Aula 50

Introdução a Autenticação

Labenu_

Sumário

Labenu_

O que vamos ver hoje?

- Como implementar um fluxo de **autenticação** no backend

- Duas das libs que vão nos ajudar no processo:
  - ***uuid***
  - ***json-web-token***

Introdução

Labenu_

Introdução

- Na penúltima semana, vocês criaram uma API completa, a ToDoList

- Aquela aplicação possuía alguns pontos fracos como, por exemplo:

    - O usuário não precisava provar sua identidade para alterar seus dados cadastrais
    - Qualquer um tem acesso às tarefas de um usuário apenas com seu id, que é uma informação relativamente pública

Introdução

- Nas aulas de hoje e amanhã, resolveremos esses problemas implementando um sistema de **autenticação**, que envolve:
    - Criação de um identificador único (**id**) para esse usuário
    - Uso de um **token** de acesso a determinados endpoints
    - Implementação de um endpoint de login, que gere um token para o usuário que apresentar **credenciais** (email e senha, por exemplo) válidas

UUID

Labenu_

UUID 

- A **autenticação** é o processo de **identificação** dos usuários nas nossas aplicações
- Como não temos controle sobre as informações dadas pelos usuários, nossa aplicação precisa gerar um valor único (id) que garanta essa identificação
- Embora consigamos pensar em várias formas de gerar esses valores manualmente, existe um formato padrão: o UUID

UUID 

- UUID é um acrônimo para *Universally Unique Identifier* que também é chamado de GUID (*Globally Unique Identifier*)
- Ele é uma string com 36 caracteres hexadecimais exibidos em 5 grupos separados por hífen, seguindo o padrão:
  8 - 4 - 4 - 4 - 12

`123e4567-e89b-12d3-a456-426655440000`

UUID 

- Existem 5 versões do UUID:
  - **v1:** Gerados a partir do tempo e do MAC Address
  - **v2:** Gerados a partir de um outro id, tempo e MAC Address
  - **v3:** Gerados a partir da criptografia MD5 a partir de outro identificador e um nome
  - **v4:** Gerados a partir de números aleatórios e pseudo-aleatórios
  - **v5:** Gerados a partir da criptografia SHA1 a partir de outro identificador e um nome

UUID 

- De todas essas, recomendam-se a **v1** e a **v4** para ids de usuários
- No curso, vamos usar o **v4** simplesmente por uma **escolha**. O **v1** serviria muito bem para isso também.
- Antes de vermos a implementação disso tudo no *Typescript*, é importante perceber que a chance de repetição (ou colisão) de algum UUID é muito pequena:

Se você gerar 1 bilhão de UUIDs por segundo durante 100 anos, existe a chance de 50% de existir uma colisão

UUID 

- No *Typescript*, o UUID é implementado por uma dependência chamada **uuid**

```
npm install uuid @types/uuid

yarn add uuid @types/uuid
```

UUID 

- Para utilizá-lo é muito simples:

```
import { v4 } from uuid;

const generatedId = v4();
```

UUID 

- Vamos isolar essa lógica em uma função, para não termos que importar a lib em todos os arquivos onde ela for utilizada
- Essa abordagem também facilita a manutenção do código, pois vamos alterar somente um lugar quando necessário

Exercício 1:

- Crie uma função **generateId**, que retorne um identificador no padrão UUID

- Para guardar o arquivo que contém essa função, crie uma pasta própria dentro da **src/** .

- Refatore os endpoints de criação de usuários e de tarefas para que eles utilizem a função criada

Exercício 1

```
import { v4 } from 'uuid'

export const generateId = (): string => {
        return v4()
}
```

Labenu_

Pausa para relaxar 😴

- O **UUID** é uma das formas mais usadas para gerar *id*
- Para o Typescript, temos a lib **uuid**

JWT

Labenu_

JWT 

- Agora que temos uma forma padronizada de identificar os usuários, precisamos pensar em como controlar o acesso aos endpoints
- Uma maneira seria exigir as credenciais do usuário nos cabeçalhos de todas as requisições. Além de uma péssima experiência para o usuário, é uma solução pouco segura, pois teríamos informação sensível trafegando constantemente
- Um **token** de autenticação cumpre bem melhor esse propósito, pois o usuário se identifica uma única vez e garante seu acesso à aplicação por um tempo determinado

JWT 

- A forma mais famosa de se criar tokens de autenticação é através do JWT, JsonWebToken
- Ele utiliza alguns algoritmos de criptografia, tais como: RSA, HS256 e Base64
- Um token JWT sempre tem a forma:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

- header
- payload
- assinatura

JWT 

- O fluxo para gerar o JWT é um pouco complexo por envolver criptografia
- Mas, como quase tudo no mundo, alguém já fez e deixou de uma forma fácil: a lib ***jsonwebtoken***

```
npm install jsonwebtoken @types/jsonwebtoken

yarn add jsonwebtoken @types/jsonwebtoken
```

JWT 

- Para gerarmos o *token*, podemos usar a função ***sign*** disponível nesta lib
- Ela espera receber três parâmetros:

JWT 

- *payload*: um objeto com as informações do usuário que queremos guardar no token
- *key*: uma string que representa a chave que será usada como base para gerar o token
- *options*: um objeto com as informações adicionais do token, como, por exemplo, o tempo de expiração

```
import * as jwt from 'jsonwebtoken'

const token = jwt.sign(
   { id: "abcdef" },
   "bananinha",
   { expiresIn: "24h" }
)
```

JWT 

- Para recuperarmos a informação do usuário, e, então, identificá-lo, iremos usar a função ***verify***
- Ela não só decodifica a nossa informação como também verifica se ela é válida (formato correto, se já não expirou, etc.)

```
import * as jwt from 'jsonwebtoken'

const token = jwt.sign(
   { id: "abcdef" },
   "bananinha",
   { expiresIn: "24h" }
)

const tokenData = jwt.verify(
   token,
   "bananinha"
)
```

Exercício 2:

- Crie um arquivo **authenticator**, contendo as funções ***generateToken getTokenData***, para implementar, respectivamente, os métodos *sign* e *verify* do *jwt*.

- Crie também um type **AuthenticationData** para representar o *payload* do seu token

(Lembrando: tudo isso é para tornar o código organizado e escalável)

Labenu_

Exercício 2

```
import * as jwt from 'jsonwebtoken'

type AuthenticationData = { id: string }


   export generateToken = (input: AuthenticationData): string =>{
       return jwt.sign(
           input,
           process.env.JWT_KEY as string,
           { expiresIn: process.env.JWT_EXPIRES_IN }
       )

   export getTokenData(token: string): AuthenticationData =>{

       const tokenData = jwt.verify(token, process.env.JWT_KEY as string)

       return tokenData as AuthenticationData
   }
```

Labenu_

Pausa para relaxar 😴

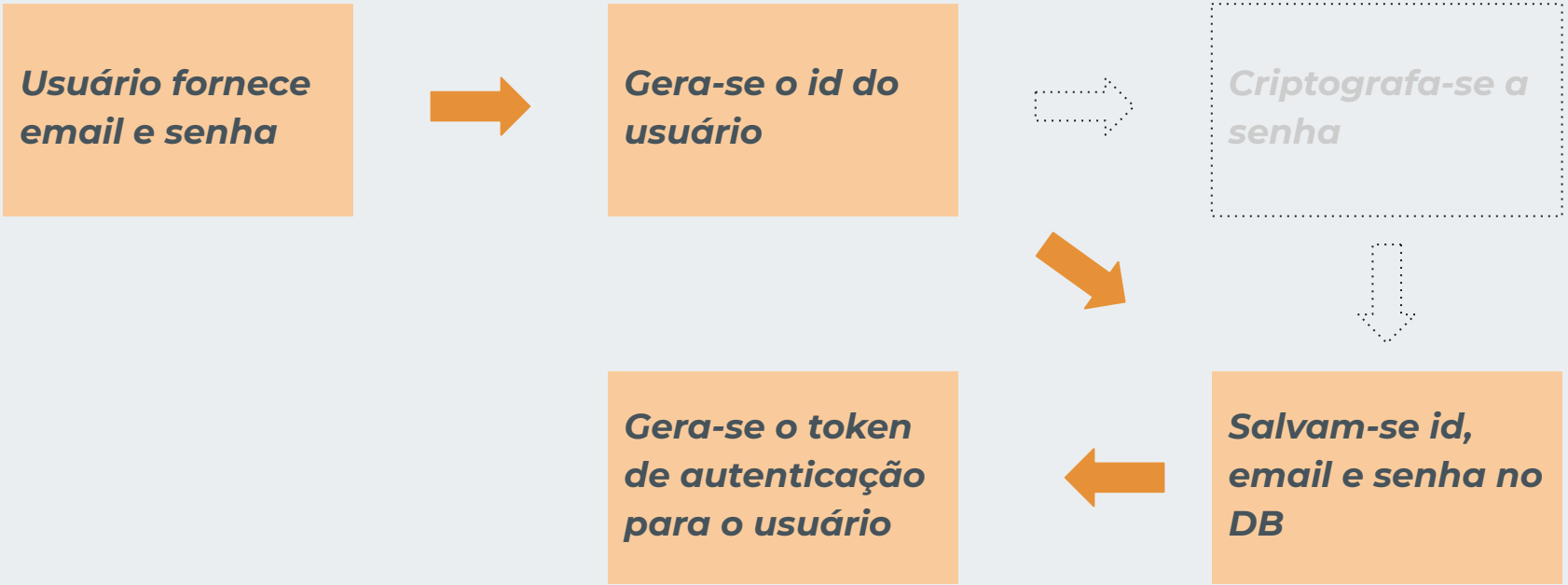
- Existem várias formas de se gerar um token, a que vamos usar aqui é o JWT
- Para o Typescript, temos a lib **jsonwebtoken**
- Utilizamos as funções **sign** e **verify**

Implementação do Signup

Labenu_

Implementação do *Signup* 

- A implementação do *Signup* vai seguir o diagrama abaixo (a criptografia de senha vai ficar para amanhã)



Exercício 3:

Refatore o endpoint de cadastro para incluir um fluxo de autenticação. Os requisitos são:

- O caminho deve ser "/user/signup"
- O usuário precisa escolher uma senha ao se cadastrar (altere a tabela de usuários)
- O usuário deve receber um identificador no padrão UUID
- A resposta deve ter um corpo contendo um token de autenticação (altere o método HTTP para **POST**)

Exercício 3

```
export default async function signup(req: Request, res: Response) {
   try {
       const { name, nickname, email, password } = req.body
       const id = generateId()

       if (!name || !nickname || !email || !password) {
           throw new Error("\"name\",\"nickname\", \"email\" e \"password\" são
obrigatórios")
       }

       await connection.insert({ id, name, nickname, email, password }).into(userTableName)

       res.status(200).send({token: generateToken({id})});
   } catch (error) {...}
}
```

Labenu_

Pausa para relaxar 😴

- *Usuário fornece email e senha*
- *Gera-se o id do usuário*
- *Salvam-se id, email e senha no DB*
- *Gera-se o token de autenticação para o usuário*

Implementação do Login

Labenu_

Implementação do Login 

Login de Usuários

Usuário fornece email e senha

Compara-se a senha salva no banco com a criptografada

Com o id salvo no banco, gera-se o token

Compara-se a senha do banco diretamente com a enviada

Exercício 4:

Crie um endpoint de login de usuários.
Os requisitos são:

- Caminho: "/user/login"
- Método: POST
- Informar email e senha
- Devolver um token de autenticação, caso as credenciais sejam válidas, ou uma mensagem de erro, caso contrário

Exercício 4

```
export default async function login(req: Request, res: Response) {
   try {
       const { email, password } = req.body

       if (!email || !password) throw new Error("\"email\" e \"password\" são obrigatórios")

       const result = await connection(userTableName).select("id").where({email, password})

       const user = result[0]

       if(!user) throw new Error("Usuário ou senha incorretos")

       res.status(200).send({token: generateToken({ id: user.id }) })
   } catch (error) { ... }
}
```

Labenu_

Pausa para relaxar 😴

- *Usuário fornece email e senha*
- *Compara-se a senha do banco diretamente com a enviada*
- *Com o id salvo no banco, gera-se o token*

Endpoints autenticados

Labenu_

Exercício 5:

Transforme o endpoint de editar usuário em um endpoint autenticado. Para isso, ele deve:

- Receber um token pelo cabeçalho da requisição (não será mais necessário passar o id por *path parameters*)
- Editar os dados do usuário, caso o token seja válido, ou devolver um status 401 (Unauthorized), caso contrário

Exercício 5

```
export default async function editUser(req: Request, res: Response) {
   try {
       const tokenData = getTokenData(
           req.headers.auth as string
       )

       ...

       await connection("to_do_list_users")
            .update({ name, nickname, email })
            .where({ tokenData.id })

       res.status(200).send({message: "Usuário atualizado!"});
   } catch (error) {...}
}
```

Labenu_

Resumo

Labenu_

Resumo 

- A **autenticação** é o processo de identificar o usuário
- É uma funcionalidade **complexa** que exige uma análise calma
- Os principais componentes da autenticação são:
  - **Credenciais**: email e senha;
  - **id**
  - **Criptografia** da senha
  - **Token** de autenticação

Resumo 

- O **uuid** é uma das formas mais usadas para gerar *id*
  - Função: v4()

- Existem várias formas de se gerar um token, a que vamos usar aqui é o JWT

- Para o Typescript, temos as libs **uuid** e a **jsonwebtoken**
  - uuid - função: *v4()*
  - jsonwebtoken - função: *sign()*
  - jsonwebtoken - função: *verify()*

Obrigado!