resposta;
  - 304 Not Modified: indica que o recurso solicitado não foi modificado desde a última requisição.
- 400 a 499 (erros do cliente): indica que houve um erro por parte do cliente na requisição:
  - 400 Bad Request: a requisição foi malformada ou incompreensível para o servidor;
  - 401 Unauthorized: o cliente não foi autorizado a acessar o recurso;
  - 403 Forbidden: o cliente não tem permissão para acessar o recurso;
  - 404 Not Found: o recurso solicitado não foi encontrado no servidor;
  - 409 Conflict: o servidor não pode completar a requisição devido a um conflito no estado atual do recurso.
- 500 a 599 (erros do servidor): indica que houve um erro no servidor ao processar a requisição:
  - 500 Internal Server Error: o servidor encontrou uma situação inesperada que o impediu de atender à requisição;
  - 502 Bad Gateway: o servidor atuando como um gateway ou proxy recebeu uma resposta inválida do servidor upstream;
  - 503 Service Unavailable: o servidor não está pronto para lidar com a requisição. Geralmente, isso ocorre quando o servidor está em manutenção ou sobrecarregado.

Esses são alguns dos códigos de erro mais comuns usados em respostas de requisição HTTP. É importante compreender esses códigos para interpretar corretamente as respostas do servidor e lidar com os diferentes cenários que podem ocorrer durante a comunicação entre o cliente e o servidor. Para mais detalhes https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status.

II. Node e Express

Node (ou formalmente Node.js) é um ambiente em tempo de execução (runtime) open-source (código aberto) e multiplataforma que permite a execução de código JS no lado servidor. Node destina-se a ser usado fora do contexto de um navegador, ou seja, executando diretamente no computador ou servidor. Como tal, o ambiente omite APIs JS específicas do

navegador e adiciona suporte para APIs de SO, incluindo bibliotecas HTTP e manipulação de arquivos (https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction).

Podemos criar um servidor web usando apenas o pacote HTTP padrão do Node. Porém algumas tarefas do desenvolvimento web não são suportadas diretamente pelo Node. Por exemplo, manipulação específica de requisições HTTP (GET, POST, PUT e DELETE), criação de caminhos de URL (roteamento), servir arquivos estáticos ou usar modelos para criar dinamicamente a resposta. A solução é escrevermos esse código ou usarmos alguma biblioteca pronta, assim como o Express (https://expressjs.com/en/5x/api.html).

O Express é um framework popular e amplamente usado para criar aplicativos web e APIs (Application Programming Interface - Interface de Programação de Aplicação) usando o Node. O Express oferece uma abstração de alto nível para lidar com várias tarefas comuns ao desenvolver aplicativos web, como roteamento, tratamento de requisições HTTP, manipulação de middlewares (funções intermediárias) e suporte para resposta em diferentes formatos de dados (JSON, texto, HTML etc.). Com o Express, os desenvolvedores podem criar facilmente servidores web e APIs de forma rápida e eficiente, economizando tempo e esforço.

> Node.js é uma plataforma baseada em JS que permite a execução de código JS no servidor. O Express é um framework web minimalista para Node.js que facilita a criação de aplicativos web e APIs.

III. Criar servidor Node com Express

A seguir tem-se os passos para criar um servidor usando o framework Express (https://www.npmjs.com/package/express):

1. Crie a pasta aula2 (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
2. Abra a pasta no VS Code e acesse o terminal do VS Code;
3. No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node (arquivo package.json). O parâmetro `init` é usado para indicar ao programa `npm` que é para ser criado o arquivo package.json e o parâmetro `-y` (yes) é para não perguntar os valores de cada propriedade do JSON a ser criado.

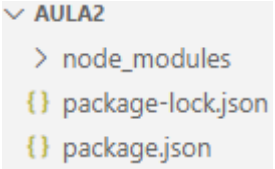
```
PS D:\aula2> npm init -y
Wrote to D:\aula2\package.json:
{
  "name": "aula2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Foi criado o arquivo package.json com um JSON com estas propriedades

4. No terminal, execute o comando `npm i express` para instalar o pacote express;

```
PS D:\aula2> npm i express
added 58 packages, and audited 78 packages in 4s
8 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Estrutura do projeto:



5. No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém os tipos de declaração (definições de tipos) para o pacote express. Essas declarações de tipo são usadas pelo TS para fornecer informações sobre os tipos de dados e as interfaces fornecidas pelo express.

   Quando usamos um pacote é preciso ter acesso às declarações de tipo do pacote para que o TS saiba quais tipos de dados esperar do framework.

   O parâmetro `-D` indica que o pacote será instalado como dependência de desenvolvimento.

```
PS D:\aula2> npm i -D @types/express
added 10 packages, and audited 88 packages in 5s
8 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

6. No terminal, execute o comando `npm i -D ts-node ts-node-dev typescript` para instalar os pacotes ts-node, ts-node-dev e typescript como dependências de desenvolvimento;

```
PS D:\aula2> npm i -D ts-node ts-node-dev typescript
added 43 packages, and audited 131 packages in 11s
15 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

   Observe que os nomes e versões dos pacotes foram colocados nas propriedades `dependencies` e `devDependencies` do arquivo package.json:

```json
{
  "name": "aula2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/express": "^4.17.17",
    "ts-node": "^10.9.1",
    "ts-node-dev": "^2.0.0",
    "typescript": "^5.1.6"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

> A numeração de versões em um pacote geralmente segue o formato "`major.minor.patch`". Como exemplo, o pacote `express` instalado possui a versão `4.18.2`, isso significa que é a versão `4` do pacote, a `18ª` versão da versão major (principal) e a `2ª` correção para a versão `4.18`.

Quando especificamos uma dependência no package.json com o símbolo ^, significa que estamos permitindo que o gerenciador de pacotes (npm ou yarn) instale qualquer versão que tenha o mesmo número da versão principal (major), mas com atualizações menores (minor) e correções de bugs (patch), mas não com atualizações que mudem a versão principal (major).

Como exemplo, a dependência `"express": "^4.18.2"` diz que o gerenciador de pacotes pode instalar qualquer versão que seja 4.x.x, como 4.19.0, 4.18.3, mas não permitirá instalar uma versão 5.x.x, pois isso representaria uma mudança de versão principal.

Diferença entre ts-node e ts-node-dev:

- ts-node: é uma ferramenta que permite executar arquivos TS diretamente no Node sem a necessidade de compilar manualmente o código para JS antes da execução. Ele age como um compilador TS "just-in-time" (JIT) para o Node, ou seja, executamos arquivos TS no Node como se estivessem sendo executados em um ambiente TS;
- ts-node-dev: é uma variação do ts-node que adiciona suporte a reinicialização automática do servidor (reloading) sempre que houver alterações nos arquivos TS. O ts-node-dev é adequado somente durante o desenvolvimento, pois nos permite fazer alterações no código e ver os resultados imediatamente, sem precisar reiniciar manualmente o servidor a cada vez.

7. No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo tsconfig.json):

```
PS D:\aula2> tsc --init
Created a new tsconfig.json with:
  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true
```

Apesar do arquivo ter várias propriedades, essas não estão comentadas

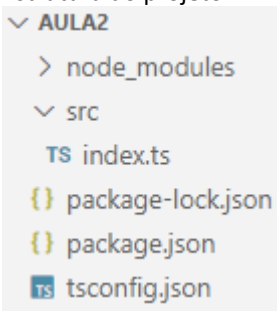
8. Crie a pasta src na raiz do projeto;
9. Crie o arquivo index.ts na pasta src;
10. Adicione na propriedade `scripts`, do package.json, os comandos para executar o arquivo index.ts:

```
"scripts": {
  "start": "ts-node ./src/index",
  "dev": "ts-node-dev ./src/index"
},
```

Observação: se o comando `ts-node` não funcionar, então use o programa `npx` para executar os comandos ou instale o pacote ts-node globalmente usando npm i -g ts-node.

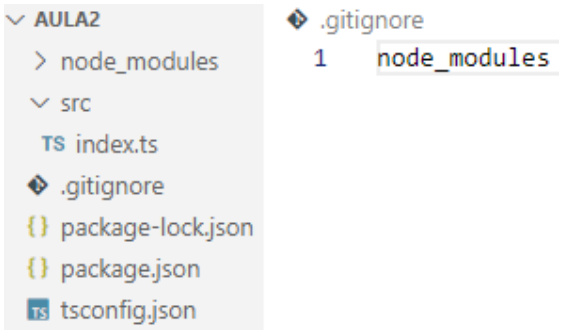
```
"scripts": {
  "start": "npx ts-node ./src/index",
```

Estrutura do projeto:

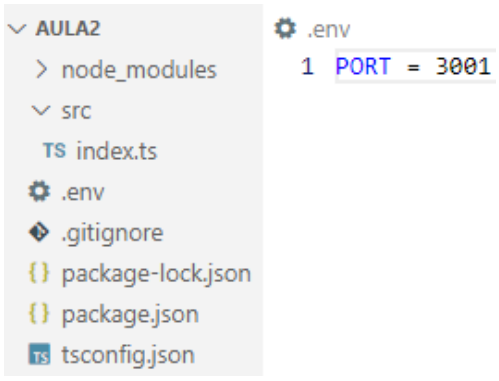


```
    "dev": "npx ts-node-dev ./src/index"
  },
```

11. Crie o arquivo .gitignore na raiz do projeto e coloque nele a instrução para evitar a pasta node_modules ao fazer deploy da aplicação na nuvem ou subir a aplicação no repositório do GitHub;

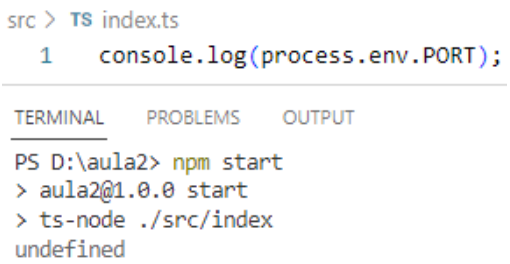


12. Crie o arquivo .env na raiz do projeto e coloque nele a variável PORT. Posteriormente, faremos uso da variável PORT;



> O arquivo .env é usado no projeto Node para armazenar variáveis de ambiente. Essas variáveis são informações sensíveis ou configurações específicas do ambiente que o aplicativo precisa para funcionar corretamente. Elas podem incluir senhas, chaves de API, credenciais de BD etc.
>
> A função do arquivo .env é fornecer uma maneira de definir e gerenciar variáveis de ambiente sem incluí-las diretamente no código fonte. Desta forma, através do .gitignore podemos excluir o arquivo .env do controle de versões, evitando a exposição de dados sensíveis. Cada desenvolvedor deve criar seu próprio arquivo .env com suas próprias configurações de ambiente específicas.

13. As variáveis de ambientes são acessadas através do objeto `process.env`. Porém, as variáveis do arquivo .env não são carregadas pelo ambiente de execução do Node no objeto `process.env`. Veja o exemplo a seguir:



Uma das maneiras mais comuns de carregar as variáveis definidas no arquivo .env é usar a biblioteca dotenv (https://www.npmjs.com/package/dotenv). Digite o comando a seguir para instalar a biblioteca no projeto:

```
PS D:\aula2> npm i dotenv
```

As variáveis do arquivo .env serão carregadas no objeto `process.env` a partir do momento que a função `config` for chamada.

```
src > TS index.ts
  1   import dotenv from "dotenv";
  2   dotenv.config();
  3
  4   console.log(process.env.PORT);

TERMINAL   PROBLEMS   OUTPUT

PS D:\aula2> npm start
> aula2@1.0.0 start
> ts-node ./src/index
3001
```

Usamos fazer um or (`||`) da variável de ambiente `PORT` com um número, desta forma, usaremos o número `3000` se a variável de ambiente `PORT` não estiver disponível:

```
import dotenv from "dotenv";
dotenv.config();
// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;

console.log(PORT);
```

`process.env` é um objeto global no ambiente de execução do Node que contém as variáveis de ambiente do sistema operacional. Uma das maneiras mais comuns de configurar variáveis de ambiente no Node é usando o arquivo .env.

14. O código a seguir é usado para criar um servidor que assiste a porta 3001:

```
import express from "express";
import dotenv from "dotenv";
dotenv.config();
// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
// cria o servidor e coloca na variável app
const app = express();
// inicializa o servidor na porta especificada
app.listen(PORT, ()=>{
    console.log(`Rodando na porta ${PORT}`);
});
```

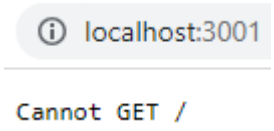
A aplicação pode ser executada usando `ts-node` ou `ts-node-dev`. Durante o desenvolvimento é melhor usar `ts-node-dev` para a aplicação ser reinicializada a cada vez que o código é salvo.

```
PS D:\aula2> npm run dev
> aula2@1.0.0 dev
> ts-node-dev ./src/index
[INFO] 21:20:01 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.1, typescript ver. 5.1.6)
Rodando na porta 3001
```

Usando o programa ts-node-dev, a cada modificação no código o servidor será reinicializado

Função call-back do método listen

Digite a URL http://localhost:3001 no navegador para testar o serviço. A mensagem exibida no navegador ocorre pelo fato de não termos definido uma rota para a URL raiz.



IV. Definição de rotas

Uma rota define um caminho específico no servidor para o qual as solicitações dos clientes são direcionadas e como essas solicitações são tratadas. O navegador ou app de celular são exemplos de clientes.

A definição de uma rota é formada pela combinação da URL com o método HTTP:

- URL: é o caminho que os clientes usam para acessar um recurso específico no servidor. Ela geralmente começa com o domínio do servidor e é seguida por uma série de caminhos que indicam a localização do recurso desejado. A URL `http://localhost:3001/teste` representa um recurso no servidor que é identificado pelo caminho `/teste`;
- Método HTTP: é uma ação que o cliente deseja realizar no recurso identificado pela URL. Os métodos mais comuns são GET, POST, PUT e DELETE, mas existem outros métodos, tais como, HEAD, PATCH, OPTIONS etc.:
  - GET: usado para solicitar dados do servidor. Como exemplo, fazer uma operação que resulta em um select numa tabela do BD;
  - POST: usado para enviar dados ao servidor para serem processados. Como exemplo, fazer uma operação que resulta em um insert numa tabela do BD;
  - PUT: usado para atualizar um recurso existente no servidor. Como exemplo, fazer uma operação que resulta em um update numa tabela do BD;
  - DELETE: usado para excluir um recurso no servidor. Como exemplo, fazer uma operação que resulta em um delete numa tabela do BD.

Ao combinar a URL e o método HTTP, obtemos uma rota completa que define a ação que o cliente deseja realizar em um recurso específico no servidor.

O Express possui os métodos `get`, `post`, `put` e `delete` para lidar, respectivamente, com os métodos HTTP GET, POST, PUT e DELETE. No código a seguir são definidas as seguintes rotas:

- Método HTTP GET para a raiz: ao encontrar essa rota será executado a seguinte função call-back:

```
(req, res) => res.send("Método HTTP GET")
```

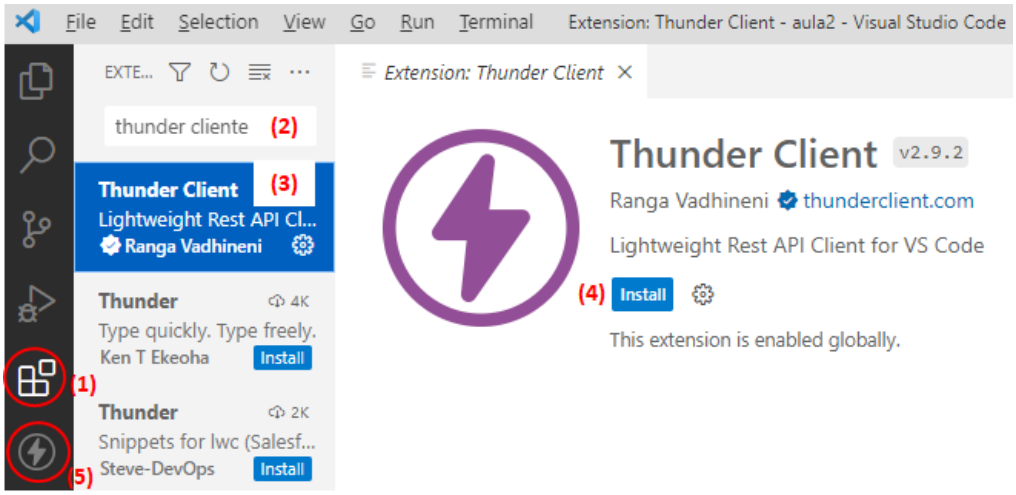
  O servidor web chamará a função call-back passando os objetos `Request` e `Response` (assim como ilustrado na Figura 1). Nesse exemplo, esses objetos serão colocados, respectivamente, nas variáveis `req` e `res`. O objeto Response possui as operações para enviar o resultado para o cliente (navegador), aqui estamos usando o método send para retornar o texto `"Método HTTP GET"`.
- Método HTTP GET para o caminho `/teste`;

- Método HTTP POST para o caminho /teste;
- Método HTTP PUT para o caminho /teste;
- Método HTTP DELETE para o caminho /teste.

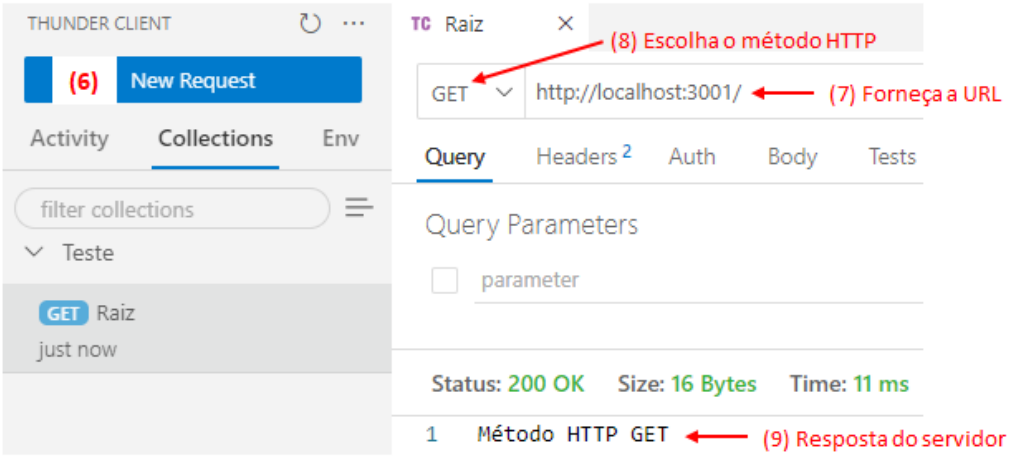
```
import express from "express";
import dotenv from "dotenv";
dotenv.config();
// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
// cria o servidor e coloca na variável app
const app = express();
// para aceitar parâmetros no formato JSON
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});
// define a rota para a raiz GET /
app.get("/", (req, res) => res.send("Método HTTP GET"))
// define a rota para GET /teste
app.get("/teste", (req, res) => res.send("Método HTTP GET"));
// define a rota para POST /teste
app.post("/teste", (req, res) => res.send("Método HTTP POST"));
// define a rota para PUT /teste
app.put("/teste", (req, res) => res.send("Método HTTP PUT"));
// define a rota para DELETE /teste
app.delete("/teste", (req, res) => res.send("Método HTTP DELETE"));
```

Observação: no navegador você consegue testar apenas as rotas que usam o método HTTP GET. Para testar as demais rotas sugere-se instalar a extensão Thunder Client no VS Code.

Passos para instalar a extensão Thunder Client:



Passos para criar uma requisição no Thunder Client:



Podemos enviar dados para o servidor de duas formas:

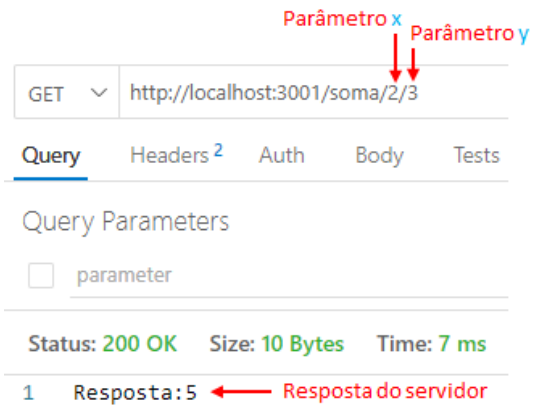
1. Pelos parâmetros da URL: observe que os parâmetros são definidos na URL usando dois pontos antes do nome do parâmetro:

```
app.get("/soma/:x/:y", (req, res) => {
    //os parâmetros da requisição são colocados na propriedade params
    const {x,y} = req.params;
    //os parâmetros são recebidos como string
    const xx = parseInt(x);
    const yy = parseInt(y);
    res.send("Resposta:" + (xx + yy));
});
```

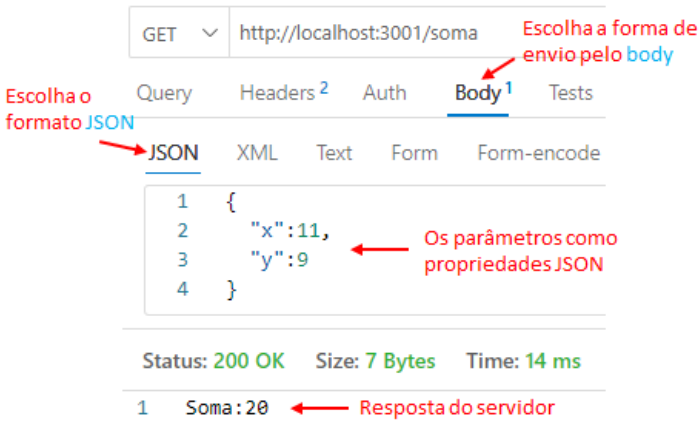
2. Pelo corpo da requisição: só conseguiremos receber parâmetros do tipo JSON se o servidor tiver sido configurado usando a instrução app.use(express.json()):

```
app.get("/soma", (req, res) => {
    //os parâmetros da requisição são colocados na propriedade body
    const {x,y} = req.body;
    res.send("Soma:" + (x + y));
});
```

Parâmetros pela URL da requisição:



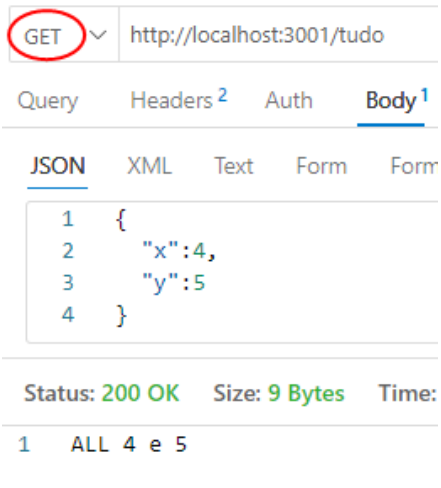
Parâmetros pelo corpo da requisição:



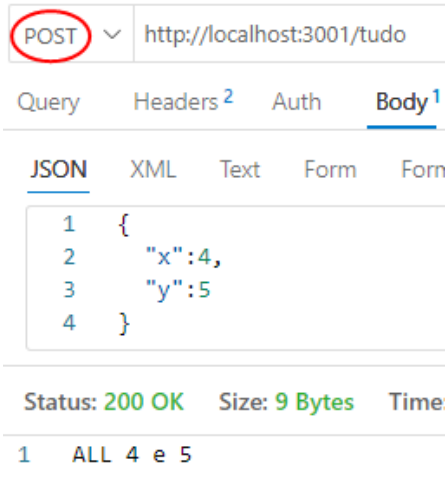
O método `all` é usado para mapear rotas oriundas de qualquer método HTTP (GET, POST etc.):

```
// o método all aceita requisições oriundas de qualquer método HTTP
app.all('/tudo', function (req, res){
    let { x, y } = req.body;
    res.send(`ALL ${x} e ${y}`);
});
```

Requisição usando HTTP GET:

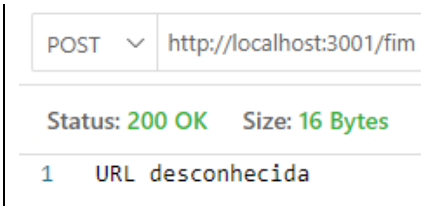


Requisição usando HTTP POST:



O método `use` é usado para receber qualquer rota não mapeada. Recomenda-se colocar ele no final, pois as rotas são analisadas de cima para baixo no arquivo.

```
//aceita qualquer método HTTP e URL
app.use(function(req, res){
    res.send('URL desconhecida');
});
```



Observações:

- Podemos definir quantas rotas quisermos, porém não podem existir duas rotas mapeadas para a mesma URL + método HTTP;
- Podemos criar uma estrutura como se fossem pastas na definição da URL, por exemplo, /cadastro/cliente, mas na prática não existem pastas.

V. Rotas para arquivo estático

Um arquivo estático é aquele arquivo que será enviado para o cliente da forma como ele se encontra, isto é, ele não será executado no servidor. HTML, CSS, PDF, TXT e imagens são alguns exemplos de formatos de arquivos estáticos.

A função `express.static` (http://expressjs.com/en/5x/api.html#express.static) é usada para servir arquivos estáticos.

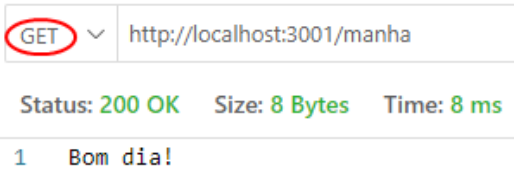
Para testar, crie a pasta `public` (pode ser qualquer nome de pasta, porém usa-se o nome public quando a pasta contém arquivos que serão entregues para o cliente) na raiz do projeto e crie os arquivos `dia.txt` e `noite.txt` na pasta `public`. Coloque os textos Bom dia! e Boa noite! nos arquivos `dia.txt` e `noite.txt`.

A função `express.static` recebe o caminho para o arquivo ou pasta a ser roteado. No 1º exemplo a seguir foi criada uma rota para o arquivo `dia.txt` usando o caminho /manha e no 2º exemplo foi criada uma rota para a pasta `public` usando o caminho /arquivo seguido pelo nome do arquivo que estiver dentro da pasta `public`.

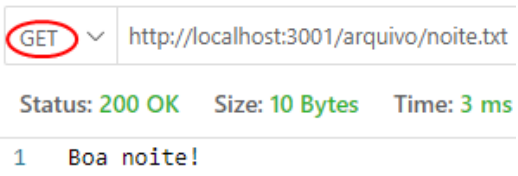
```
// rota para um arquivo específico
app.use("/manha", express.static('public/dia.txt'));

// rota para a pasta public
app.use("/arquivo", express.static('public'));
```

Requisição HTTP GET e caminho /manha será roteada para o arquivo `public/dia.txt`.



Requisição HTTP GET e caminho /arquivo/noite.txt será roteada para o arquivo `public/noite.txt`.



O Express irá automaticamente mapear as URLs que correspondem aos arquivos na pasta public e servirá o arquivo estático correspondente como resposta.

VI. Hierarquia de rotas

Em aplicações web complexas, a hierarquia de rotas é uma abordagem que permite organizar e estruturar as rotas de forma mais eficiente e modular. Podemos dividir o roteamento em diferentes arquivos e diretórios, tornando o código mais organizado, fácil de manter e escalável.

No Express a hierarquia de rotas é criada usando o conceito de roteadores (Router).

Como exemplo foi criado o projeto com a estrutura mostrada ao lado. Os recursos a serem roteados foram colocados nos métodos das classes que estão nos arquivos Matematica.ts (Figura 2) e Texto.ts (Figura 3).

Observe que os métodos dessas classes recebem os objetos Request e Response criados pelo servidor web e retornam promises.

Nesta organização os métodos controladores não possuem a responsabilidade de criar as rotas.

```
import { Request, Response } from 'express';

class Matematica {
    public async somar(req:Request,res:Response): Promise<Response>{
        let {x,y} = req.body;
        const r = parseFloat(x) + parseFloat(y);
        if( isNaN(r)){
            return res.json({error:"Parâmetros incorretos"});
        }
        return res.json({r});
    }

    public async subtrair(req:Request,res:Response): Promise<Response>{
        let {x,y} = req.body;
        const r = parseFloat(x) - parseFloat(y);
        if( isNaN(r)){
            return res.json({error:"Parâmetros incorretos"});
        }
        return res.json({r});
    }
}

export default new Matematica(); // exporta o objeto do tipo de dado Matematica
```

Figura 2 – Código do arquivo src/controllers/Matematica.ts.

```
import { Request, Response } from 'express';

class Texto {
    public async concatenar(req: Request, res: Response): Promise<Response> {
        let { x, y } = req.body;
        if (x === undefined || y === undefined) {
            return res.status(400).send("Parâmetros incorretos");
        }
        const r = x + y;
        return res.json({ r });
    }

    public async inverter(req: Request, res: Response): Promise<Response> {
        let { entrada } = req.body;
        if (entrada === undefined) {
            return res.status(400).send("Parâmetro incorreto");
        }
        const r = entrada.split('').reverse().join('');
        return res.json({ r });
    }
}

export default new Texto(); // exporta o objeto do tipo de dado Texto
```

Figura 3 – Código do arquivo src/controllers/Texto.ts.

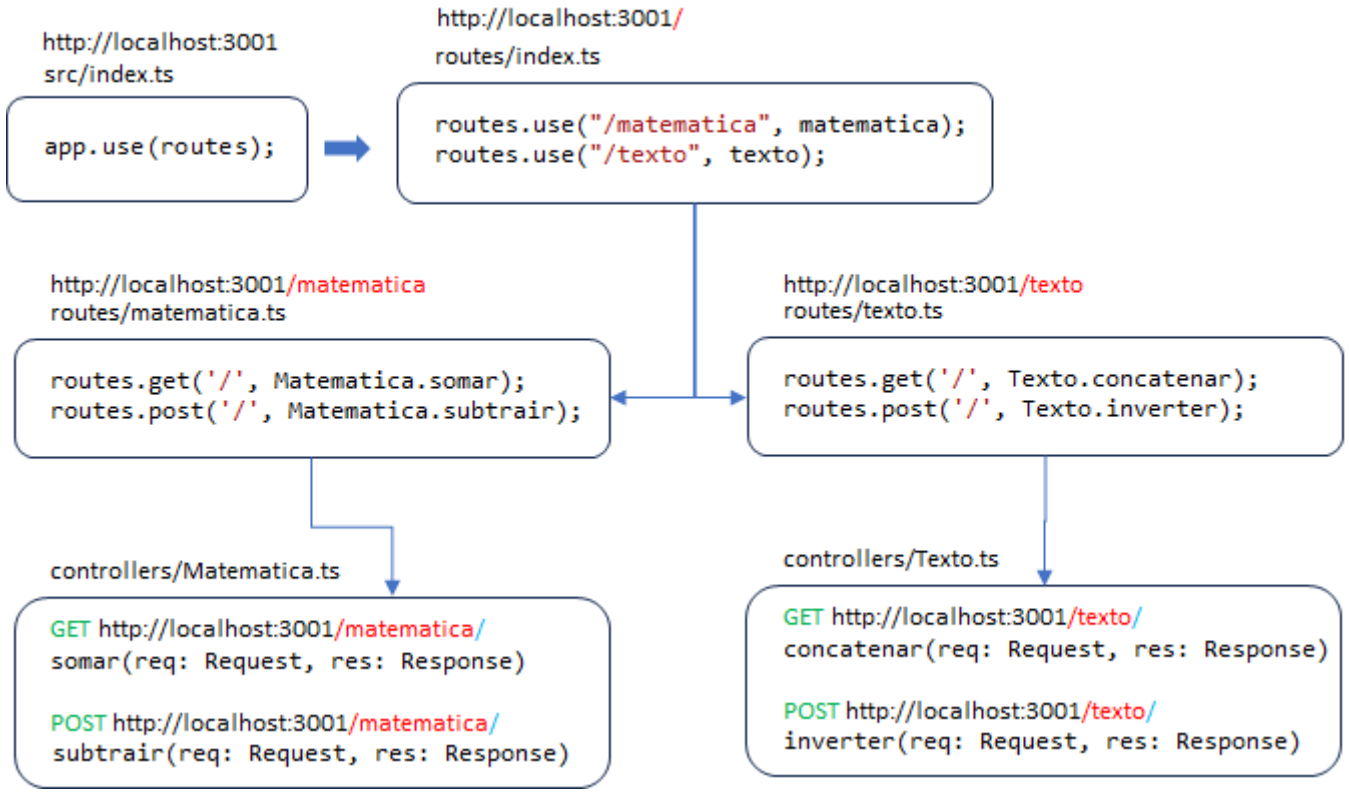
No Express a hierarquia de rotas é criada usando o conceito de roteadores (Router). Nos arquivos da Figura 4 e Figura 5 foram criados objetos do tipo Router para definir essas rotas

```
const routes = Router();
```

as rotas são definidas usando o objeto que está na variável `routes`. Cabe aos métodos `somar` e `subtrair` processar as requisições

```
routes.get('/', Matematica.somar);
routes.post('/', Matematica.subtrair);
```

As rotas exportadas nos arquivos routes/matematica.ts e routes/texto.ts são importadas e exportadas pelo arquivo routes/index.ts (Figura 6). O fluxo a seguir mostra a sequência usada para compor a hierarquia de rotas, veja que a rota inicia no arquivo src/index.ts (Figura 7).




```
import { Router } from "express";
import Matematica from "../controllers/Matematica";

const routes = Router();

routes.get('/', Matematica.somar);
routes.post('/', Matematica.subtrair);

export default routes;
```

Figura 4 – Código do arquivo src/routes/matematica.ts.

```
import { Router } from "express";
import Texto from "../controllers/Texto";

const routes = Router();

routes.get('/', Texto.concatenar);
routes.post('/', Texto.inverter);

export default routes;
```

Figura 5 – Código do arquivo src/routes/texto.ts.

```
import { Router, Request, Response } from "express";
import matematica from './matematica';
import texto from './texto';

const routes = Router();

routes.use("/matematica", matematica);
routes.use("/texto", texto);

//aceita qualquer método HTTP ou URL
routes.use( (req:Request,res:Response) => res.json({error:"Requisição desconhecida"}) )

export default routes;
```

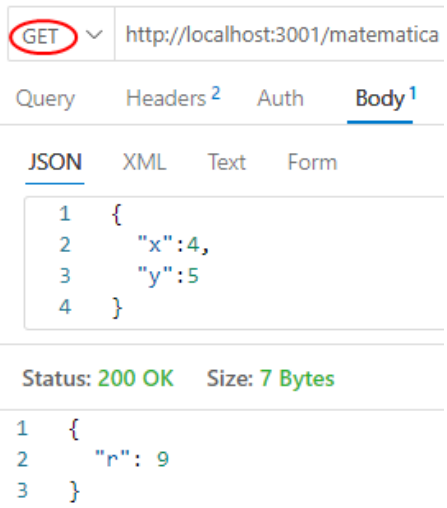
Figura 6 – Código do arquivo src/routes/index.ts.

```
import express from "express";
import dotenv from "dotenv";
import routes from "./routes";
dotenv.config();
// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
// cria o servidor e coloca na variável app
const app = express();
// para aceitar parâmetros no formato JSON
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});
// define a rota para o pacote /routes
app.use(routes);
```

Figura 7 – Código do arquivo src/index.ts.

A seguir tem-se exemplos de uso das rotas para o caminho /matematica:

O método somar da classe Matematica foi mapeado para a rota HTTP GET e caminho /matematica:



O método subtrair da classe Matematica foi mapeado para a rota HTTP POST e caminho /matematica:

Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercício 1 - https://youtu.be/u-BW-orHx5E

Exercício 2 - https://youtu.be/kEd_JuM0DOY

Exercício 3 - https://youtu.be/18_S2y-r2xU

Exercício 4 - https://youtu.be/FroFiRS1WOw

Exercício 5 - https://youtu.be/8y84F1c6z5E

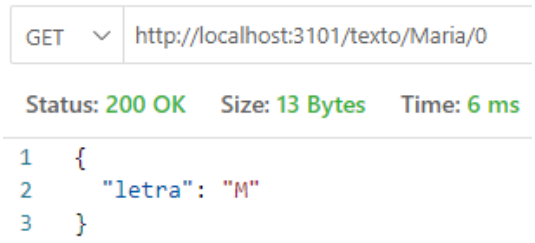
Exercício 6 - https://youtu.be/CAHhM1N_VmM

Exercício 7 - https://youtu.be/fqjXLnT8ems

Exercício 1 – Criar um servidor Express com os seguintes requisitos:

- Responder na porta 3101;
- A porta 3101 deverá ser definida no arquivo .env;
- Criar uma rota para o caminho /texto/nome/índice usando o método HTTP GET. O nome e índice são parâmetros. A rota deverá responder com a letra que está na posição do índice fornecido, por exemplo, /texto/Maria/0 retornará M, pelo fato da letra M estar na posição 0 da string Maria;

Exemplo de resposta no Thunder Client:

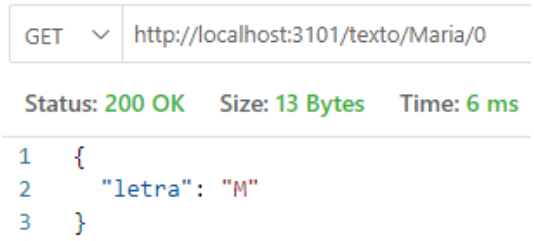


- A resposta deverá ser no formato JSON. Observação: um objeto JSON pode ser enviado usando os métodos send ou json do objeto Response da solicitação criada pelo express. O método res.json() configura automaticamente o cabeçalho Content-Type para application/json, indicando que a resposta é um objeto JSON. Já o método res.send() detecta automaticamente o tipo de dado enviado e configura o cabeçalho Content-Type de acordo.

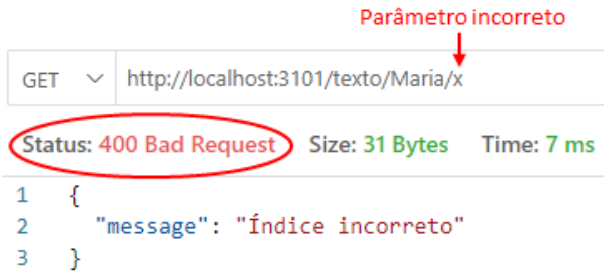
Exercício 2 – Alterar o código do Exercício 1 para responder com o código de status 400 no caso de o usuário não fornecer o índice na requisição.
Dica: o use o método status do objeto Response para setar o código de status da mensagem de retorno.

Observação: o código de status 400 é parte da classe de códigos de erro 400 a 499, que indica erros causados pelo cliente. Ele é usado quando o servidor não pode ou não irá processar a requisição devido a uma sintaxe inválida do lado do cliente. Em outras palavras, o servidor não conseguiu entender ou processar a requisição porque os parâmetros ou a estrutura da requisição enviada pelo cliente estavam incorretos ou incompreensíveis.

Exemplo de sucesso:



Exemplo de falha:

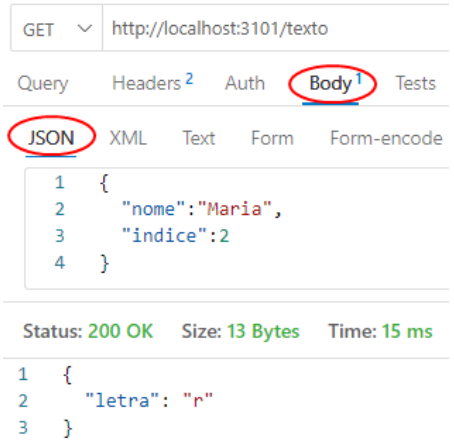


Exercício 3 – Adicionar no código do Exercício 1 uma rota para o caminho /texto e método HTTP GET. Os parâmetros nome e índice deverão ser passados pelo corpo da requisição no formato de objeto JSON.

Observação: adicione a instrução a seguir para que o servidor tenha a capacidade de manusear requisições com o body no formato JSON:

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

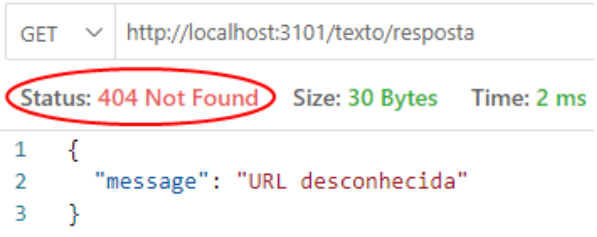
Exemplo de sucesso:



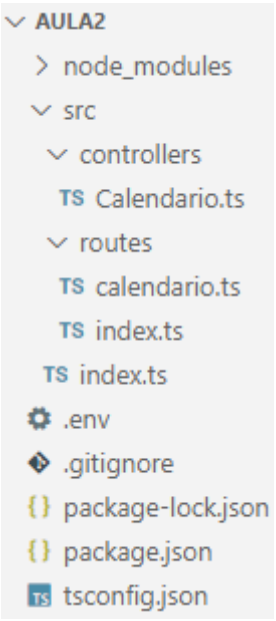
Exercício 4 – Adicionar no código do Exercício 1 a capacidade de manusear rotas desconhecidas. A resposta deverá ser no formato JSON e usar o código de status 404.

Observação: o código de status 404 indica que a rota solicitada pelo cliente não existe no servidor.

Exemplo de resposta:



Exercício 5 – Criar um projeto Node com a estrutura mostrada ao lado. O servidor deverá estar na porta 3105.

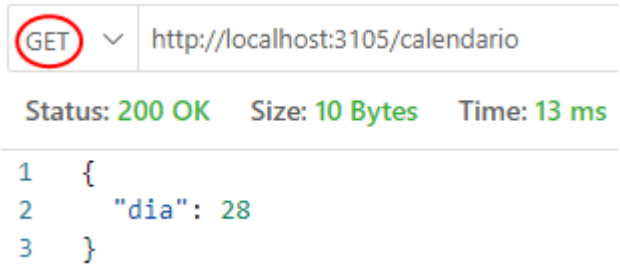


O arquivo Calendario.ts possui uma classe com as seguintes operações:

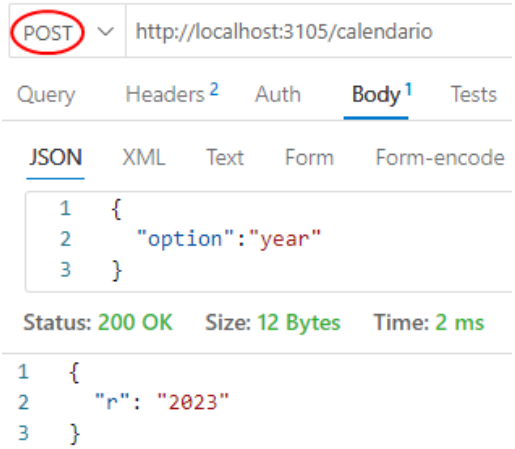
```
dayOfMonth (req:Request,res:Response) {
    const dia = (new Date()).getDate();
    return res.json({dia});
}

date(req:Request,res:Response) {
    let {option} = req.body;
    let r = "Parâmetro inválido";
    if( option === 'day' ){
        r = (new Date()).getDate() + "";
    }
    else if( option === 'month' ){
        r = ((new Date()).getMonth() + 1) + "";
    }
    else if( option === 'year' ){
        r = (new Date()).getFullYear() +"";
    }
    return res.json({r});
}
```

O método dayOfMonth responderá a seguinte rota:



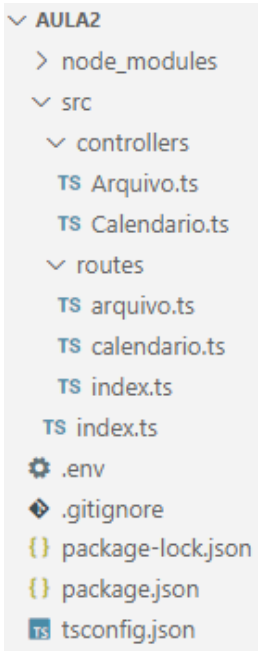
O método date responderá a seguinte rota:



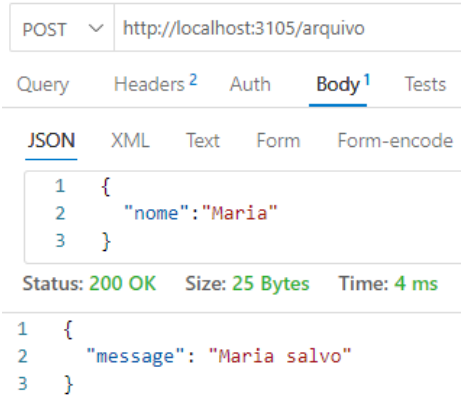
Exercício 6 – Adicionar no projeto do Exercício 5 o arquivo controllers/Arquivo.ts e colocar nele a classe a seguir. Na classe Arquivo existe os seguintes métodos:

- write: criar o arquivo nomes.txt na raiz do projeto e adicionar um nome no arquivo nomes.txt;
- append: adicionar um nome no arquivo nomes.txt;
- read: listar o conteúdo do arquivo nomes.txt.

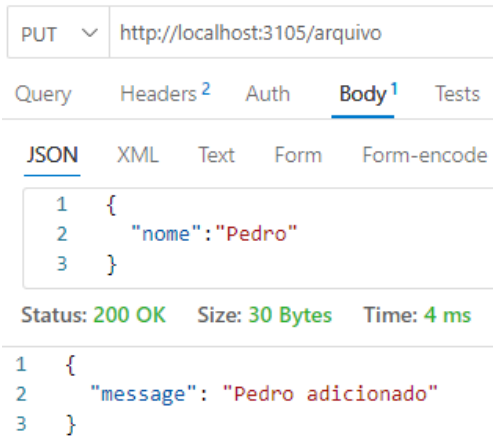
Programar as seguintes rotas no projeto usando hierarquia de rotas:



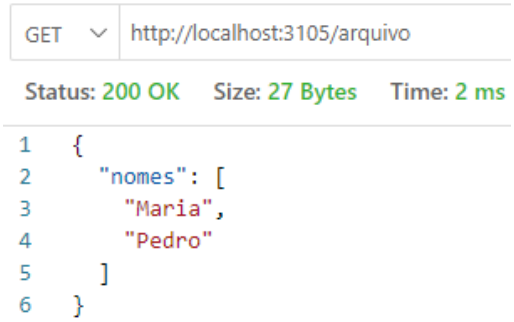
- Rota HTTP POST para o caminho /arquivo: criar o arquivo e colocar nele o nome passado como parâmetro pelo corpo da requisição:



- Rota HTTP PUT para o caminho /arquivo: adicionar no final do arquivo o nome passado como parâmetro pelo corpo da requisição:



- Rota HTTP GET para o caminho /arquivo: listar o conteúdo do arquivo:



Observação: será necessário instalar o pacote fs-extra (https://www.npmjs.com/package/fs-extra) para podermos manipular arquivos. Instale também os tipos do fs-extra como dependência de desenvolvimento:

```
npm i fs-extra
npm i -D @types/fs-extra
```

O arquivo controllers/Arquivo.ts possui a seguinte classe:

```
import { Request, Response } from 'express';
import fs from "fs-extra";

class Arquivo {
    static filename: string = "./nomes.txt";

    async write(req: Request, res: Response) {
        const { nome } = req.body;
        if (nome && nome !== "") {
            fs.writeFile(Arquivo.filename, nome)
            .then(() => res.json({ message: `${nome} salvo` }))
            .catch((e: any) => res.status(400).json({message: "Problemas ao escrever" }));
        }
        else {
            res.status(400).json({ message: "Nome não fornecido" });
        }
    }

    async read(req: Request, res: Response) {
        fs.readFile(Arquivo.filename)
        .then((data) => res.send({ nomes: data.toString().split("\n") }))
        .catch((e: any) => res.status(400).json({ message: e.message }));
    }

    async append(req: Request, res: Response) {
        const { nome } = req.body;
        if (nome && nome !== "") {
            fs.appendFile(Arquivo.filename, "\n" + nome)
            .then(() => res.json({ message: `${nome} adicionado` }))
            .catch((e: any) => res.status(400).json({ message: e.message }));
        }
        else {
            res.status(400).json({ message: "Nome não fornecido" });
        }
    }
}

export default new Arquivo();
```

Exercício 7 – Adicionar no projeto do Exercício 6 o arquivo controllers/Loteria.ts e colocar nele o código a seguir. No arquivo existem as seguintes funções exportadas:

- mega: faz uma requisição no serviço de loterias da Caixa e retorna um JSON com os dados do último sorteio da Mega-sena;
- quina: faz uma requisição no serviço de loterias da Caixa e retorna um JSON com os dados do último sorteio da Quina.

Programar as seguintes rotas no projeto usando hierarquia de rotas:

- Rota HTTP GET para o caminho /loteria/mega:

GET http://localhost:3105/loteria/mega Send

Status: 200 OK Size: 348 Bytes Time: 793 ms

Response Headers 6 Cookies Results Docs { }


```
{
  "acumulado": true,
  "concursoEspecial": false,
  "dataApuracao": "27/07/2023",
  "dataPorExtenso": "Quinta-feira, 27 de Julho de 2023",
  "dataProximoConcurso": "29/07/2023",
  "dezenas": [
    "05",
    "07",
    "22",
    "23",
    "41",
    "59"
  ],
  "numeroDoConcurso": 2615,
  "quantidadeGanhadores": 0,
  "tipoPublicacao": 3,
  "tipoJogo": "MEGA_SENA",
  "valorEstimadoProximoConcurso": 40000000,
  "valorPremio": 0
}
```


- Rota HTTP GET para o caminho /loteria/quina:

GET http://localhost:3105/loteria/quina Send

Status: 200 OK Size: 337 Bytes Time: 654 ms

Response Headers 6 Cookies Results Docs { }


```
{
  "acumulado": true,
  "concursoEspecial": false,
  "dataApuracao": "28/07/2023",
  "dataPorExtenso": "Sexta-feira, 28 de Julho de 2023",
  "dataProximoConcurso": "29/07/2023",
  "dezenas": [
    "21",
    "32",
    "37",
    "38",
    "49"
  ],
  "numeroDoConcurso": 6201,
  "quantidadeGanhadores": 0,
  "tipoPublicacao": 3,
  "tipoJogo": "QUINA",
  "valorEstimadoProximoConcurso": 1500000,
  "valorPremio": 0
}
```

Observação: será necessário instalar o pacote axios (https://www.npmjs.com/package/axios) para podermos fazer requisições HTTP a partir do Node:

```
npm i axios
```

Código do arquivo controllers/Loteria.ts:

```
import axios, { AxiosInstance } from 'axios';
import { Request, Response } from 'express';
import https from "https";

const api: AxiosInstance = axios.create({
    baseURL: "https://servicebus2.caixa.gov.br/portaldeloterias/api/home/ultimos-resultados",
    headers: {
        'Content-Type': 'application/json',
    },
    // permitirá que o Axios ignore temporariamente a verificação do certificado SSL
    httpsAgent: new https.Agent({ rejectUnauthorized: false })
});

export async function mega(req: Request, res: Response) {
    //desestrutura o objeto megasena
    const { data:{megasena} } = await api.get("/");
    res.send(megasena);
}

export async function quina(req: Request, res: Response) {
    //desestrutura o objeto quina
    const { data:{quina} } = await api.get("/");
    res.send(quina);
}
```