EXERCÍCIO 1

a. O que são os `round` e `salt`? Que valores são recomendados para o `round`? Que valor você usou? Por quê?
RESPOSTA:
bcrypt é um método de criptografia do tipo hash para senhas baseado no Blowfish.
SALT é uma string que é aleatória na senha (ou texto).
São recomendados para o round o valor de 12.

---

b. Instale o bcryptjs no seu projeto e comece criando a classe HashManager. Por ora, implemente a função que **criptografe** uma string usando o bcryptjs.
RESPOSTA:
```
export class HashManager {
public async hash(t: string): Promise<string> {
const rounds = 12;
const salt = await bcrypt.genSalt(rounds);
const result = await bcrypt.hash(t, salt);

return result;

}
}
```

---

c. Agora, crie a função que verifique se uma string é correspondente a um hash, use a função `compare` do bcryptjs
RESPOSTA:
```
export class HashManager {
public async hash(t: string): Promise<string> {
const rounds = 12;
const salt = await bcrypt.genSalt(rounds);
const result = await bcrypt.hash(t, salt);
return result;
}
public async compare(t: string, h: string): Promise<boolean> {
return bcrypt.compare(t, h);
}
}
```

EXERCÍCIO 2

a. Para realizar os testes corretamente, qual deles você deve modificar primeiro? O cadastro ou o login? Justifique.
RESPOSTA:
Farei primeiro o endpoint de signup pois é ele que gera o ID e cria a senha que vai ser guardada no banco e deve ser criptografada

---

b. Faça a alteração do primeiro endpoint
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 encryptedPassword = await hashManager.hash(userData.password);
const id = idManager.generate();
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
});
}
});
```

---

c. Faça a alteração do segundo endpoint
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,
});
}
});
```

---

d. No exercício de ontem, nós criamos o endpoint `user/profile`. Também temos que modificar esse endpoint devido à adição da criptografia? Justifique.
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,
});
}
```

EXERCÍCIO 3

a. Altere a sua tabela de usuários para ela possuir uma coluna `role`. Considere que pode assumir os valores `normal` e `admin`. Coloque `normal` como valor padrão.
RESPOSTA:

```
ALTER TABLE userTableName ADD role VARCHAR(255) NOT NULL default 'normal';
```

---

b. Altere a interface `AuthenticationData` e `Authenticator` para representarem esse novo tipo no token.
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,
role: input.role,
},
process.env.JWT_KEY as string,
{
expiresIn: Authenticator.EXPIRES_IN,
}
);
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,
role: payload.role,
};
return result;
}
}

interface AuthenticationData {
id: string;
role: string;
}
```

---

c. Altere o cadastro para receber o tipo do usuário e criar o token com essa informação
RESPOSTA:

```
app.post("/signup", async (req: Request, res: Response) => {
try {
const userData = {
email: req.body.email,
password: req.body.password,
```

```
role: req.body.role,
};
const idGenerator = new IdGenerator();
const id = idGenerator.generate();
const hashManager = new HashManager();
const hashPassword = await hashManager.hash(userData.password);
const userDb = new UserDatabase();
await userDb.createUser(id, userData.email, hashPassword, userData.role);
const authenticator = new Authenticator();
const token = authenticator.generateToken({
id,
role: userData.role,
});
res.status(200).send({
token,
});
} catch (err) {
res.status(400).send({
message: err.message,
});
}
});
```

---

d. Altere o login para crair o token com o `role` do usuário
RESPOSTA:

```
app.post("/login", async (req: Request, res: Response) => {
  try {
    if (!req.body.email || req.body.email.indexOf("@") === -1) {
      throw new Error("Invalid email");
    }
    const userData = {
      email: req.body.email,
      password: req.body.password,
    };
    const userDatabase = new UserDatabase();
    const user = await userDatabase.getUserByEmail(userData.email);
    const hashManager = new HashManager();
    const comapreResult = await hashManager.compare(
      userData.password,
      user.password
    );
    if (!comapreResult) {
      throw new Error("Invalid password");
    }
    const authenticator = new Authenticator();
    const token = authenticator.generateToken({
      id: user.id,
      role: user.role,
    });
    res.status(200).send({
      token,
    });
  } catch (err) {
    res.status(400).send({
      message: err.message,
    });
  }
});
```

EXERCÍCIO 4

a. Altere o endpoint para que retorne um erro de Unauthorized para os usuários que "não sejam normais" e tentem acessar esse endpoint
RESPOSTA:

```
app.get("/user/profile", async (req: Request, res: Response) => {
  try {
    const token = req.headers.authorization as string;
    const authenticator = new Authenticator();
    const authenticationData = authenticator.getData(token);
    if (authenticationData.role !== "normal") {
      throw new Error("Only a normal user can access this funcionality");
    }
    const userDb = new UserDatabase();
    const user = await userDb.getUserById(authenticationData.id);
    res.status(200).send({
      id: user.id,
      email: user.email,
      role: authenticationData.role,
    });
  } catch (err) {
    res.status(400).send({
      message: err.message,
    });
  }
});
```

EXERCICIO 5

Implemente o endpoint que realizará a deleção de um usuário. As especificações são:
- *Verbo/Método*: **DELETE**
- *Path:* `/user/:id`
- Somente admins podem acessar esse endpoint

RESPOSTA:

```
public async deleteUser(id: string): Promise<any> {
```

```
    await this.getConnection().del().from(UserDB.TABLE_NAME).where({ id });
}

app.delete("/user/:id", async (req: Request, res: Response) => {
  try {
    const token = req.headers.authorization as string;
    const authenticator = new Authenticator();
    const authenticationData = authenticator.getData(token);
    if (authenticationData.role !== "admin") {
      throw new Error("Only a admin user can access this funcionality");
    }
    const id = req.params.id;
    const userDatabase = new UserDatabase();
    await userDatabase.deleteUser(id);
    res.sendStatus(200);
  } catch (err) {
    res.status(400).send({
      message: err.message,
    });
  }
  await BaseDatabase.destroyConnection();
});
```

EXERCÍCIO 6

Implemente o endpoint que retorne as informações (id e email) de um usuário a partir do seu id. As especificações são:
- *Verbo/Método*: **GET**
- *Path:* `/user/:id`
- Tanto admins como usuários normais conseguem usar essa funcionalidade

RESPOSTA:

```
app.get("/user/:id", async (req: Request, res: Response) => {
  try {
    const token = req.headers.authorization as string;
    const authenticator = new Authenticator();
    authenticator.getData(token);
    const id = req.params.id;
    const userDatabase = new UserDatabase();
    const user = await userDatabase.getUserById(id);
    res.status(200).send({
      id: user.id,
      email: user.email,
      role: user.role,
    });
  } catch (err) {
    res.status(400).send({
      message: err.message,
    });
  }
});
```

EXERCÍCIO 7

a. Implemente essa classe e faça com que o `UserDatabase` a implemente. Faça todas as alterações necessárias nessa classe.
RESPOSTA:

```
import knex from "knex";
import Knex from "knex";
export abstract class BaseDatabase {
  private static connection: Knex | null = null;
  protected getConnection(): Knex {
    if (BaseDatabase.connection === null) {
      BaseDatabase.connection = knex({
        client: "mysql",
        connection: {
          host: process.env.DB_HOST,
          port: 3306,
          user: process.env.DB_USER,
          password: process.env.DB_PASSWORD,
          database: process.env.DB_DATABASE_NAME,
        },
      });
    }
    return BaseDatabase.connection;
  }
   public static async destroyConnection(): Promise<void> {
    if (BaseDatabase.connection) {
      await BaseDatabase.connection.destroy();
      BaseDatabase.connection = null;
    }
  }
}


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

---

b. Utilize o método `destroyConnection` nos momentos oportunos (vulgo, no final dos endpoints)
RESPOSTA:

```
 await BaseDatabase.destroyConnection();
```