Sprint 03 - Cognitive Cibersecurity

Integrantes

- Ana Araújo - RM 99816
- Cauã Simões - RM 98319
- Erick Camargo - RM 99771
- Lucas Silva - RM 550466
- Raphael Papa - RM 552432

Parte prática

Implementamos autenticação e segurança dentro do nosso dashboard (agrovisionaries.com) usando JWT para autenticação e fazendo hash da senha usando bcrypt. O hash da senha permanece salvo dentro do nosso banco de dados e não é disponibilizado para os usuários.

Os usuários só podem entrar na página de dashboard se possuírem um cookie de autenticação, que só é cadastrado após fazer login.

Dentro desse cookie está o JWT do usuário, nele está contido o ID do usuário, assim, podemos verificar se o JWT é válido e se o usuário existe.

Códigos (JS):

```
// LOGIN.TS

import { PrismaClient } from '@prisma/client'
import { NextResponse } from 'next/server'
import bcrypt from 'bcrypt'
import jwt from 'jsonwebtoken'
import ms from 'ms'
import z from 'zod'

import { AUTH_COOKIE_NAME } from '@/shared/constants'

const prisma = new PrismaClient()

const schema = z.object({
  email: z.string().email(),
  password: z.string(),
})
```

```
export async function POST(req: Request) {
  try {
    const data = await req.json()

    const validated = await schema.safeParseAsync(data)

    if (!validated.success) {
      return NextResponse.json(
        { error: validated.error, field: validated.error.errors[0].path[0] },
        { status: 400 },
      )
    }

    const user = await prisma.user.findFirst({
      where: {
        email: data.email,
      },
    })

    if (!user) {
      return NextResponse.json(
        { error: 'Não existe um usuário com esse e-mail', field: 'email' },
        { status: 404 },
      )
    }

    const passwordMatch = await bcrypt.compare(data.password, user.password_hash)

    if (!passwordMatch) {
      return NextResponse.json(
        { error: 'A senha está incorreta', field: 'password' },
        { status: 400 },
      )
    }

    const expiresIn = process.env.JWT_EXPIRES_IN || '1d'

    const token = jwt.sign({ id: user.id.toString() }, process.env.JWT_SECRET as string, {
      expiresIn: expiresIn,
    })

    const res = NextResponse.json({
      ...user,
      password_hash: undefined,
    })

    res.cookies.set(AUTH_COOKIE_NAME, token, {
      httpOnly: true,
      secure: true,
      path: '/',
      sameSite: 'strict',
      maxAge: ms(expiresIn) / 1000,
    })

    return res
```

```
  } catch (error) {
    console.log(error)

    return NextResponse.json({ error: (error as any).message }, { status: 500 })
  }
}
```

```
// VALIDATE-SESSION.TS
// Usado dentro dos endpoints protegidos para verificar se o usuário está logado.

import { AUTH_COOKIE_NAME } from '@/shared/constants'
import { NextRequest } from 'next/server'
import jwt from 'jsonwebtoken'

export function validateSession(req: NextRequest) {
  const token = req.cookies.get(AUTH_COOKIE_NAME)?.value

  if (!token) {
    return {
      error: {
        status: 401,
        message: 'Não autorizado',
      },
    }
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET!)

    return {
      data: decoded,
      success: true,
    }
  } catch (error) {
    return {
      error: {
        status: 401,
        message: 'Não autorizado',
      },
    }
  }
}
```

Esse códigos ficam dentro do servidor e não são acessíveis para o usuário, assim, ele não tem acesso ao secret do JWT nem aos hashs gerados pelo bcrypt

Parte documental

Tipo de criptografia

Usamos o JWT para salvar a sessão atual do usuário, pela simplicidade do código e da estrutura, assim, salvamos o JWT dentro de um cookie e podemos descriptografar ele para pegar o usuário logado.

Usamos também o Bcrypt para fazermos o hash das senhas geradas pelos usuários, assim, podemos salvar as senhas com a segurança de que caso os dados sejam vazados, ficará mais díficl para o invasor de achar o texto correspondente.

Dados sensíveis

Dentro do nosso site, salvamos os seguintes dados sensíveis:

Usuários

Nome	Tipo do dado	Descrição
ID	Texto	ID do usuário
Email	Texto	Email do usuário
Nome	Texto	Nome do usuário
Hash da senha	Texto	Hash gerado correspondente a senha do usuário

Tokens

Nome	Tipo do dado	Descrição
Token	Texto	O token que permite redefinir a senha do usuário, ele é usado dentro de “esqueci a senha”. Ele possui validade

Dados não sensíveis

Usuários

Nome	Tipo do dado	Descrição
Data de criação	Data	Data de criação do usuário
Data de atualização	Data	Data de atualização do usuário

Tokens

Nome	Tipo de dado	Descrição
Data de criação	Data	Data de criação do token
Data de validade	Data	Data de validade do token. Após passar dessa data, o token não é mais aceito, mas continua salvo dentro do

		nosso banco

Também temos salvo os dados do nosso modelo, são todos dados públicos retirados de portais como o CONAB ou o INMET. Vamos resumir aqui os principais:

Nome	Tipo de dado	Descrição
Ano	Número	Ano da safra
Produto	Texto	Cultura produzida
Estado	Texto	Estado de produção (SP, MG, etc.)
Área plantada	Número	Total de hectares plantados
Produção	Número	Total de toneladas produzidas
Produtividade	Número	Total de toneladas por hectares produzidas

Queries do banco de dados

Para fazer a conexão e chamadas para o banco de dados, usamos um ORM chamado Prisma. Assim, as queries do nosso banco de dados estão em JS puro, mas seguem a mesma lógica do SQL.

O prisma já trata os dados, escapa os caracteres, evita SQL injection, tudo para a gente.

Criação de usuário

```
const passwordHash = await bcrypt.hash(data.password, 10)

const user = await prisma.user.create({
  data: {
    email: data.email,
    name: data.name,
    password_hash: passwordHash,
  },
})
```

Procura por usuário (login)

```
const user = await prisma.user.findFirst({
  where: {
    email: data.email,
  },
})
```

```
// Removendo o password_hash antes de retornar os dados
return { ...user, password_hash: undefined }
```

Listagem de usuários

```
const users = await prisma.user.findMany()

// Removendo o password_hash antes de retornar os dados
return users.map((user) => ({ ...user, password_hash: undefined }))
```

Listagem com paginação dos dados

```
const query: Prisma.DataFindManyArgs = {
  where: {
    state: filters.state,
    product: filters.product,
    year: filters.year ? Number(filters.year) : undefined,
  },
  orderBy: [{ year: 'desc' }, { product: 'asc' }, { state: 'asc' }],
  take: Number(filters.limit),
  skip: Number(filters.limit) * (Number(filters.page) - 1),
}

const data = await prisma.data.findMany(query)
```