Criptografia

Labenu_

O que vamos ver hoje? 👀

- Criptografia e *hash*
- Bcrypt

Criptografia e Hash

Labenu_

Criptografia e Hash 🔐

- Um **algoritmo de criptografia** tem o objetivo de tornar mais segura a transmissão de dados
- A ideia é simples:

Criptografia e Hash 🔐

- Alguns termos importantes:
  - ***cipher/cypher*** (cifra): o algoritmo em si que encripta o texto
  - ***plaintext*** (texto simples): a entrada do algoritmo, ou seja, o texto que se deseja encriptar
  - ***ciphertext*** (texto encriptado/criptografado): a saída do algoritmo
  - ***cryptanalysis*** (criptoanálise): processo de quebrar um algoritmo de criptografia, ou seja, decifrar a mensagem sem ter acesso às chaves

- Algoritmos de criptografia são divididos em duas grandes categorias: **simétrico** e **assimétrico**

Criptografia e Hash 🔐

- Um algoritmo de criptografia é **simétrico** se ele utiliza **apenas uma chave** para encriptar e decriptar a mensagem
- Ele é **assimétrico** se utiliza um par de chaves, de modo que os dados encriptados por uma só podem ser decriptados pela outra. Uma das chaves é **pública** e a outra é **privada.**

Criptografia e Hash 🔐

- **Hash** é o processo **irreversível** de transformar um dado em uma string aparentemente aleatória
- Hash devolve o **mesmo valor** para a **mesma entrada**
    - Se a entrada for "123456", ele sempre devolve "abcdef", por exemplo
- Usamos algoritmos de hash para 'armazenar' senhas

Criptografia e Hash 🔐

- Começamos guardando o resultado do **hash**

- E, então, podemos usar isso para comparar com uma "tentativa do usuário"

Bcrypt

Labenu_

Bcrypt 

- ***Bcrypt*** é um método de criptografia criado por Niels Provos e David Mazierès em 1999
- Ele utiliza o algoritmo de **hash** chamado de **Blowfish**, criado por Bruce Schneier em 1993
  - É complexo, mas é baseado na manipulação de bits da entrada
  - Não foi encontrada uma criptoanálise desse algoritmo até então
- Uma **vantagem** desse método é que ele possui um fator chamado ***cost*** (custo - numérico) que está relacionado à segurança da senha

Bcrypt 

- Quanto **maior** o *cost*, maior o tempo de execução do algoritmo
- **Qual o melhor *cost* para usar?**
  - Depende do sistema
  - A recomendação é: utilizar o **maior** que conseguir para os equipamentos utilizados rodarem no tempo desejado
- Durante o curso, vamos usar um *cost* de 12, por ser o padrão na maioria das libs

Bcrypt 🐡

- Outra vantagem desse algoritmo é que ele adiciona uma string (chamada de salt) que é **aleatória** antes de criar o **hash**.

- Dessa forma ela evita os ataques chamados ***rainbow table***
    - É a ideia de criar uma tabela de conversão senha-hash, substituindo a necessidade de *processamento* pela de *armazenamento*

Bcrypt 

- Abaixo temos um exemplo de **hash resultante do bcrypt**, no qual o **salt** está em negrito

$2y$10$abcdefghijklmnopqrstuvENCODEDINPUT

- **algorithm:** Assume valores como 2, $2y$, $2a$
- **cost:** o custo, indicado por dois dígitos, seguido por $
- **string aleatória:** com 22 caracteres
- **encoded input:** encriptado a partir da junção do input com o salt

Bcrypt 

- Para node.js, vamos usar a lib ***bcryptjs***

```
npm install bcryptjs @types/bcryptjs
```

Bcrypt 

- Para criar um hash podemos usar a função ***hash*** do bcrypt ou sua versão síncrona, ***hashSync***

- Ela precisa de um **salt**. Este é gerado a partir da função ***genSalt*** (ou ***genSaltSync***) que espera receber o ***cost***, ou, como a lib bcryptjs chama, os ***rounds***.

- No próximo slide, há um arquivo chamado **hashManager**

Bcrypt 

- A nossa função hash:
  - Pega os rounds das variáveis de ambiente
  - Gera o salt
  - Cria o hash com o salt gerado

```
import * as bcrypt from "bcryptjs"

export hash = async (s: string): Promise<string> =>{

    const rounds = Number(process.env.BCRYPT_COST);
    const salt = await bcrypt.genSalt(rounds);
    const result = await bcrypt.hash(s, salt);

  return result;
}
```

Bcrypt 

- O *bcryptjs* também tem funções para comparar um hash com a string que supostamente o gerou (***compare*** e ***compareSync***)

```
import * as bcrypt from "bcryptjs"

export hash = async (s: string): Promise<string> =>{

    const rounds = Number(process.env.BCRYPT_COST);
    const salt = await bcrypt.genSalt(rounds);
    const result = await bcrypt.hash(s, salt);

  return result;
}

export compare = async (s: string, hash: string): Promise<boolean> =>{
    return bcrypt.compare(s, hash);
}
```

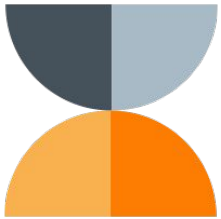
Exercício 1

Crie uma classe **hashManager**, contendo as funções ***hash*** e ***compare***, que implementem as funções homônimas do *bcryptjs*

```
import * as bcrypt from 'bcryptjs'

   export hash = async (s: string): Promise<string> {
       const rounds = Number(process.env.BCRYPT_COST);
       const salt = await bcrypt.genSalt(rounds);
       const cypherText = await bcrypt.hash(text, salt);
       return cypherText;
   }

   export compare = async(s: string, hash: string){
       const result = await bcrypt.compare(s, hash);
       return result;
   }
```



Pausa para relaxar 😴

10 min

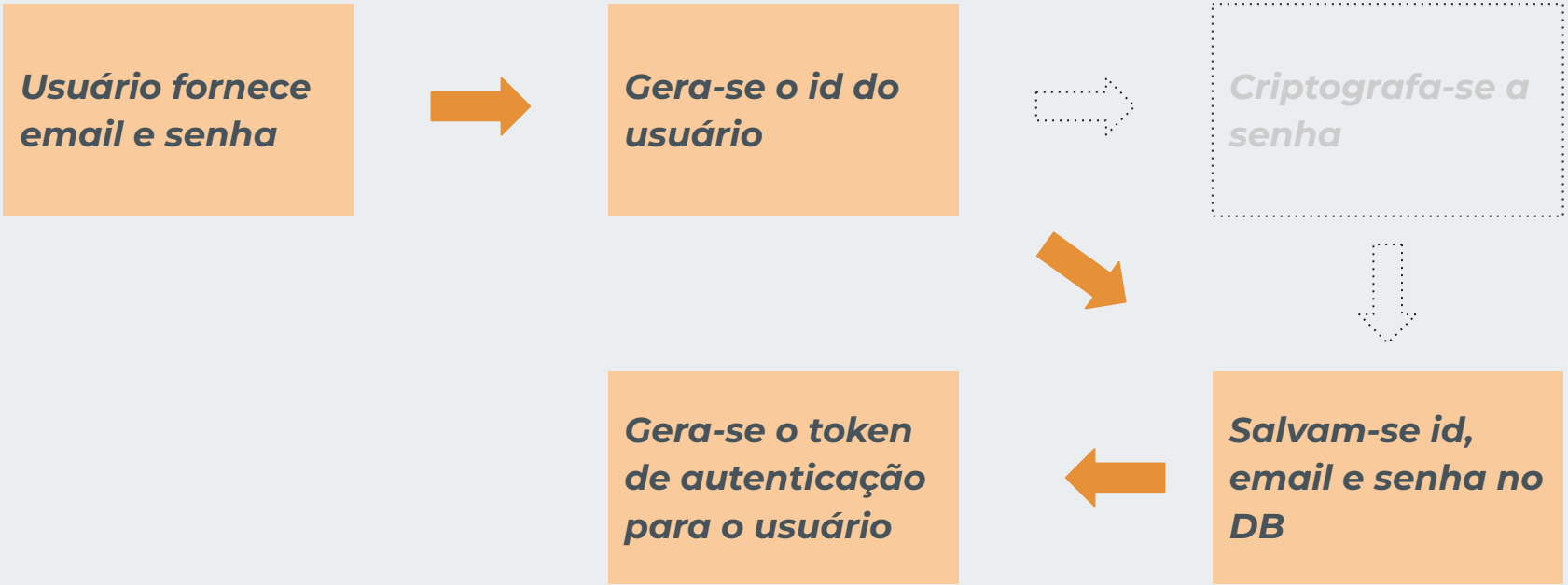
- **Criptografia** e **Hash** são, por definição diferentes em um ponto: a primeira dá para **reverter** e a segunda **não**
- Entretanto, "no jargão", podemos usar a palavra **criptografia** para se **referir a hash**, sem problemas
- Vimos como usar o hash bcrypt, implementado pela lib **bcryptjs**, para encriptar a nossa senha

Alteração no Signup

Labenu_

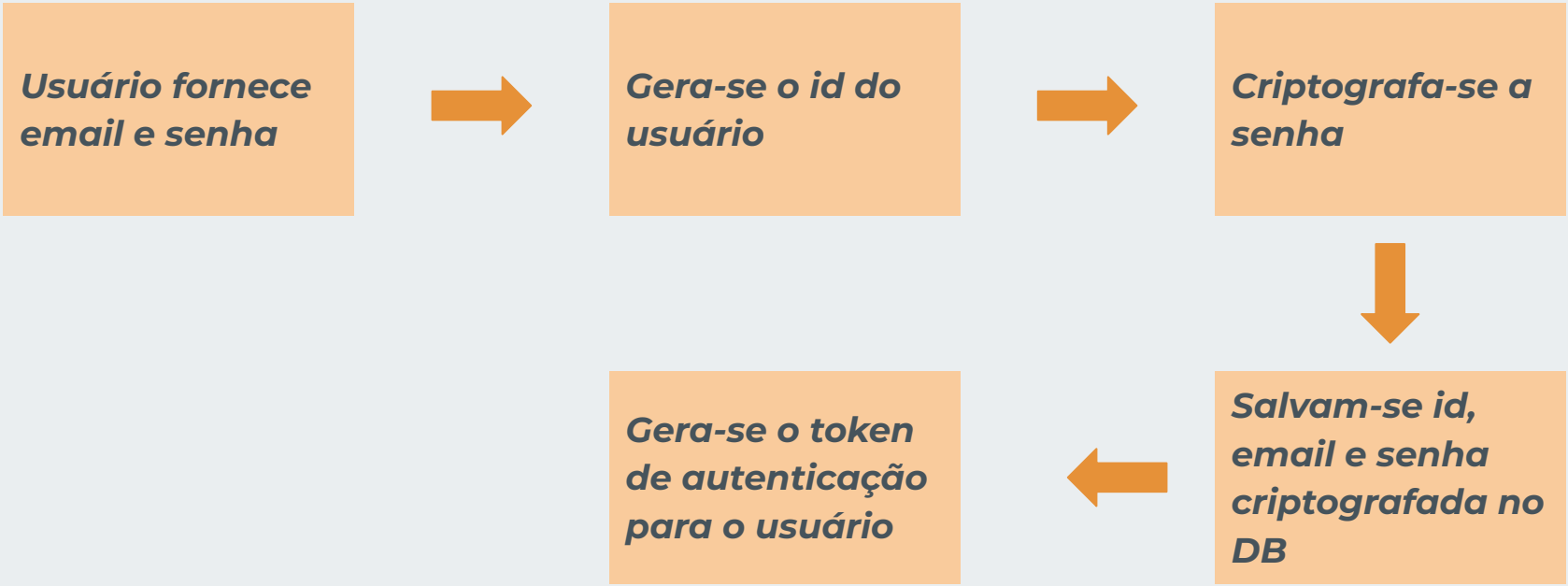
Alteração no *Signup* 📝

- Antes não tínhamos criptografado a senha ainda



Alteração no *Signup* 📝

- Agora vamos alterar a nossa implementação do login para **criptografar a senha antes de salvar no banco**

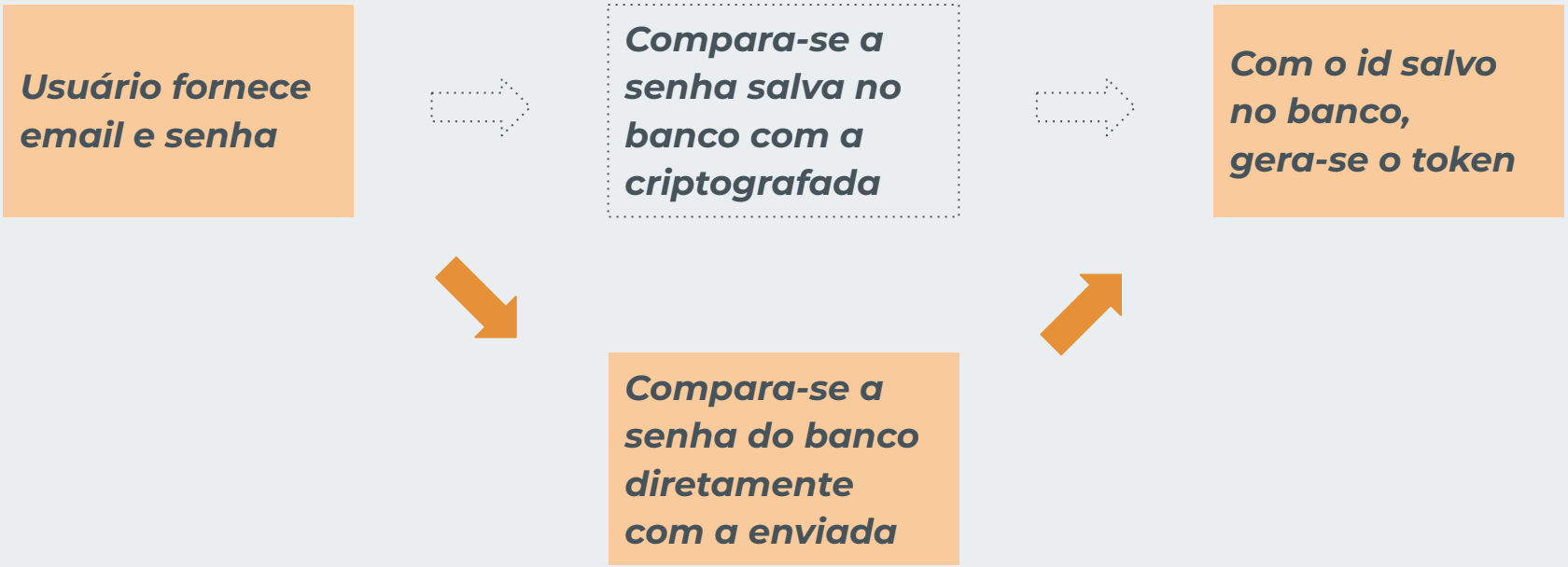


Alteração no Login

Labenu_

Alteração no Login 📝

Login de Usuários



Alteração no 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

Exercício 2

a) Refatore o endpoint de cadastro, utilizando o **hashManager** para que a senha seja salva no banco como *cypherText*.

b) Refatore o endpoint de login utilizando o **hashManager**, para que a senha informada possa ser comparada com o *cypherText* salvo no banco.

```
...

const { name, email, password } = req.body
const id = generateId()

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

const cypherPassword = await hash(password);

await connection("to_do_list_users")
    .insert({ id, name, nickname, email, password: cypherPassword })
    ...
```

```
...

const { email, password } = req.body

const [user] = await connection("to_do_list_users").where({ email })

const passwordIsCorrect = await compare(password, user?.password)

if (!user || !passwordIsCorrect) throw new Error("Credenciais inválidas")

const token = generateToken({ id: user.id })

...
```

Dúvidas? 

Labenu_

Resumo

Labenu_

Resumo 📓

- **Criptografia** e **Hash** são, por definição diferentes em um ponto: a primeira dá para **reverter** e a segunda **não**

- Entretanto, "no jargão", podemos usar a palavra **criptografia** para se **referir a hash**, sem problemas

- Vimos como usar o bcrypt, implementado pela lib **bcryptjs**, para encriptar a nossa senha
  - **hash**
  - **compare**

Obrigado!