Objetivos:

I. Estrutura de uma aplicação do lado servidor;
II. Node.js e Express;
III. Criar servidor Node.js com Express usando TypeScript;
IV. Arquivo principal do servidor: src/index.ts;
V. Definição de rotas;
VI. Envio de dados para o servidor;
VII. Servir arquivos estáticos;
VIII. Hierarquia de rotas.

I. Estrutura de uma aplicação do lado servidor

Os navegadores se comunicam com servidores web usando o protocolo HTTP (Hypertext Transfer Protocol). Servidores como Apache Tomcat, Apache com PHP e Node.js com Express são exemplos comuns para rodar aplicações no back-end.

A Figura 1 representa uma requisição HTTP, que envolve dois principais objetos:

- Request (Requisição): contém informações enviadas pelo cliente, como:
  - URL (Uniform Resource Locator): define o endereço do recurso solicitado;
  - Método HTTP: define a ação a ser realizada (ex.: GET, POST, PUT, DELETE);
  - Parâmetros e corpo da requisição: incluem dados adicionais necessários para processar a solicitação.

  Exemplo de URL com parâmetros:

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

  nome e idade são parâmetros passados para o servidor.
- Response (Resposta): contém a mensagem de retorno do servidor, que inclui:
  - Código de status HTTP: indica o resultado da requisição (ex.: 200 OK, 401 Unauthorized, 404 Not Found);
  - Corpo da resposta: pode conter dados, mensagens ou arquivos.

Aplicações dinâmicas x estáticas:

- Aplicações dinâmicas: o conteúdo do objeto Response é gerado de acordo com o resultado processado pelo programa (ex.: consultas a um BD, cálculos ou lógica específica);
- Aplicações estáticas: sempre retornam o mesmo conteúdo, geralmente arquivos fixos, como HTML, CSS ou imagens.

Códigos de status HTTP: as respostas HTTP utilizam códigos de status de três dígitos para informar o resultado de uma requisição. Eles são divididos em cinco classes principais:

- 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. Compreender esses códigos é essencial para interpretar corretamente as respostas do servidor e lidar com diferentes cenários durante a comunicação entre cliente e servidor. Para mais detalhes https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status.

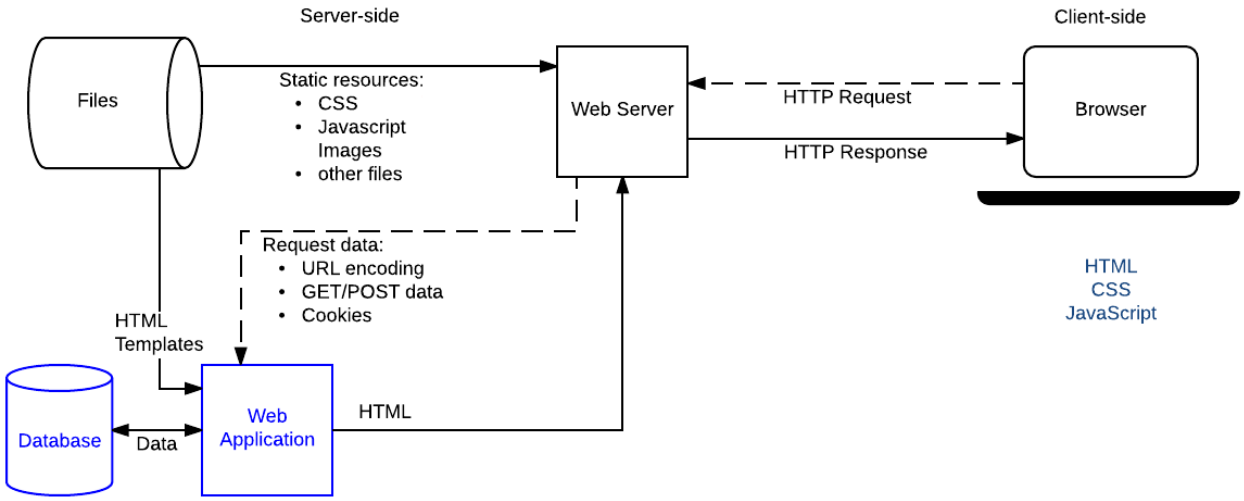


Figura 1 – Representação de uma requisição HTTP.

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

II. Node.js e Express

Node.js é um ambiente de execução (*runtime*) open-source e multiplataforma que permite a execução de código JS no lado do servidor. Ele foi projetado para ser usado fora do contexto de um navegador, funcionando diretamente em um computador ou servidor. Dessa forma, o Node.js omite APIs específicas do navegador (como DOM e window) e adiciona suporte para APIs do sistema operacional, incluindo bibliotecas HTTP e manipulação de arquivos. Para mais detalhes https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

É possível criar um servidor web usando apenas o módulo HTTP padrão do Node.js. Contudo, tarefas comuns no desenvolvimento web não são diretamente suportadas pelo módulo nativo, como:

- Manipulação detalhada de requisições HTTP (GET, POST, PUT, DELETE);
- Criação de rotas dinâmicas e específicas para URLs;
- Servir arquivos estáticos (imagens, CSS, JS);
- Uso de modelos (templates) para geração dinâmica de conteúdo

Para resolver essas limitações, os desenvolvedores têm duas opções:

- Escrever manualmente todo o código necessário para essas funcionalidades;
- Utilizar bibliotecas ou frameworks especializados, como o Express.

O Express é um framework minimalista e amplamente utilizado para desenvolvimento de aplicações web e APIs com Node.js. Ele fornece uma abstração de alto nível para simplificar tarefas comuns, como:

- Roteamento: definição de caminhos e regras para URLs específicas;
- Tratamento de requisições e respostas HTTP: processamento de dados recebidos e envio de respostas adequadas;

- Middlewares: funções intermediárias que modificam requisições ou respostas antes de chegarem ao destino final;
- Servir arquivos estáticos: disponibilizar arquivos públicos como CSS, JS e imagens;
- Suporte para diferentes formatos de resposta: JSON, HTML, texto simples, entre outros.

III. Criar servidor Node.js com Express usando TypeScript

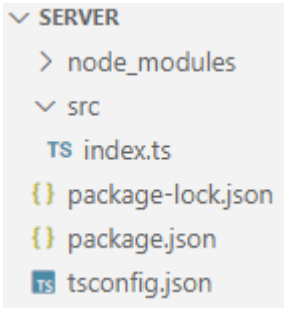
A seguir, tem-se os passos para configurar um servidor Node.js utilizando o framework Express e TypeScript (https://www.npmjs.com/package/express):

1. Criar uma pasta para o projeto
   - Crie uma pasta no seu computador. Neste exemplo, ela se chamará server, mas pode ser qualquer outro nome, desde que não contenha caracteres especiais ou espaços;
2. Inicializar o projeto Node.js
   - Abra a pasta no terminal (CMD ou terminal integrado do VS Code);
   - Execute o comando `npm init -y`. O parâmetro `-y` evita perguntas interativas e cria automaticamente o arquivo package.json com configurações padrão;
3. Instalar o Express
   - `npm i express`
   - Esse comando cria a pasta node_modules e adiciona a dependência no package.json;
4. Instalar o TypeScript e definições de tipo para Express
   - Como será usado TypeScript, é necessário instalar o TypeScript no projeto:
     `npm i typescript -D`
   - Instale também as definições de tipo do Express:
     `npm i @types/express -D`
   - O parâmetro `-D` indica que essas dependências são para desenvolvimento (devDependencies);
5. Configurar o TypeScript
   - Inicialize o arquivo de configuração do TypeScript
     `npx tsc --init`
   - Edite o arquivo tsconfig.json para garantir que o código será compilado corretamente. Certifique-se de configurar as seguintes opções:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "CommonJS",
    "rootDir": "./src",
    "outDir": "./dist",
    "esModuleInterop": true,
    "strict": true
```

```
    }
}
```

6. Criar a estrutura do projeto (assim como é mostrado ao lado)
   - No VS Code, crie uma pasta chamada src para armazenar os arquivos TS;
   - Dentro da pasta src, crie um arquivo chamado index.ts;



7. Abra o projeto no VS Code e crie a pasta src e o arquivo index.ts, assim como é mostrado ao lado. Constitui boa prática manter o código na pasta src. Como teste coloque a seguinte instrução no corpo do arquivo index.ts:

```
console.log("oi");
```

8. Instalar pacotes para rodar o servidor com TS
   - Para executar diretamente arquivos TS sem precisar compilar manualmente para JS, utilizaremos o ts-node e o ts-node-dev:

     ```
     npm i ts-node ts-node-dev -D
     ```

   - ts-node: permite executar arquivos TS diretamente no Node.js;
   - ts-node-dev: monitora alterações nos arquivos .ts e reinicia automaticamente o servidor. Amplamente usando durante a codificação do projeto;
9. Adicionar scripts no package.json
   - No arquivo package.json, adicione os seguintes scripts para facilitar a execução do projeto:

```
"scripts": {
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
  "build": "tsc",
  "start": "node dist/index.js"
},
```

   - dev: executa o servidor com ts-node-dev, monitorando alterações. A seguir tem-se o comando executado no terminal do VS Code:

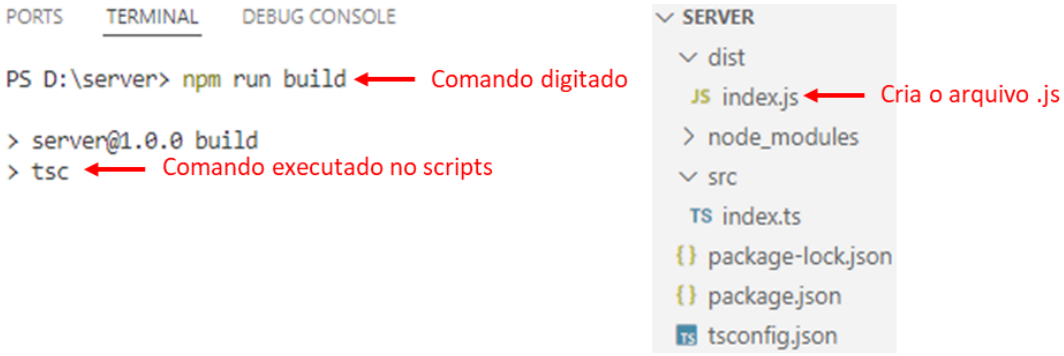
```
PORTS   TERMINAL   DEBUG CONSOLE   OUTPUT   PROBLEMS

PS D:\server> npm run dev  <-- Comando digitado

> server@1.0.0 dev
> ts-node-dev --respawn --transpile-only src/index.ts  <-- Comando executado no scripts

[INFO] 14:29:33 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typescript ver. 5.7.2)
oi  <-- Resultado da execução do arquivo src/index.ts
```

Explicação das flags do ts-node-dev:

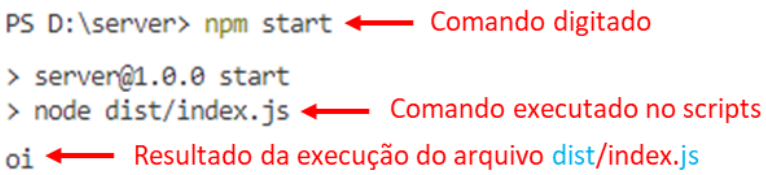
`--respawn`: garante que o processo seja reiniciado corretamente após alterações;

`--transpile-only`: compila apenas os arquivos necessários, acelerando o processo.

- `build`: compila os arquivos TS (.ts) para JS (.js) usando tsc. Os arquivos JS serão gerados na pasta `dist`, conforme especificado na propriedade `"outDir": "./dist"` do arquivo tsconfig.json.



- `start`: para rodar a versão compilada que na pasta `dist`. Esse comando executa o arquivo dist/index.js.



10. Crie o arquivo .gitignore no raiz do projeto e coloque a instrução `node_modules`.

11. Crie o arquivo .env no raiz do projeto e coloque a instrução `PORT = 3010` (não pode ter ponto e vírgula);

12. As variáveis do arquivo .env não são carregadas pelo ambiente de execução do Node. Para resolver esse problema temos de instalar o pacote dotenv (https://www.npmjs.com/package/dotenv)

```
npm i dotenv
```

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

```
{
  "name": "server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
    "build": "tsc",
    "start": "node dist/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
```

```
  "description": "",
  "dependencies": {
    "dotenv": "^16.4.7",
    "express": "^4.21.2"
  },
  "devDependencies": {
    "@types/express": "^5.0.0",
    "ts-node": "^10.9.2",
    "ts-node-dev": "^2.0.0",
    "typescript": "^5.7.2"
  }
}
```

A numeração de versões em um pacote segue geralmente o formato "major.minor.patch", conforme o padrão SemVer (Semantic Versioning):

- major (principal): alterações que quebram compatibilidade com versões anteriores;
- minor (secundária): adição de novos recursos que não afetam a compatibilidade existente;
- patch (correção): correções de bugs ou melhorias que não afetam funcionalidades existentes.

Significado do símbolo ^ no package.json: permite que o gerenciador de pacotes (npm ou yarn) atualize automaticamente as versões minor e patch, mas não altera a versão major. Exemplo: "ts-node": "^10.9.2" permite instalar versões como:

- 10.9.3 (patch atualizado);
- 10.10.0 (minor atualizado);
- No entanto, não permitirá instalar uma versão 11.0.0, pois isso representaria uma mudança na versão major e, potencialmente, uma quebra de compatibilidade.

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.

IV. Arquivo principal do servidor: src/index.ts

O arquivo src/index.ts é o ponto de entrada principal do servidor Node.js com Express. Ele centraliza as configurações iniciais, incluindo middlewares essenciais, definição de rotas básicas e a configuração da porta onde o servidor será disponibilizado.

Estrutura do arquivo src/index.ts:

A seguir tem-se um exemplo de inicialização de um servidor Express com suporte a variáveis de ambiente por meio do pacote dotenv:

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

// Carrega variáveis de ambiente do arquivo .env
dotenv.config();

// Inicializa o aplicativo Express
const app = express();
const PORT = process.env.PORT || 3000;

// Middleware para permitir que o servidor interprete JSON
app.use(express.json());

// Rota padrão
app.get('/', (req: Request, res: Response) => {
  res.send('Rota padrão');
});

// Inicializa o servidor
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

```
D:\server> npm run dev ← Comando para subir o servidor em modo de desenvolvimento
> server@1.0.0 dev
> ts-node-dev --respawn --transpile-only src/index.ts

[INFO] 12:01:12 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.2, typescript ver. 5.7.2)
Servidor rodando em http://localhost:3010 ← Rodando na porta 3010 do localhost
```

Explicação do código:

- Importação do Express:

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

  - `Request`: tipo fornecido pelo Express para representar requisições HTTP;
  - `Response`: tipo fornecido pelo Express para representar respostas HTTP.
- Carregamento das variáveis de ambiente:

```
dotenv.config();
```

- Carrega as variáveis definidas no arquivo .env e as disponibiliza no objeto global `process.env`.

- Inicialização do aplicativo Express:

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

- Cria uma instância do servidor Express, responsável por gerenciar as requisições e respostas HTTP.

- Configuração da porta:

```
const PORT = process.env.PORT || 3000;
```

- process.env.PORT: permite definir a porta através de uma variável de ambiente;
- Caso não haja uma variável definida, utiliza a porta padrão 3000.

- Middleware para JSON:

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

- Permite que o servidor interprete dados no formato JSON enviados nas requisições HTTP.

- Definição de uma rota:

```
app.get('/', (req: Request, res: Response) => {
  res.send('Rota padrão');
});
```

- Define uma rota GET para o caminho raiz /;
- O método send envia uma resposta ao cliente.

- Inicialização do servidor:

```
app.listen(PORT, () => { … });
```

- Inicia o servidor na porta definida;
- A função de callback é chamada após o servidor entrar em atividade.

> `process.env` é um objeto global no ambiente Node.js que armazena variáveis de ambiente do sistema. Ele é amplamente utilizado para configurar informações sensíveis ou dependentes do ambiente, como portas de serviço, credenciais de BD e chaves de API.

V. Definição de rotas

Uma rota define um caminho específico no servidor para o qual as solicitações dos clientes são direcionadas, bem como a forma como essas solicitações serão tratadas. Clientes comuns incluem navegadores, aplicativos móveis ou outros sistemas que fazem requisições HTTP.

Estrutura de uma rota: uma rota é formada pela combinação de dois elementos principais:

- URL: é o caminho que os clientes usam para acessar um recurso específico no servidor. Geralmente começa com o domínio (ou IP) do servidor, seguido de um caminho que aponta para um recurso específico.

- Exemplo: `http://localhost:3010/teste` representa um recurso acessado pelo caminho `/teste`.

- Método HTTP: define a ação que o cliente deseja realizar no recurso especificado pela URL. Alguns dos métodos mais comuns são:
  - GET: usado para solicitar dados do servidor. Geralmente associado a operações de leitura, como um SELECT em um BD;
  - POST: usado para enviar dados ao servidor para serem processados, como um INSERT em umBD;
  - PUT: usado para atualizar dados um recurso existente, como um UPDATE em um BD;
  - DELETE: usado para remover dados de um recurso específico, com um DELETE em um BD.

  Outros métodos HTTP menos comuns incluem HEAD, PATCH, OPTIONS, entre outros.

No Express, há métodos correspondentes para cada tipo de método HTTP: get, post, put, delete, entre outros. A seguir está um exemplo prático demonstrando a criação de rotas básicas:

```
import express, { Request, Response } from "express";
import dotenv from "dotenv";
dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

// Rota GET para a raiz "/"
app.get("/", (req: Request, res: Response) => {
  res.send("Método HTTP GET");
});

// Rota GET para "/teste"
app.get("/teste", (req: Request, res: Response) => {
  res.send("Método HTTP GET");
});

// Rota POST para "/teste"
app.post("/teste", (req: Request, res: Response) => {
  res.send("Método HTTP POST");
});

// Rota PUT para "/teste"
app.put("/teste", (req: Request, res: Response) => {
  res.send("Método HTTP PUT");
```

```
});

// Rota DELETE para "/teste"
app.delete("/teste", (req: Request, res: Response) => {
  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.

VI. Envio de dados para o servidor

Para que um cliente possa interagir com o servidor de forma eficiente, é necessário enviar dados de diferentes maneiras, dependendo do propósito e do tipo de requisição. Os principais métodos para envio de dados são:

- Parâmetros na URL (Route Parameters);
- Query Parameters;
- Body (Corpo da Requisição).

Cada método tem uma aplicação específica e pode ser usado em diferentes cenários.

Parâmetros na URL:

Os parâmetros na URL são usados para enviar informações diretamente no caminho da rota. Geralmente, são utilizados para identificar recursos específicos no servidor, como um ID ou um nome. Exemplo de uso:

```
// Rota com parâmetro dinâmico na URL
app.get("/usuario/:id", (req: Request, res: Response) => {
  const { id } = req.params;
  res.send(`Usuário com ID: ${id}`);
});
```

Explicação:

- A URL `/usuario/:id` define um parâmetro dinâmico chamado `id`;
- O valor do parâmetro pode ser acessado através de `req.params.id`.

Exemplo de requisição. O valor 123 será atribuído ao parâmetro `id`.

- GET http://localhost:3000/usuario/123

Query Parameters:

Os Query Parameters permitem enviar dados adicionais na URL por meio de pares chave=valor. Eles são frequentemente usados para filtragem, paginação ou parâmetros opcionais. Exemplo de uso:

```
// Rota com Query Parameters
app.get("/usuarios", (req: Request, res: Response) => {
  const { page, limit } = req.query;
  res.send(`Página: ${page}, Limite: ${limit}`);
```

```
});
```

Explicação:

- Os parâmetros são adicionados após o símbolo ? na URL;
- Parâmetros múltiplos são separados por &;
- Os valores podem ser acessados através de `req.query`.

Exemplo de Requisição:

- GET http://localhost:3000/usuarios?page=1&limit=10

Body (Corpo da Requisição)

O Body é usado para enviar dados estruturados no corpo da requisição, geralmente em formatos como JSON. Esse método é ideal para envio de informações mais complexas ou sensíveis, como formulários, dados de cadastro, ou arquivos. Exemplo de uso:

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

// Rota com Body
app.post("/usuario", (req: Request, res: Response) => {
  const { nome, email } = req.body;
  res.send(`Nome: ${nome}, Email: ${email}`);
});
```

Explicação:

- O middleware `express.json()` é necessário para que o servidor consiga interpretar o corpo da requisição em formato JSON;
- Os dados são acessados através de `req.body`.

Exemplo de Requisição (usando JSON). É necessário usar o Thunder Client ou algum outro recurso capaz de fazer requisições do tipo POST:

```
POST http://localhost:3000/usuario
Content-Type: application/json

{
  "nome": "João Silva",
  "email": "joao@email.com"
}
```

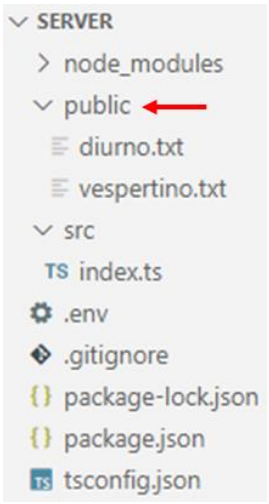
VII. Servir arquivos estáticos

Aplicações web frequentemente precisam servir arquivos estáticos, como imagens, arquivos CSS, scripts JavaScript e documentos. Esses arquivos não mudam dinamicamente com cada solicitação e são essenciais para a interface e funcionalidade no lado do cliente.

Arquivos estáticos são recursos que o servidor envia diretamente para o cliente, sem qualquer processamento adicional no backend. Exemplos comuns incluem:

- Imagens: .png, .jpg, .svg;
- Estilos: .css;
- Scripts: .js;
- Documentos: .pdf, .txt.

Esses arquivos geralmente são armazenados em uma pasta específica no servidor – aqui, utilizaremos a pasta public. Como exemplo, crie dois arquivos na pasta public:



- diurno.txt com o conteúdo *Bom dia!*;
- vespertino.txt com o conteúdo *Boa tarde!*.

Configurando o servidor para servir arquivos estáticos:

No Express, usamos o middleware `express.static` para servir arquivos estáticos.

Explicação do código a seguir:

- `express.static`: serve arquivos da pasta especificada (`public`);
- Mapeamento de rotas: a rota `/public` permite acessar os arquivos da pasta `/public` diretamente pelo navegador (ex.: http://localhost:3010/public/diurno.txt);
- `__dirname`: em /src/index.ts, o `__dirname` referencia o diretório /src. Como a pasta `public` está fora de /src, usamos `..` para subir um nível até a raiz do projeto e acessar corretamente /public.

Exemplo de código:

```
import express, { Request, Response } from 'express';
import path from 'path';
import dotenv from "dotenv";

dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

// Configura a pasta pública para arquivos estáticos
app.use('/public', express.static(path.join(__dirname, '..', 'public')));
```

```
app.get('/', (req: Request, res: Response) => {
  res.send('Rota padrão');
});

// Para rotas não localizada
app.use((req: Request, res: Response) => {
  res.status(404).send('Recurso não localizado');
});
```

Teste no navegador:

- http://localhost:3010/public/diurno.txt
- http://localhost:3010/public/vespertino.txt

Alterando o caminho público:

Se você não deseja expor diretamente a pasta /public no caminho da URL, pode usar um alias para acessar os arquivos. Por exemplo:

```
app.use('/dados', express.static(path.join(__dirname, '..', 'public')));
```

Neste caso, os arquivos podem ser acessador por:

- http://localhost:3010/dados/diurno.txt.;
- http://localhost:3010/dados/vespertino.txt.

VIII. Hierarquia de rotas

Em aplicações web, a organização das rotas desempenha um papel crucial na estrutura e manutenção do código. A hierarquia de rotas permite organizar endpoints de forma clara e escalável, facilitando o entendimento, a manutenção e a expansão do sistema.

A hierarquia de rotas é a estruturação das rotas em diferentes níveis ou camadas, organizando endpoints de acordo com seus contextos e funcionalidades. Isso ajuda a evitar conflitos, duplicação e inconsistências entre rotas.

Rotas aninhadas:

No Express, é possível criar rotas aninhadas, onde uma rota principal (ou pai) contém rotas específicas (ou filhas). Isso é útil quando várias rotas compartilham um mesmo prefixo.

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. Eles serão as funções objetivos das rotas.

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

class Matematica {
  public async somar(req: Request, res: Response): Promise<void> {
    const x = req.query.x as string;
    const y = req.query.y as string;
    const r = parseFloat(x) + parseFloat(y);
    res.json({ r });
  }

  public async subtrair(req: Request, res: Response): Promise<void> {
    const {x, y} = req.params;
    const r = parseFloat(x) - parseFloat(y);
    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 inverter(req: Request, res: Response): Promise<void> {
    const entrada = req.query.entrada as string;
    const r = entrada.split("").reverse().join("");
    res.json({ r });
  }
}

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

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

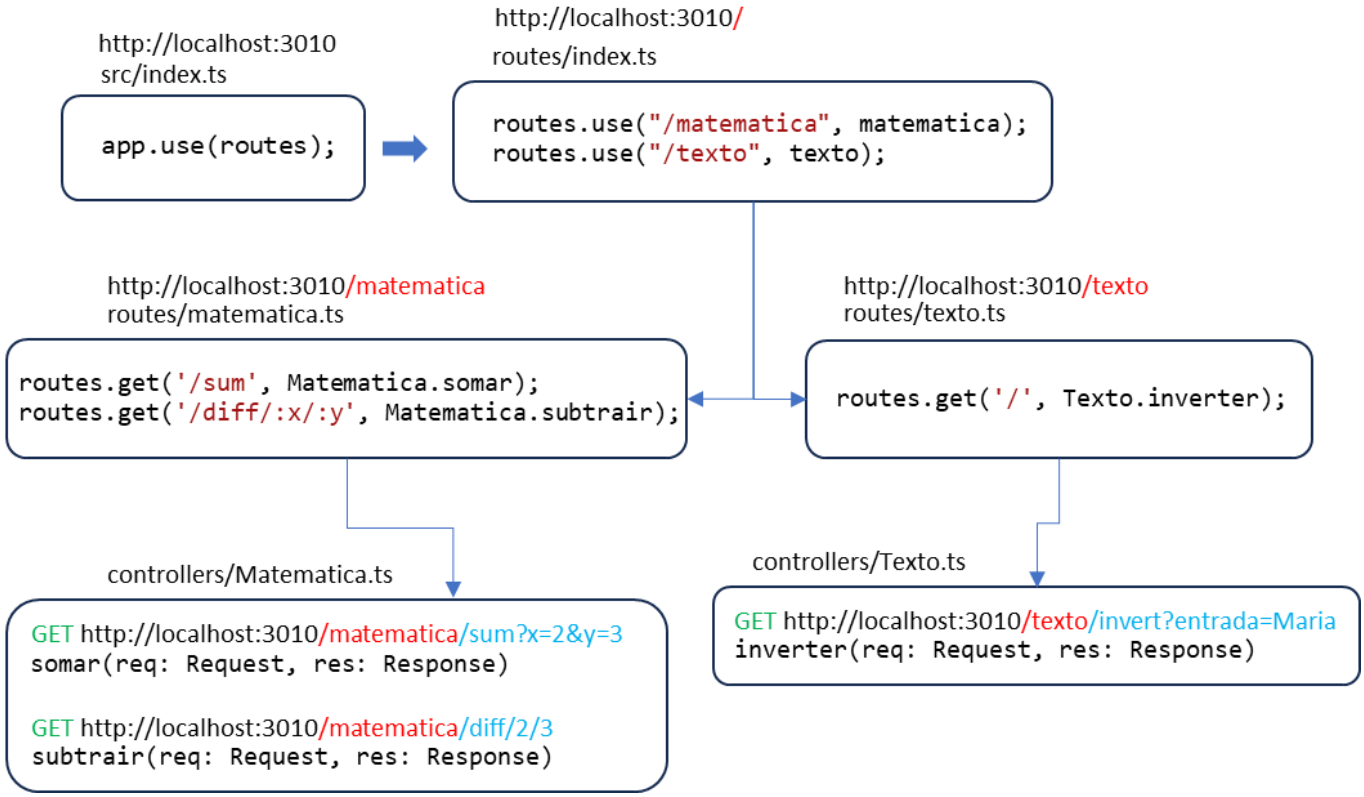
No Express, a hierarquia de rotas é criada usando o conceito de roteadores (Router). Nos arquivos da Figura 4 e Figura 5 foram criados objetos do tipo `Router` para definir essas rotas,

```
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('/sum', Matematica.somar);
routes.get('/diff/:x/:y', 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("/sum", Matematica.somar);
routes.get("/diff/:x/:y", 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('/invert', 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();

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

// define a rota para o pacote /routes
app.use(routes);
```

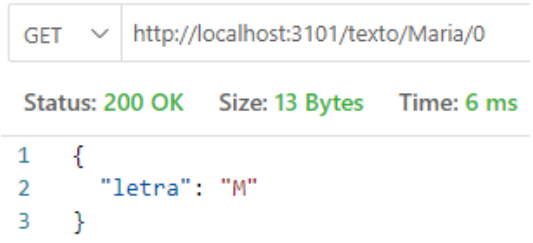
Figura 7 – Código do arquivo src/index.ts.

IX. Exercícios

Exercício 1 – Complete o código a seguir para atender aos seguintes requisitos:

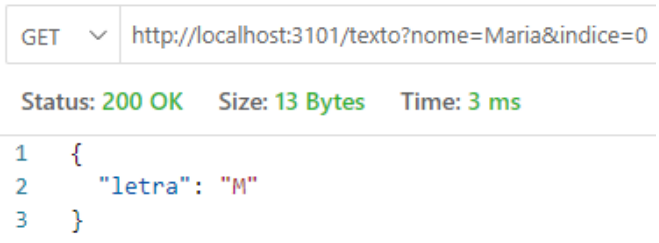
Exemplo de resposta no Thunder Client:

- O servidor deve responder na porta 3101, que deve ser configurada no arquivo .env;
- Crie uma rota utilizando o método HTTP GET que receba um nome e um índice como parâmetros na URL, por exemplo: /texto/nome/indice;
  - O nome e o índice devem ser parâmetros na rota;
  - A rota deverá retornar a letra correspondente à posição indicada pelo índice no nome fornecido;
  - Exemplo, /texto/Maria/0 deve retornar M, pois a letra M está na posição 0 da string Maria.




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

dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

Exercício 2 – Adicione ao código do Exercício 1 uma nova rota que receba o nome e índice utilizando Query Parameters em vez de parâmetros na URL.



Requisitos:

- A rota deverá utilizar o método HTTP GET;
- Os parâmetros nome e indice devem ser recebidos via query string, por exemplo: /texto?nome=Maria&indice=0

Exercício 3 – Adicione ao código do Exercício 2 uma nova rota que receba o nome e índice utilizando o corpo da requisição (request body) no formato JSON.

Requisitos:

- A rota deverá utilizar o método HTTP POST;
- Os parâmetros nome e indice devem ser enviados no corpo da requisição no formato JSON.

Dica:

- Certifique-se de que o servidor está configurado para processar o corpo da requisição em formato JSON, utilizando o middleware express.json().

Exercício 4 – Altere o código das rotas implementadas nos Exercícios 1 a 3 para tratar casos em que o índice fornecido seja inválido.

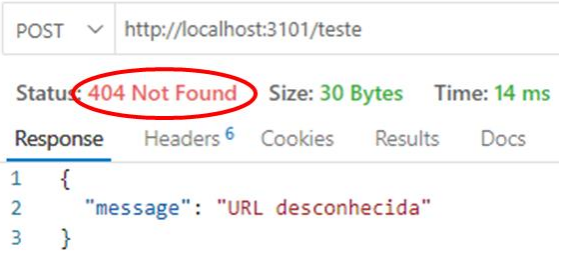
Requisitos:

- A lógica deve verificar se:
  - O índice é um número inteiro válido;
  - O índice está dentro do intervalo da string fornecida (entre 0 e o tamanho da string - 1).
- Em caso de índice inválido, as rotas devem retornar uma mensagem de erro com Status code: 400 (Bad Request).

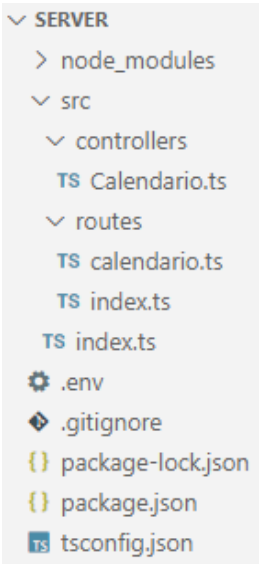
Exercício 5 – Adicione ao código do Exercício 4 a funcionalidade para lidar com rotas desconhecidas.

Requisitos:

- Resposta para rotas inexistentes: o servidor deve retornar:
  - Formato da resposta: JSON;
  - Status code: 404 (Not Found).



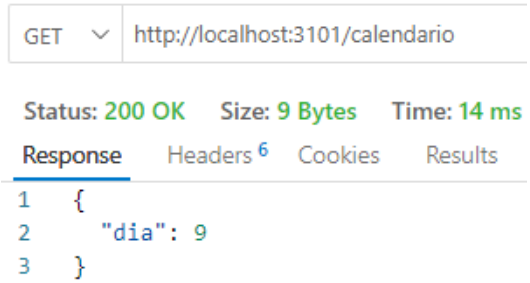
Exercício 6 – Crie um projeto com a estrutura mostrada ao lado com rotas para os métodos da classe Calendario.




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

class Calendario {
  dayOfMonth(req: Request, res: Response) {
    const dia = new Date().getDate();
    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() + "";
    }
   res.json({ r });
  }
}

export default new Calendario();
```

Exemplos de requisições:

O método dayOfMonth deve ser chamado pela seguinte rota:



O método date deve ser chamado pela seguinte rota: