

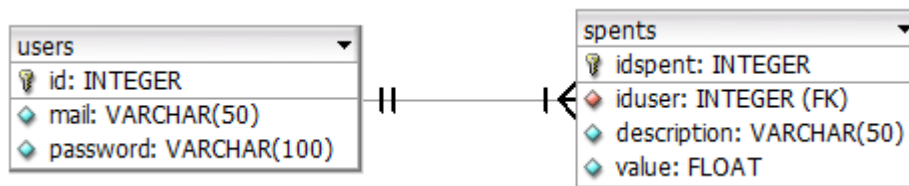
Objetivos:

- TypeORM;
- Criar o projeto Node;
- Migrations e entities;
- Controllers.

i. TypeORM

O TypeORM é um ORM (Object–Relational Mapping - Mapeamento Objeto-Relacional) que pode ser executado em plataformas NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo e Electron e pode ser usado com TypeScript e JavaScript (ES5, ES6, ES7, ES8). Ele utiliza uma nomenclatura semelhante ao Hibernate (<https://typeorm.io>).

Para apresentar os conceitos e comandos do TypeORM faremos a persistência de dados nas tabelas users (usuários) e spents (gastos) no SGBD PostgreSQL. Use o SQLite no caso de você não ter instalado o SGBD PostgreSQL.

**ii. Criar o projeto Node**

Siga os passos a seguir para criar um servidor:

- a) Crie uma pasta de nome **servidor** (pode ser qualquer outro nome de pasta) no local de sua preferência do computador;
- b) Abra a pasta **servidor** no VS Code;
- c) Acesse o terminal do VS Code e digite o comando **npm init -y** para criar o arquivo de configuração **package.json** do projeto Node;
- d) Adicione as seguintes dependências:

```
npm i typeorm@0.3.10
```

Observação: tente instalar a versão atual do pacote typeorm, se não der certo use a versão 0.3.10. No momento da confecção desta aula a última versão do pacote typeorm é 0.3.12, porém, por algum motivo ela não consegue gerar o arquivo de migração no Passo K desta aula.

```
npm i express dotenv jsonwebtoken bcrypt pg
```

Usaremos o pacote bcrypt para nos ajudar a codificar e decodificar a senha armazenada no campo password da tabela users do SGBD (<https://www.npmjs.com/package/bcrypt>);

Usaremos o pacote `pg` para ter acesso ao SGBD PostgreSQL (<https://www.npmjs.com/package/pg>). Substitua o pacote `pg` pelo `sqlite3` (<https://www.npmjs.com/package/sqlite3>) no caso de você não ter o SGBD PostgreSQL;

- e) Adicione as dependências de desenvolvimento:

```
npm i @types/express @types/jsonwebtoken ts-node ts-node-dev typescript -D
npm i @types/bcrypt @types/pg -D
```

- f) Crie o arquivo `.gitignore` na raiz do projeto e coloque a linha para ignorar a pasta `node_modules`;

- g) 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.

- h) Execute o comando `tsc --init` para gerar o arquivo `tsconfig.json` na raiz do projeto e depois substitua o conteúdo pelo JSON a seguir:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  }
}
```

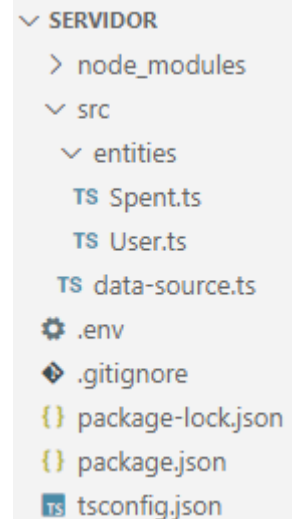
iii. Migrations e entities

- i) Coloque as seguintes propriedades na propriedade `scripts` do arquivo `package.json`. As migrações serão usadas para criarmos as cláusulas SQL e submeter elas para o SGBD:

```
"scripts": {
  "dev": "ts-node-dev src/index.ts",
  "start": "ts-node src/index.ts",
  "migration:generate": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:generate ./src/migrations/default",
  "migration:run": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:run",
  "migration:revert": "typeorm-ts-node-commonjs -d ./src/data-source.ts migration:revert"
},
```

- j) Crie a estrutura de pastas e arquivos mostrada ao lado e copie os códigos da Figura 1 a Figura 3 nos respectivos arquivos;

Na pasta `entities` colocamos os arquivos que definem as regras de criação das entidades (tabelas) e no arquivo `data-source.ts` (este arquivo pode ter qualquer nome) colocamos a regra de conexão com o BD.



A propriedade `entities`, do JSON definido no arquivo `data-source.ts`, possui o caminho das entidades a serem utilizadas como modelo para criar as tabelas:

```
entities: ["src/entities/*.ts"], // entidades que serão convertidas em tabelas
```

A propriedade `migrations`, do JSON definido no arquivo `data-source.ts`, possui o caminho onde serão criados os arquivos de migração com as cláusulas SQL:

```
migrations: ["src/migrations/*.ts"], // local onde estarão os arquivos de migração
```

```
import { Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate } from
"typeorm";
import * as bcrypt from "bcrypt";

@Entity({name:"users"})
export class User {
    // define a chave primária como auto incremento
    @PrimaryGeneratedColumn()
    id: number;

    @Column({nullable: false, unique:true, length: 70})
    mail: string;

    @Column({nullable: false, select: false, length: 100})
    password: string;

    @BeforeInsert() //a função hashPassword é disparada antes do insert e update
    @BeforeUpdate()
    hashPassword(): void {
        if (this.password) {
            // a senha é codificada usando o algoritmo do pacote bcrypt
            this.password = bcrypt.hashSync(this.password, bcrypt.genSaltSync(10));
        }
    }
}
```

```
compare(input: string): Promise<boolean> {  
  // a senha fornecida em input é comparada com a senha do registro armazenado no SGBD  
  return bcrypt.compare(input, this.password);  
}  
}
```

Figura 1 – Código do arquivo src/entities/User.ts.

```
import { Entity, PrimaryGeneratedColumn, ManyToOne, JoinColumn, Column } from  
"typeorm";  
import { User } from "../User";  
  
@Entity({ name: "spents" })  
export class Spent {  
  @PrimaryGeneratedColumn()  
  id: number;  
  
  // cascade define que ao excluir o usuário os gastos serão excluídos  
  @ManyToOne((type) => User, { onDelete: 'CASCADE' })  
  // JoinColumn é usado para definir o lado da relação que contém a "join column" com  
  a FK  
  @JoinColumn({  
    name: "iduser",  
    referencedColumnName: "id", // id da entidade User  
    foreignKeyName: "fk_user_id" // pode ser qualquer nome usado para  
    você identificar a FK  
  })  
  user: User;  
  
  @Column({ length: 50, nullable: false })  
  description: string;  
  
  @Column({ type: 'decimal', precision: 10, scale: 2, nullable: false })  
  value: number;  
}
```

Figura 2 – Código do arquivo src/entities/Spent.ts.

```
import { DataSource } from "typeorm";  
  
//https://orkhan.gitbook.io/typeorm/docs/data-source-options  
const AppDataSource = new DataSource({  
  database: 'bdaula', // se for SQLite, então use bdaula.sqlite  
  type: "postgres", // se for SQLite, então use sqlite  
  host: 'localhost', // não use esta propriedade se for sqlite  
  port: 5432, // não use esta propriedade se for sqlite  
  username: 'postgres', // não use esta propriedade se for sqlite  
  password: '123', // não use esta propriedade se for sqlite  
  // true indica que o schema do BD será criado a cada vez que a aplicação
```

```

inicializar
  // deixe false ao usar migrations
  synchronize: false,
  logging: true, // true indica que as consultas e erros serão exibidas no terminal
  entities: ["src/entities/*.ts"], // entidades que serão convertidas em tabelas
  migrations: ["src/migrations/*.ts"], // local onde estarão os arquivos de migração
  maxQueryExecutionTime: 2000 // 2 seg.
});

// https://orkhan.gitbook.io/typeorm/docs/data-source
AppDataSource
  .initialize()
  .then(() => {
    console.log("Data Source inicializado!")
  })
  .catch((e) => {
    console.error("Erro na inicialização do Data Source:", e)
  });

export default AppDataSource;

```

Figura 3 – Código do arquivo src/data-source.ts.

- k) Execute o comando `npm run migration:generate` para gerar o arquivo de migração na pasta migrations. Observe que a pasta `migrations` não existia antes;

Na prática foi executado o seguinte comando

```

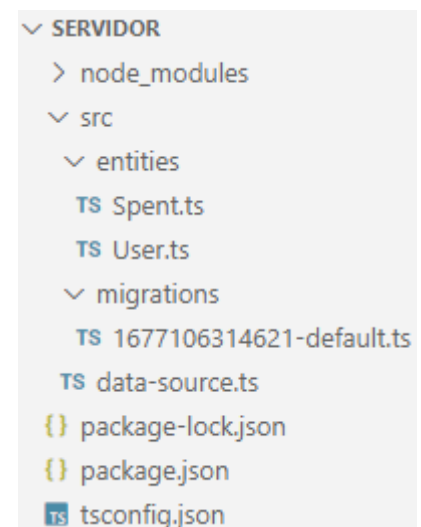
typeorm-ts-node-commonjs -d ./src/data-source.ts
migration:generate ./src/migrations/default

```

O comando `migration:generate` funciona apenas com arquivos JS. Alternativamente usamos ts-node em conjunto com typeorm para executar os arquivos de migração TS (<https://orkhan.gitbook.io/typeorm/docs/migrations#running-and-reverting-migrations>).

O parâmetro `-d` é usado para passar o caminho do arquivo que possui as configurações do data source (Figura 3). A direita do comando `migration:generate` passamos o local onde criaremos o arquivo de migração. Neste caso criaremos um arquivo nomeado como `src/migrations/{timestamp}-default.ts`. A migração possui como parte do nome o timestamp do horário de criação da migration.

O arquivo de migração (Figura 4) possui uma classe com os métodos `up` e `down`. No método `up` estão os comandos SQL para fazer a migração. No método `down` estão os comandos SQL para reverter a migração. Você poderá editar os comandos SQL dos métodos `up` e `down`.



```
import { MigrationInterface, QueryRunner } from "typeorm";

export class default1677106314621 implements MigrationInterface {
    name = 'default1677106314621'

    public async up(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`CREATE TABLE "users" ("id" SERIAL NOT NULL, "mail"
character varying(70) NOT NULL, "password" character varying(100) NOT NULL, CONSTRAINT
"UQ_2e5b50f4b7c081ecee476ad128" UNIQUE ("mail"), CONSTRAINT
"PK_a3ffb1c0c8416b9fc6f907b7433" PRIMARY KEY ("id"))`);
        await queryRunner.query(`CREATE TABLE "spents" ("id" SERIAL NOT NULL,
"description" character varying(50) NOT NULL, "value" numeric(10,2) NOT NULL, "iduser"
integer, CONSTRAINT "PK_fdf8432c53458c1211cd521463c" PRIMARY KEY ("id"))`);
        await queryRunner.query(`ALTER TABLE "spents" ADD CONSTRAINT "fk_user_id"
FOREIGN KEY ("iduser") REFERENCES "users"("id") ON DELETE CASCADE ON UPDATE NO
ACTION`);
    }

    public async down(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`ALTER TABLE "spents" DROP CONSTRAINT "fk_user_id"`);
        await queryRunner.query(`DROP TABLE "spents"`);
        await queryRunner.query(`DROP TABLE "users"`);
    }
}
```

Figura 4 – Código do arquivo criado na pasta src/migrations.





- I) Execute o comando `npm run migration:run` para submeter as cláusulas SQL no SGBD.

Além de criar as tabelas `users` e `spents` no SGBD, será inserido um registro na tabela `migrations` do BD com os dados da última migração. A tabela `migrations` será criada se ela não existir. A seguir tem-se conteúdo da tabela `migrations` ao executar o comando `migration:run`:

```
select * from migrations;
```

id [PK] integer	timestamp bigint	name character varying
1	1677106314621	default1677106314621

Tabelas no SGBD:

- ▼  Tables (3)
 - >  migrations
 - >  spends
 - >  users

Observações:

- Ser for realizada alguma alteração nos arquivos de entidade será necessário executar os comandos `migration:generate` e `migration:run`, nesta sequência;
- Utilize o comando `npm run migration:revert` para desfazer a última modificação no SGBD. Na prática serão submetidos os comandos SQL do método `down` da classe de migração (Figura 4).


```
routes.post('/', UserController.create);
```

- O método create, da classe UserController, usa a instrução `AppDataSource.manager.save` para submeter os dados no SGBD e, na sequência, usa a função `generateToken({ id: usuario.id, mail: usuario.mail })` para criar o token com os dados de login do usuário. Este token é retornado para o cliente.

Operação de login: A seguir tem-se um exemplo de requisição.

The screenshot shows a REST client interface. The top bar indicates a POST request to `http://localhost:3001/login` with a status of 200 OK, size of 183 Bytes, and time of 96 ms. The 'Body' tab is selected, showing the request body in JSON format: `{ "mail": "a@teste.com", "password": "abc" }`. The 'Response' tab is also selected, showing the response body in JSON format: `{ "id": 2, "mail": "a@teste.com", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MiwibWVpbCI6ImFAdGVzdGUuY29tIiwiaWF0IjoxNjc3NDIwNzczfQ.pdbtb56zJWF6GqilivR2puQ9VFiaSQQ-1miJsuRn2M0" }`.

- O servidor recebe os dados de login na rota definida no arquivo `src/routes/index.ts`
- ```
routes.post('/login', UserController.login);
```
- Essa rota não possui controle de acesso neste nível. Ela apenas direciona para o método `login` da classe `UserController`;
  - O método `login` da classe `UserController` usa a instrução a seguir para obter o registro que possui o e-mail fornecido. Na prática ela faz `select * from users where mail = 'abc@teste.com' limit 1`:

```
const usuario: any = await AppDataSource
 .getRepository(User)
 .createQueryBuilder("user")
 .select()
 .addSelect('user.password')
 .where("user.mail=:mail", { mail })
 .getOne();
```

- A instrução `usuario.compare(password)` é usada para comparar as senhas codificadas;
- A instrução `generateToken({ id: usuario.id, mail: usuario.mail })` é usada para criar o token a ser retornado para o cliente.

**Operação de update e-mail:** Para fazer a requisição é necessário passar o token no formato Bearer Token:



PUT ▼ http://localhost:3001/usuario Send Status: 200 OK Size: 30 Bytes

Query Headers <sup>2</sup> Auth <sup>1</sup> **Body <sup>1</sup>** Tests

**Json** Xml Text Form Form-encode

Json Content

```
1 {
2 "mail": "aa@teste.com"
3 }
```

Response Headers <sup>6</sup> Cookies

```
1 {
2 "id": 2,
3 "mail": "aa@teste.com"
4 }
```

Query Headers <sup>2</sup> **Auth <sup>1</sup>** Body <sup>1</sup> Tests Pre Run

None Basic **Bearer** OAuth 2 Ntlm Aws

Bearer Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MiwiWFpbCI6ImFAdGVzdGUuY29tliwiaWF0IjoxNzNDIiwNzcfQ.pdbtb56zJWF6GqilivR2puQ9VFiaSQQ-1miJsuRn2M0
```

- O servidor recebe os dados na rota definida no arquivo `src/routes/index.ts`  
`routes.post('/usuario', user);`
- Essa rota não possui controle de acesso neste nível. Porém no nível seguinte, a chamada do método `update` está condicionada a aprovação na função `authorization`. Desta forma, temos o controle de acesso ao método `update`:

```
routes.put('/', authorization, UserController.update);
```

**Operação para criar gasto:** Para fazer a requisição é necessário passar o token no formato Bearer Token:

POST ▼ http://localhost:3001/gasto Send Status: 200 OK Size: 41 Bytes

Query Headers <sup>2</sup> Auth <sup>1</sup> **Body <sup>1</sup>** Tests

**Json** Xml Text Form Form-encode

Json Content

```
1 {
2 "description": "Posto",
3 "value": 89.5
4 }
```

Response Headers <sup>6</sup> Cookies

```
1 {
2 "id": 2,
3 "descricao": "Posto",
4 "valor": 89.5
5 }
```

- O servidor recebe os dados na rota definida no arquivo `src/routes/index.ts`  
`routes.use("/gasto", authorization, spent);`
- Essa rota possui controle de acesso neste nível, ou seja, as rotas definidas no nível `spent` estarão disponíveis após a aprovação na função `authorization`. Após aprovado será chamado o método `create`, da classe `SpentController`

```
routes.post('/', SpentController.create);
```

- No método create a instrução `const { id } = res.locals` é usada para obter o id do usuário atual. Lembre-se que este id foi colocado no objeto locals no método authorization;
- De forma equivalente são feitas as operações para listar, excluir e atualizar gastos.

```
import * as express from "express";
import * as dotenv from 'dotenv';
dotenv.config();

import routes from './routes';

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

const app = express();
app.use(express.json());
app.listen(PORT, () => console.log(`Rodando na porta ${PORT}`));

app.use(routes);
```

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

```
import AppDataSource from "../data-source";
import { Request, Response } from 'express';
import { Spent } from '../entities/Spent';
import { User } from '../entities/User';

class SpentController {
 public async create(req: Request, res: Response): Promise<Response> {
 const { description, value } = req.body;
 //verifica se foram fornecidos os parâmetros
 if (!description || !value || description.trim() === "" || value === "") {
 return res.json({ error: "A descrição e valor do gasto são necessários" });
 }
 // obtém o id do usuário que foi salvo na autorização na middleware
 const { id } = res.locals;
 const usuario: any = await AppDataSource.manager.findOneBy(User, { id }).catch((e)
=> {
 return { error: "Identificador inválido" };
 })

 if (usuario && usuario.id) {
 const gasto = new Spent();
 gasto.user = usuario;
 gasto.description = description;
 gasto.value = parseFloat(value);
 await AppDataSource.manager.save(Spent, gasto);
 res.json({ id: gasto.id, description: gasto.description, value: gasto.value });
 }
 }
}
```

```
 }
 else {
 return res.json(usuario);
 }
}

public async update(req: Request, res: Response): Promise<Response> {
 const { id, description, value } = req.body;
 if(!id || id === "" || !description || description === "" || !value || value ===
 ""){
 return res.json({ error: "Identificação, descrição e valor são necessários" });
 }
 const gasto: any = await AppDataSource.manager.findOneBy(Spent, { id }).catch((e) =>
 {
 return { error: "Identificador inválido" };
 })
 if (gasto && gasto.id) {
 gasto.description = description;
 gasto.value = value;
 const r = await AppDataSource.manager.save(Spent, gasto).catch((e) =>
e.message);
 return res.json(r);
 }
 else if (gasto && gasto.error) {
 return res.json({gasto});
 }
 else {
 return res.json({ error: "Gasto não localizado" });
 }
}

public async delete(req: Request, res: Response): Promise<Response> {
 const { id } = req.body;
 if(!id || id === ""){
 return res.json({ error: "Identificação necessária" });
 }
 const gasto: any = await AppDataSource.manager.findOneBy(Spent, { id }).catch((e) =>
 {
 return { error: "Identificador inválido" };
 });

 if (gasto && gasto.id) {
 const r = await AppDataSource.manager.remove(Spent, gasto).catch((e) =>
e.message);
 return res.json(r);
 }
 else if (gasto && gasto.error) {
 return res.json(gasto);
 }
}
```

```

 else {
 return res.json({ error: "Gasto não localizado" });
 }
}

public async list(req: Request, res: Response): Promise<Response> {
 // obtém o id do usuário que foi salvo na autorização na middleware
 const { id } = res.locals;
 const usuario: any = await AppDataSource.manager.findOneBy(User, { id }).catch((e)
=> {
 return { error: "Identificador inválido" };
 })

 if (usuario && usuario.id) {
 const repo = AppDataSource.getRepository(Spent);
 const gastos = await repo.find({
 /*relations:{
 user:true
 },*/
 where: { user: { id } },
 order: {
 description: 'asc'
 }
 });
 return res.json(gastos);
 }
 else if (!usuario) {
 return res.json({ error: "Usuário não identificado" });
 }
 else {
 return res.json(usuario)
 }
}

}

export default new SpentController();

```

Figura 6 – Código do arquivo src/controllers/SpentController.ts.

```

import AppDataSource from "../data-source";
import { Request, Response } from 'express';
import { User } from '../entities/User';
import { generateToken } from '../middlewares';

class UserController {
 public async login(req: Request, res: Response): Promise<Response> {
 const { mail, password } = req.body;
 //verifica se foram fornecidos os parâmetros
 if (!mail || !password || mail.trim() === "" || password.trim() === "") {

```

```

 return res.json({ error: "e-mail e senha necessários" });
 }
 // como a propriedade password não está disponível para select {select: false},
 // então precisamos usar esta consulta para forçar incluir a propriedade
 const usuario: any = await AppDataSource
 .getRepository(User)
 .createQueryBuilder("user")
 .select()
 .addSelect('user.password')
 .where("user.mail=:mail", { mail })
 .getOne();

 if (usuario && usuario.id) {
 const r = await usuario.compare(password);
 if (r) {
 // cria um token codificando o objeto {id,mail}
 const token = await generateToken({ id: usuario.id, mail: usuario.mail });
 // retorna o token para o cliente
 return res.json({
 id: usuario.id,
 mail: usuario.mail,
 token
 });
 }
 return res.json({ error: "Dados de login não conferem" });
 }
 else {
 return res.json({ error: "Usuário não localizado" });
 }
}

public async create(req: Request, res: Response): Promise<Response> {
 const { mail, password } = req.body;
 //verifica se foram fornecidos os parâmetros
 if (!mail || !password || mail.trim() === "" || password.trim() === "") {
 return res.json({ error: "e-mail e senha necessários" });
 }
 const obj = new User();
 obj.mail = mail;
 obj.password = password;
 // o hook BeforeInsert não é disparado com AppDataSource.manager.save(User,JSON),
 // mas é disparado com AppDataSource.manager.save(User,objeto do tipo User)
 // https://github.com/typeorm/typeorm/issues/5493
 const usuario: any = await AppDataSource.manager.save(User, obj).catch((e) => {
 // testa se o e-mail é repetido
 if (/^(mail)[\s\S]+(already exists)/.test(e.detail)) {
 return { error: 'e-mail já existe' };
 }
 })
 return { error: e.message };
}

```

```
 })
 if (usuario.id) {
 // cria um token codificando o objeto {idusuario,mail}
 const token = await generateToken({ id: usuario.id, mail: usuario.mail });
 // retorna o token para o cliente
 return res.json({
 id: usuario.id,
 mail: usuario.mail,
 token
 });
 }
 return res.json(usuario);
 }

 // o usuário pode atualizar somente os seus dados
 public async update(req: Request, res: Response): Promise<Response> {
 const { mail, password } = req.body;
 // obtém o id do usuário que foi salvo na autorização na middleware
 const { id } = res.locals;
 const usuario: any = await AppDataSource.manager.findOneBy(User, { id }).catch((e) => {
 return { error: "Identificador inválido" };
 })
 if (usuario && usuario.id) {
 if (mail !== "") {
 usuario.mail = mail;
 }
 if (password !== "") {
 usuario.password = password;
 }
 const r = await AppDataSource.manager.save(User, usuario).catch((e) => {
 // testa se o e-mail é repetido
 if (/((mail)[\s\S]+(already exists))/).test(e.detail)) {
 return ({ error: 'e-mail já existe' });
 }
 return e;
 })
 if (!r.error) {
 return res.json({ id: usuario.id, mail: usuario.mail });
 }
 return res.json(r);
 }
 else if (usuario && usuario.error) {
 return res.json(mail)
 }
 else {
 return res.json({ error: "Usuário não localizado" });
 }
 }
}
```

```
}

export default new UserController();
```

Figura 7 – Código do arquivo src/controllers/UserController.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 chave armazenada na variável de ambiente
JWT_SECRET
export const generateToken = async usuario => jwt.sign(usuario, process.env.JWT_SECRET);

// 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 = req.headers.authorization;
 try {
 // autorização no formato Bearer token
 const [,token] = authorization.split(" ");
 // valida o token
 const decoded = <{id:string,mail:string}>jwt.verify(token, process.env.JWT_SECRET);
 if(!decoded || !decoded.id){
 res.status(401).send({error:"Não autorizado"});
 }
 else{
 // passa os dados pelo res.locals para ser acessado nos controllers
 res.locals = {id: decoded.id};
 }
 } catch (error) {
 // o token não é válido, a resposta com HTTP Method 401 (unauthorized)
 res.status(401).send({error:"Não autorizado"});
 return;
 }
 return next(); //chama a próxima função
};
```

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

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

const routes = Router();

routes.get('/', SpentController.list);
routes.post('/', SpentController.create);
routes.put('/', SpentController.update);
```

```
routes.delete('/', SpentController.delete);

export default routes;
```

Figura 9 – Código do arquivo src/routes/spent.ts.

```
import { Router } from "express";
import UserController from "../controllers/UserController";
import { authorization } from "../middlewares";

const routes = Router();

routes.post('/', UserController.create);
routes.put('/', authorization, UserController.update);

export default routes;
```

Figura 10 – Código do arquivo src/routes/user.ts.

```
import { Router } from "express";
import UserController from "../controllers/UserController";
import { authorization } from "../middlewares";

const routes = Router();

routes.post('/', UserController.create);
routes.put('/', authorization, UserController.update);

export default routes;
```

Figura 11 – Código do arquivo src/routes/index.ts.