Objetivo:

I. Middleware;
II. Objeto locals;
III. Middleware global;
IV. JSON Web Token.

Instruções para criar um projeto para reproduzir os exemplos:

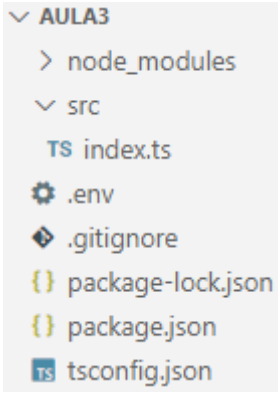
1. Crie uma pasta de nome aula3 (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
2. Abra a pasta aula3 no VS Code e acesse o terminal do VS Code;
3. No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node (arquivo package.json). 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;
4. No terminal, execute o comando `npm i express` para instalar o pacote express;
5. No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém as definições de tipos do 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.
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;
7. No terminal, execute o comando `npm i dotenv` para instalar o pacote dotenv. As variáveis de ambientes são acessadas através do objeto `process.env`. Porém, as variáveis declaradas no arquivo .env não são carregadas pelo ambiente de execução do Node no objeto `process.env`. Usaremos o dotenv parar carregar as variáveis do arquivo .env no objeto `process.env`;
8. No terminal, execute o comando `npm i jsonwebtoken` para instalar a biblioteca que possui ferramentas para manipular tokens de autenticação e autorização;
9. No terminal, execute o comando `npm i -D @types/jsonwebtoken` para instalar o pacote com as definições de tipos da biblioteca JWT (JSON Web Token);
10. No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo tsconfig.json):
11. Crie o arquivo .gitignore na raiz do projeto e coloque a linha para ignorar a pasta node_modules;
12. Crie o arquivo .env na raiz do projeto e coloque as seguintes variáveis de ambiente:

```
PORT = 3001
JWT_SECRET = @tokenJWT
```

O conteúdo da variável JWT_SECRET será usado como chave pelo algoritmo JWT para codificar e decodificar os tokens. Desta forma, uma pessoa que queira decodificar o token terá de ter acesso a esta chave. Aqui foi sugerida a chave `@tokenJWT`, mas você poderá escolher qualquer outra chave.

13. Crie a pasta src na raiz do projeto e crie o arquivo index.ts dentro da pasta src;
    No momento o projeto terá a estrutura mostrada ao lado.

Estrutura de pastas e arquivos do projeto:



14. Para rodar a aplicação crie as propriedades start e dev na propriedade scripts do arquivo package.json:

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

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`.

Para rodar a aplicação usaremos os comandos:

`npm run dev` ou

`npm run start`

No momento o arquivo package.json terá a estrutura mostrada a seguir:

```
{
  "name": "aula3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "ts-node ./src",
    "dev": "ts-node-dev ./src"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.3.1",
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.1"
```

```
  },
  "devDependencies": {
    "@types/express": "^4.17.17",
    "@types/jsonwebtoken": "^9.0.2",
    "ts-node": "^10.9.1",
    "ts-node-dev": "^2.0.0",
    "typescript": "^5.1.6"
  }
}
```

I. Middleware

No Node, uma requisição HTTP precisa ser processada por uma função que recebe como parâmetro os objetos `Request` e `Response`. No exemplo a seguir, a rota `HTTP GET /um` será direcionada para a função `objetivo`. A função `objetivo` faz o tratamento da rota.

```
const objetivo = (req: Request, res: Response) => {
    res.send("Resposta");
}

app.get("/um", objetivo);
```

Podemos adicionar uma função para ser executada antes da função que faz o tratamento da rota. No exemplo a seguir a função `intermediaria` será chamada antes da função `objetivo`. A função `intermediaria` é uma middleware, ou seja, uma interface entre a rota e a função `objetivo`.

```
const objetivo = (req: Request, res: Response) => {
    res.send("Resposta");
}

const intermediaria = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}

app.get("/dois", intermediaria, objetivo);
```

A função middleware precisa receber como parâmetro os objetos `Request`, `Response` e `NextFunction`. A chamada da função `next()`, no corpo da middleware, faz a execução continuar para a próxima função. Nesse exemplo a chamada da função `next()` fará a função `objetivo` ser executada.

Podemos ter várias funções compondo o middleware. No exemplo a seguir foram usadas as funções `inter1` e `inter2` para compor o middleware.

```
const objetivo = (req: Request, res: Response) => {
    res.send("Resposta");
}

const inter1 = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}
```

```
const inter2 = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}

app.get("/tres", inter1, inter2, objetivo);
```

O código terá a seguinte ordem de execução: a instrução `next()`, no corpo da função `inter1`, chamará a função `inter2` e a instrução `next()`, no corpo da função `inter2`, chamará a função `objetivo`.

> Os middlewares são funções intermediárias adicionadas no processamento da rota.

Coloque o código a seguir no arquivo /src/index.ts do projeto para testar as rotas HTTP GET /um, /dois e /tres.

```
import express, {Request, Response, NextFunction} 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;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});

const objetivo = (req: Request, res: Response) => res.send("Resposta");

app.get("/um", objetivo);

const intermediaria = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}

app.get("/dois", intermediaria, objetivo);

const inter1 = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}

const inter2 = (req: Request, res: Response, next: NextFunction) => {
    next(); //chama a próxima função ou rota
}

app.get("/tres", inter1, inter2, inter1, inter2, objetivo);
```

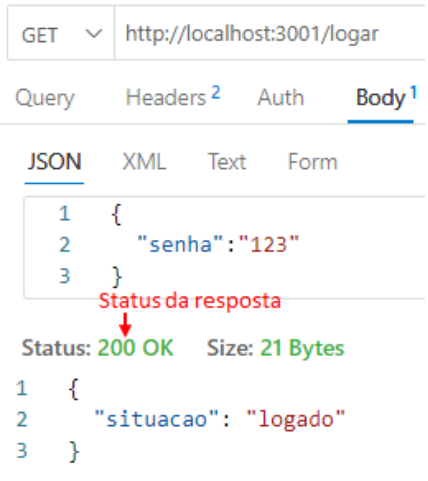
Os middlewares são usados para realizar tarefas como autenticação, validação de entrada, manipulação de cabeçalhos, registro de logs etc. No exemplo a seguir a função `objetivo` só será executada se o usuário fornecer a senha 123 como parâmetro.

```
import express, {Request, Response, NextFunction} 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;
const app = express(); // cria o servidor e coloca na variável app
app.use(express.json()); // suportar parâmetros JSON no body da requisição
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});

const validar = (req: Request, res: Response, next: NextFunction) => {
    const {senha} = req.body;
    if( senha && senha === "123" ){
        return next(); //chama a próxima função ou rota
    }
    // resposta com HTTP Method 401 (unauthorized)
    return res.status(401).send({error:"Não autorizado"});
};

const objetivo = (req: Request, res: Response) => {
    res.send({situacao: "logado"});
}

app.get("/logar", validar, objetivo);
```

Se a instrução `next()` não for executada no corpo da função middleware, então a instrução `return res.status(401).send({error:"Não autorizado"})` será executada, fazendo o servidor interromper a execução da rota e retornar o código de status 401. Podemos usar qualquer código de status na resposta, porém é importante sermos coerente.

Exemplo usando senha incorreta:

Exemplo usando senha correta:



II. Objeto locals

O objeto `locals` é uma propriedade do objeto `Request`. O papel do `locals` é permitir o compartilhamento de dados entre os middlewares e as funções subsequentes no processamento da rota, sem que esses dados sejam expostos diretamente para fora do escopo da requisição, ou seja, é um mecanismo seguro e eficiente para passar informações temporárias dentro do contexto de uma única requisição.

Como exemplo substitua as funções `validar` e `objetivo` no código anterior para testar o uso do objeto `locals`.

```
const validar = (req: Request, res: Response, next: NextFunction) => {
    const {senha} = req.body; // obtém a propriedade senha do body da requisição
    if( senha && senha === "123" ){
        // passa os dados pelo res.locals para o próximo nível da middleware
        res.locals = {status:"logado"};
        return next(); //chama a próxima função ou rota
    }
    // resposta com HTTP Method 401 (unauthorized)
    return res.status(401).send({error:"Não autorizado"});
};

const objetivo = (req: Request, res: Response) => {
    // obtém os dados do nível anterior da middleware que foram armazenados no objeto locals
    const {status} = res.locals;
    res.send({situacao: status});
}
```

Observe que o conteúdo do objeto `locals` foi definido na função `validar` e, posteriormente, acessado no corpo da função `objetivo`.

III. Middleware global

Uma rota definida usando o método `use`, do Express, e sem caminho, será compatível com qualquer rota. Como as rotas são testadas de cima para baixo no arquivo, então a função `use` no exemplo a seguir será executada em todos os casos. Desta forma, a função `use` faz o papel de um middleware no código a seguir.

```
import express, {Request, Response, NextFunction} 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;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});

app.use((req,res,next) => {
```

```
    const {senha} = req.body;
    if( senha && senha === "123" ){
        return next(); //chama a próxima função ou rota
    }
    // resposta com HTTP Method 401 (unauthorized)
    return res.status(401).send({error:"Não autorizado"});
});

const objetivo = (req: Request, res: Response) => {
    res.send({situacao: "logado"});
}

app.get("/logar", objetivo);
```

> O objeto `Request` dentro do método `use` não possui acesso direto à propriedade `locals`. A propriedade `locals` é específica para as funções de rota e middlewares subsequentes, e não é diretamente acessível no nível do middleware global.

IV. JSON Web Token

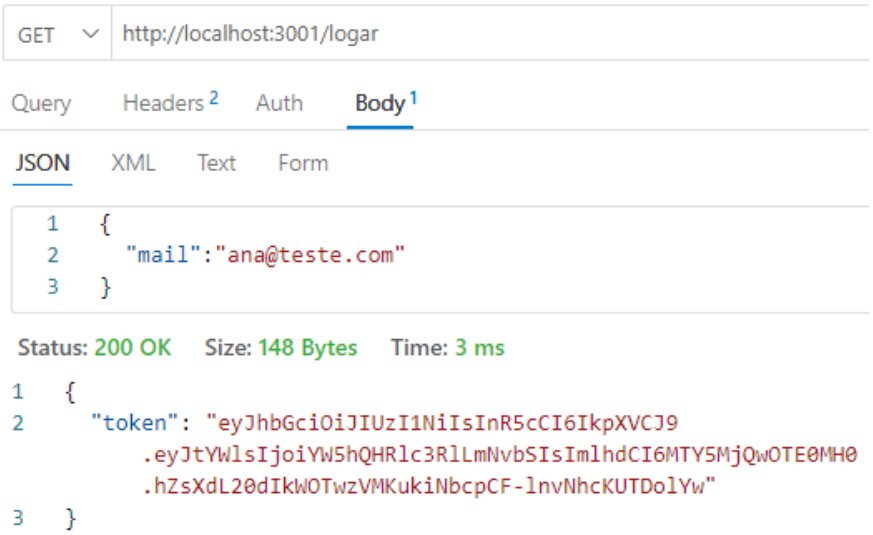
Na autenticação de rotas HTTP, um token é uma representação compacta de informações que é usado para autenticar e autorizar solicitações. Tokens são frequentemente usados como mecanismo de segurança para verificar a identidade de um usuário ou sistema que faz uma requisição a um servidor. Eles ajudam a evitar a necessidade de enviar credenciais (como nome de usuário e senha) a cada requisição, o que pode ser inseguro.

Existem diferentes tipos de tokens, sendo o JWT (JSON Web Token) um dos mais comuns na autenticação de serviços web. Um JWT é um formato de token que consiste em três partes: o cabeçalho (header), o payload (carga) e a assinatura. Cada parte é codificada em Base64 e separada por pontos (veja os dois pontos no token da próxima figura).

- Cabeçalho (header): contém informações sobre o tipo de token e o algoritmo de assinatura usado;
- Payload (carga): são os dados que queremos guardar no token. Geralmente, usamos dados de login, como a identificação do usuário, suas permissões, prazo de validade do token e outros metadados relevantes;
- Assinatura: é a parte final do token. Ela é usada para verificar se o token foi alterado durante a transmissão entre as partes.

Um fluxo típico de autenticação com token envolve os seguintes passos:

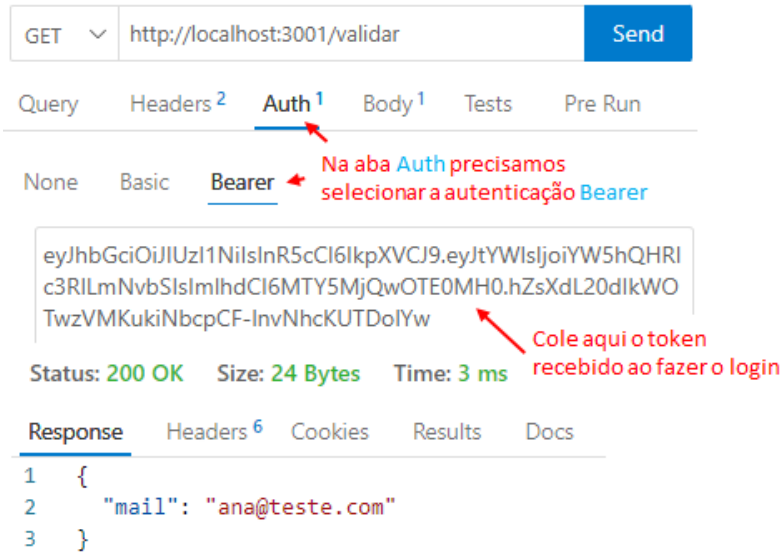
1. O usuário envia suas credenciais (como nome de usuário e senha) ao servidor. Na figura a seguir isso é representado pelo JSON com o e-mail `{"mail":"ana@teste.com"}` enviado pelo body da requisição:



2. O servidor verifica as credenciais e, se estiverem corretas, cria um token JWT contendo as informações relevantes, como a identificação do usuário e as permissões. No exemplo a seguir o token é gerado pela função `sign` do JWT tendo como payload o objeto {"mail":"ana@teste.com"} recebido como parâmetro na variável `entrada`:

```
generateToken = async (entrada:any) => jwt.sign(entrada, process.env.JWT_SECRET as string);
```

3. O token é enviado de volta ao cliente como resultado da autenticação bem-sucedida. Na figura anterior, o token foi recebido no objeto {token: "eyJhbGciOiJIUzI1..."};
4. O cliente armazena o token (geralmente em um cookie ou armazenamento local) e o envia como parte do cabeçalho de autorização em todas as futuras requisições;
5. O servidor recebe o token, verifica sua validade, decodifica o payload para obter as informações do usuário e, se necessário, verifica se o usuário tem as permissões adequadas para acessar a rota solicitada. A figura a seguir mostra o token sendo enviado pelo header da requisição:

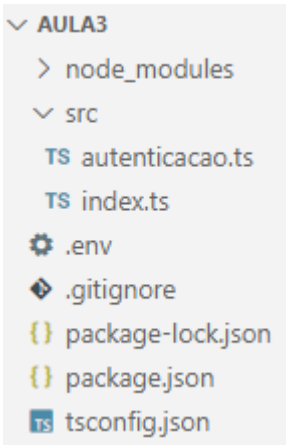


Bearer token: é um token de segurança com a propriedade de que qualquer parte em posse do token (um "portador", em inglês bearer) tem a certeza que a outra parte reconhecerá o token. O uso de um token de portador não exige que o portador comprove a posse do material

da chave criptográfica (prova de posse, em inglês proof-of-possession).

Para testar o JWT crie o arquivo src/autenticacao.ts, assim como é mostrado ao lado, e copie os códigos a seguir para os arquivos autenticacao.ts e index.ts.

Estrutura de pastas e arquivos do projeto:



Código do arquivo src/autenticacao.ts:

```
import { Request, Response, NextFunction } from "express";
import * as jwt from "jsonwebtoken";
import * as dotenv from 'dotenv';
dotenv.config()

// cria um token usando os dados de entrada e a chave da variável de ambiente JWT_SECRET
export const generateToken = async (entrada:any) => jwt.sign(entrada, process.env.JWT_SECRET as
string);

// verifica se o usuário possui autorização
export const authorization = async (req: Request, res: Response, next: NextFunction) => {
    // o token precisa ser enviado pelo cliente no header da requisição
    const authorization:any = req.headers.authorization;
    try {
        // autorização no formato Bearer token
        const [,token] = authorization.split(" ");
        // valida o token
        const decoded = <any>jwt.verify(token, process.env.JWT_SECRET as string);
        if( !decoded ){
            res.status(401).json({error:"Não autorizado"});
        }
        else{
            // passa os dados pelo res.locals para o próximo nível da middleware
            res.locals = decoded;
        }
    } catch (error) {
        // o token não é válido, a resposta com HTTP Method 401 (unauthorized)
        return res.status(401).send({error:"Não autorizado"});
    }
    return next(); //chama a próxima função
};
```

Código do arquivo src/index.ts:

```
import express, {Request, Response, NextFunction} from "express";
import dotenv from "dotenv";
import { authorization, generateToken } from "./autenticacao";
dotenv.config();
// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
    console.log(`Rodando na porta ${PORT}`);
});

const login = async (req: Request, res: Response) => {
    const { mail } = req.body;
    if( mail && mail !== "" ){
        const token = await generateToken({mail});
        return res.json({ token });
    }
    return res.status(401).send({error:"Não autorizado"});
};

app.get("/logar", login);

const processa = async (req: Request, res: Response) => {
    const {mail} = res.locals;
    res.send({mail});

};

app.get("/validar", authorization, processa);
```

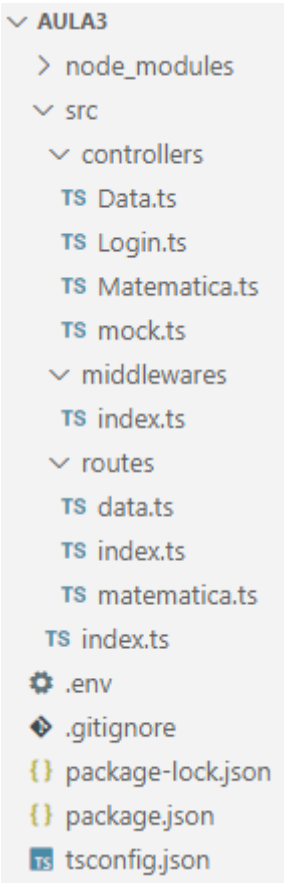
Exercício

Exercício 1 - Crie a estrutura de pastas e arquivos mostrada ao lado e coloque os códigos das figuras a seguir nos respectivos arquivos.

As operações (métodos e funções) do pacote controllers processam as rotas definidas no pacote routes.

Codificar as restrições de acesso nas seguintes rotas:

- HTTP POST /login: não possui restrição de acesso. Para efetuar o login o usuário deverá fornecer o e-mail definido no array `users` do arquivo controllers/mock.ts;
- HTTP GET /data/day e /data/month: possui a restrição de acesso para usuários logados. O usuário deverá enviar o token na requisição;
- HTTP GET /data/year: possui a restrição de acesso para usuários com o perfil admin;
- HTTP GET /matematica/sum: não possui restrição de acesso;
- HTTP GET /matematica/dif: possui a restrição de acesso para usuários com o perfil admin.




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

class Data {
    public async dia(_: Request, res: Response): Promise<Response> {
        const r = new Date().getDate() < 9?
                    "0"+(new Date().getDate()+1) : ""+(new Date().getDate());
        return res.json({ r });
    }

    public async mes(_: Request, res: Response): Promise<Response> {
        const r = new Date().getMonth() < 9?
                    "0"+(new Date().getMonth()+1) : ""+(new Date().getMonth());
        return res.json({ r });
    }

    public async ano(_: Request, res: Response): Promise<Response> {
        const r = ""+new Date().getFullYear();
        return res.json({ r });
    }
}

export default new Data();
```

Figura 1 – Código do arquivo src/controllers/Data.ts.

```
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);
        return res.json({r});
    }

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

export default new Matematica();
```

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

```
import { Request, Response } from 'express';
import { generateToken } from '../middlewares';
import users from './mock';

async function login(req: Request, res: Response): Promise<Response> {
    const { mail } = req.body;
    //verifica se o e-mail existe
    for (let i = 0; i < users.length; i++) {
        if (users[i].mail === mail) {
            const token = await generateToken(users[i]);
            return res.json({ token });
        }
    }
    return res.json({ error: "Usuário não localizado" });
}

export default login;
```

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

```
const users = [
    {
        mail:"a@teste.com",
        profile:"admin"
    },
    {
        mail:"b@teste.com",
        profile:"user"
    }
```

```
];

export default users;
```

Figura 4 – Código do arquivo src/controllers/mock.ts.

```
import { Request, Response, NextFunction } from "express";
import * as jwt from "jsonwebtoken";
import * as dotenv from 'dotenv';
dotenv.config()

// cria um token usando os dados do usuário e a chave armazenada na variável de ambiente JWT_SECRET
export const generateToken = async (usuario:any) => jwt.sign(usuario, process.env.JWT_SECRET as
string);

// verifica se o usuário possui autorização
export const authorization = async (req: Request, res: Response, next: NextFunction) => {
    // o token precisa ser enviado pelo cliente no header da requisição
    const authorization:any = req.headers.authorization;
    try {
        // autorização no formato Bearer token
        const [,token] = authorization.split(" ");
        // valida o token
        const decoded = <any>jwt.verify(token, process.env.JWT_SECRET as string);
        if( !decoded ){
            res.status(401).json({error:"Não autorizado"});
        }
        else{
            // passa os dados pelo res.locals para o próximo nível da middleware
            res.locals = decoded;
        }
    } catch (error) {
        // o token não é válido, a resposta com HTTP Method 401 (unauthorized)
        return res.status(401).send({error:"Não autorizado"});
    }
    return next(); //chama a próxima função
};

// requer a autorização de admin para acessar o recurso
export const authAdmin = async (_: Request, res: Response, next: NextFunction) => {
    // obtém os dados do nível anterior da middleware,
    // isso evita ter de ler novamente req.headers.authorization
    const {profile} = res.locals;
    if( profile !== 'admin' ){
        return res.status(401).send({error:"Sem autorização para acessar o recurso"});
    }
    return next(); //chama a próxima função
```

```
};
```

Figura 5 – Código do arquivo src/middlewares/index.ts.

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

const routes = Router();

routes.get('/day', Data.dia);
routes.get('/month', Data.mes);
routes.get('/year', Data.ano);

export default routes;
```

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

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

const routes = Router();

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

export default routes;
```

Figura 7 – Código do arquivo src/routes/matematica.ts.

```
import { Router, Request, Response } from "express";
import matematica from './matematica';
import login from '../controllers/Login';
import data from './data';

const routes = Router();

routes.post('/login', login);
routes.use("/data", data);
routes.use("/matematica", matematica);

// underscore atua como um espaço reservado para um argumento que desejamos ignorar
routes.use( (_:Request,res:Response) => res.json({error:"Requisição desconhecida"}) );

export default routes;
```

Figura 8 – Código do arquivo src/routes/index.ts.

```
import express from "express";
import routes from './routes';
```

```
import dotenv from "dotenv";
dotenv.config();

// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
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 9 – Código do arquivo src/index.ts.