EXERCÍCIO 1

a. Qual a sua opinião em relação a usar strings para representar os ids? Você concorda que seja melhor do que usar números?
RESPOSTA: Concordo por que uma string tem mais possibilidades de combinações o que torna o id mais seguro.

---

b. A partir de hoje, vamos tentar isolar, ao máximo, as nossas lógicas em classes. Uma das vantagens disso é, por exemplo, utilizar a hierarquia para fazer modificações simples. Dado isso, crie uma classe que possua um um método público para gerar um id.
RESPOSTA:

```
export class IdGenerator {
public generate(): string {
return v4();
}
}
```

EXERCÍCIO 2

```
const createUser = async (id: string, email: string, password: string) => {
await connection
.insert({
id,
email,
password,
})
.into(userTableName);
};
```

a. Explique o código acima com as suas palavras.
RESPOSTA: O código usa o query builder para enviar a tabela "userTableName"as informações do usuario

---

b. Comece criando a tabela de usuários. Coloque a query que você utilizou no arquivo de respostas.
RESPOSTA:

```
CREATE TABLE userTableName(
id VARCHAR(255) PRIMARY KEY,
email VARCHAR(255) NOT NULL,
password VARCHAR(255) NOT NULL
);
```

---

c. Pela mesma justificativa do exercício anterior, crie uma classe para ser responsável pela comunicação do usuário com a tabela de usuários. Ela deve possuir um método que cria o usuário no banco; além disso, as variáveis necessárias para realizar as queries devem ser atributos dessa classe
RESPOSTA:

```
export class UserDB extends BaseDataBase {
private static TABLE_NAME = “Users”;
public async createUser(
id: string,
email: string,
password: string
): Promise<void> {
await this.getConnection()
.insert({
id,
```

```
name,
email,
password,
})
.into(UserDB.TABLE_NAME);
}
}
```

---

d. Crie um usuário utilizando somente a classe que você criou
RESPOSTA:

```
app.post("/signup", async (req: Request, res: Response) => {
try{
const userData = {
email: req.body.email,
password: req.body.password,
};
if (!userData.email && userData.email.indexOf("@") === -1) {
throw new Error(“Invalid Email”);
}
if (!userData.password && userData.password.length < 6) {
throw new Error(“Invalid Password”);
}
const id = idManager.generate();
userDB.createUser(id, userData.email, userData.password);
const token = auth.generateToken({id});
res.status(200).send({
token : token
})
}catch (err){
res.status(400).send({
message: err.message
});
}
});
```

EXERCÍCIO 3
a. O que a linha `as string` faz? Por que precisamos usar ela ali?
RESPOSTA:
as string está ali por que a key é uma string que nós
escolhemos, a partir do qual o token é
criptografado

---

b.Agora, crie a classe que será responsável pela autorização dos usuários com um método que gere o token. Além disso, crie uma interface a parte para representar o input desse método. Lembre-se de colocar todas as constantes em atributos da classe.
RESPOSTA:

```
import * as jwt from “jsonwebtoken”;
export class Authenticator {
private static EXPIRES_IN = “1min”;
public generateToken(input: AuthenticationData): string {
const token = jwt.sign(
{
id: input.id,
},
process.env.JWT_KEY as string,
```

```
{
expiresIn: Authenticator.EXPIRES_IN,
}
);
return token;
}
}
interface AuthenticationData {
id: string;
}
```

EXERCÍCIO 4

a. Crie o endpoint que realize isso, com as classes que você implementou anteriormente
RESPOSTA:

```
app.post("/signup", async (req: Request, res: Response) => {
try{
const userData = {
email: req.body.email,
password: req.body.password,
};
if (!userData.email && userData.email.indexOf("@") === -1) {
throw new Error(“Invalid Email”);
}
if (!userData.password && userData.password.length < 6) {
throw new Error(“Invalid Password”);
}
const id = idManager.generate();
userDB.createUser(id, userData.email, userData.password);
const token = auth.generateToken({id});
res.status(200).send({
token : token
})
}catch (err){
res.status(400).send({
message: err.message
});
}
});
```

---

b. Altere o seu endpoint para ele não aceitar um email vazio ou que não possua um `"@"`
RESPOSTA:

```
if (!userData.email && userData.email.indexOf("@") === -1) {
throw new Error(“Invalid Email”);
}
```

---

c. Altere o seu endpoint para ele só aceitar uma senha com 6 caracteres ou mais
RESPOSTA:

```
if (!userData.password && userData.password.length < 6) {
throw new Error(“Invalid Password”);
}
```

EXERCÍCIO 5

a. Altere a classe do seu banco de dados para que ele tenha um método que retorne as informações de um usuário a partir do email

```
public async getUserByEmail(email: string): Promise<any> {
const result = await this.getConnection()
.select("*")
.from(UserDB.TABLE_NAME)
.where({ email });
return result[0];
}
```

---

b. Teste a sua função
Retorna o usuário

EXERCÍCIO 6

a. Crie o endpoint que realize isso, com as classes que você implementou anteriormente
RESPOSTA:

```
app.post("/login", async (req: Request, res: Response) => {
try {
const userData = {
email: req.body.email,
password: req.body.password,
};
if (!userData.email && userData.email.indexOf("@") === -1) {
throw new Error(“Invalid Email”);
}
const user = await userDb.getUserByEmail(userData.email);
const decryptedPassword = hashManager.compare(
userData.password,
user.password
);
if (!decryptedPassword) {
throw new Error(“Invalid Password”);
}
const token = auth.generateToken({ id: user.id });
res.status(200).send({
token,
});
} catch (err) {
res.status(400).send({
message: err.message,
});
}
});
```

---

b. Altere o seu endpoint para ele não aceitar um email vazio ou que não possua um `"@"`
RESPOSTA:

```
if (!userData.email && userData.email.indexOf("@") === -1) {
throw new Error(“Invalid Email”);
}
```

EXERCÍCIO 7

a. O que a linha `as any` faz? Por que precisamos usá-la ali?
RESPOSTA:

---

b. Altere a sua classe do JWT para que ela tenha um método que realize a mesma funcionalidade da função acima
RESPOSTA:

```
export class Authenticator {
public generateToken(input: AuthenticationData): string {
const token = jwt.sign(
{
id: input.id,
},
("" + process.env.JWT_KEY) as string,
{
expiresIn: “1y”,
}
);
return token;
}
public getData(token: string): AuthenticationData {
const payload = jwt.verify(
token,
("" + process.env.JWT_KEY) as string
) as any;
const result = {
id: payload.id,
};
return result;
}
}
interface AuthenticationData {
id: string;
}
```

EXERCíCIO 8

a. Comece alterando a classe do banco de dados para que ela tenha um método que retorne o usuário a partir do id
RESPOSTA:

```
public async getUserById(id: string): Promise<any> {
const result = await this.getConnection()
.select("*")
.from(UserDB.TABLE_NAME)
.where({ id });
return result[0];
}
```

---

b. Crie o endpoint com as especificações passadas
RESPOSTA:

```
app.get("/user/profile", async (req: Request, res: Response) => {
try {
const token = req.headers.authorization as string;
const authData = auth.getData(token);
const userInfo = await userDB.getUserById(authData.id);
res.status(200).send({
```

```
id: userInfo.id,
name: userInfo.name,
email: userInfo.email,
});
} catch (err) {
res.status(400).send({
message: err.message,
});
}
});
```