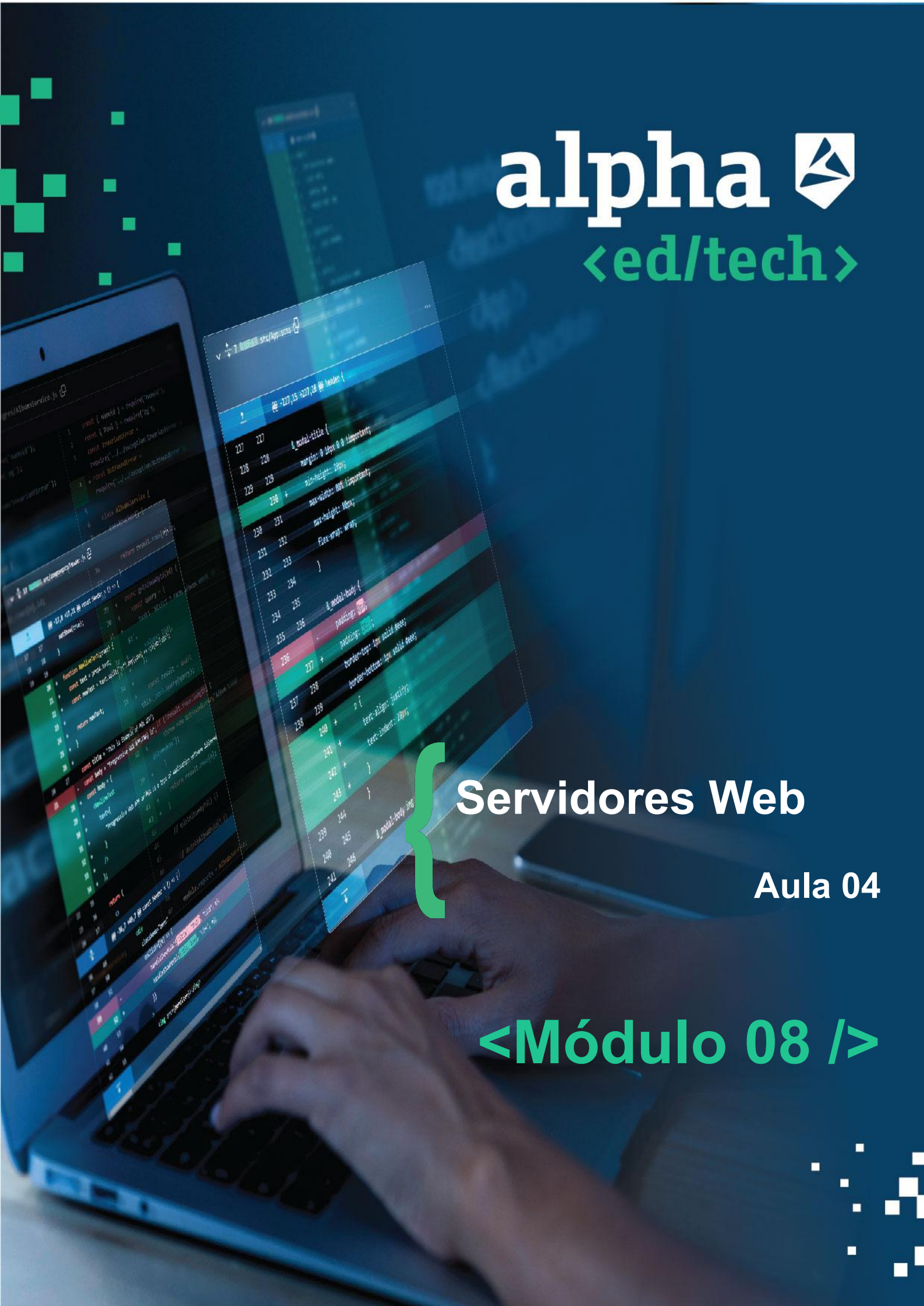

alpha
<ed/tech>
Servidores Web
Aula 04
<Módulo 08 />

Controle de sessão

Introdução

O conceito de sessão em aplicações web refere-se à capacidade de manter um estado persistente e contextualizado entre várias requisições do mesmo usuário durante uma interação com a aplicação.
Em uma sessão, informações específicas do usuário, como dados de autenticação, preferências, carrinho de compras, entre outros, podem ser retidas e acessadas em diferentes partes da aplicação.

Sessão e Requisições Independentes

Em uma arquitetura web tradicional, cada requisição enviada pelo navegador do usuário ao servidor é tratada como uma entidade independente.
Isso significa que, a princípio, o servidor não mantém informações sobre requisições anteriores.
As sessões vêm preencher essa lacuna, proporcionando um mecanismo para manter um contexto contínuo durante a interação do usuário com a aplicação.

Diferentemente das requisições independentes, onde o servidor não retém informações entre elas, uma sessão permite que dados específicos do usuário sejam armazenados e recuperados conforme necessário.
Isso é especialmente crucial para aplicações que necessitam rastrear o estado do usuário ao longo do tempo, como em processos de autenticação, personalização de conteúdo e carrinhos de compras em comércio eletrônico.

Necessidade e Vantagens

- **Persistência de Dados**: Manter uma sessão permite que dados críticos, como informações de autenticação, sejam retidos entre requisições, evitando a necessidade de autenticação a cada interação.

- **Personalização da Experiência do Usuário**: Informações sobre preferências do usuário, configurações de idioma, temas e outras personalizações podem ser armazenadas em uma sessão, proporcionando uma experiência mais personalizada.

- **Eficiência na Gestão do Estado do Aplicativo**: Ao manter um estado contínuo, o servidor pode gerenciar de forma eficiente o estado da aplicação, eliminando a necessidade de reprocessar dados a cada requisição.

- **Facilidade na Implementação de Recursos como Carrinhos de Compras**: Em aplicações de comércio eletrônico, a manutenção de um carrinho de compras persistente em uma sessão simplifica a implementação e oferece uma experiência de compra mais suave.

Autenticação vs. Autorização

- **Autenticação** é o processo pelo qual uma aplicação verifica a identidade de um usuário, garantindo que a pessoa que está tentando acessar recursos ou realizar ações é quem ela alega ser. Esse processo geralmente envolve a apresentação de credenciais, como nome de usuário e senha, ou métodos mais avançados, como autenticação de dois fatores (2FA) ou biometria. O objetivo principal da autenticação é estabelecer a confiança na identidade do usuário antes de conceder acesso.

- **Autorização**, por outro lado, trata do controle de acesso aos recursos e funcionalidades da aplicação após a autenticação ser bem-sucedida. Uma vez que a identidade do usuário é confirmada, a autorização determina quais ações ou partes da aplicação esse usuário específico tem permissão para acessar. Isso é geralmente feito atribuindo papéis, privilégios ou permissões específicas a um usuário ou a um grupo de usuários. A autorização é crucial para garantir que os usuários tenham apenas o acesso necessário, reduzindo assim os riscos de uso indevido ou acesso não autorizado.

Controle de sessão

Proteção contra Acessos Não Autorizados

A autenticação impede que usuários não autenticados obtenham acesso aos recursos da aplicação.
A autorização complementa esse processo, garantindo que mesmo usuários autenticados só possam acessar o que é apropriado ao seu papel ou nível de permissão.
A autorização é essencial para a gestão eficiente dos privilégios dos usuários.
A falta de autorização adequada pode resultar em usuários tendo mais acesso do que o necessário, aumentando os riscos de violações de segurança.
Em muitos setores, a autenticação e a autorização são requisitos para cumprir regulamentações de privacidade e segurança de dados.
A ausência de uma abordagem adequada pode resultar em penalidades legais e danos à reputação.
A combinação de autenticação e autorização ajuda a mitigar ameaças internas, limitando o acesso de usuários autenticados a apenas o necessário para suas funções.

Cookies

Cookies são pequenos fragmentos de dados armazenados no navegador do usuário como parte do protocolo HTTP.
Esses dados são enviados pelo servidor e armazenados no lado do cliente, permitindo a persistência de informações entre requisições.
Os cookies são amplamente utilizados para diversas finalidades, como rastreamento de sessões, personalização de experiência do usuário e armazenamento de preferências.
Um dos usos mais comuns dos cookies é o rastreamento de sessões em aplicações web.
Quando um usuário se autentica em uma aplicação, o servidor pode criar um cookie de sessão contendo um identificador único.
Esse cookie é enviado automaticamente pelo navegador em cada requisição subsequente, permitindo ao servidor reconhecer e associar a requisição à sessão específica desse usuário.
Além de rastrear sessões, os cookies são empregados para manter informações persistentes no navegador do cliente.
Isso pode incluir dados como preferências de idioma, configurações de tema, itens em um carrinho de compras e outros estados que precisam ser lembrados entre visitas do usuário.

Componentes Básicos de um Cookie

- **Nome e Valor**: Identificam o cookie e seu conteúdo.
- **Domínio e Caminho**: Especificam a quais domínios e caminhos o cookie pertence.
- **Data de Expiração**: Determina quando o cookie deve expirar e ser removido automaticamente.
- **Seguro e HTTPOnly**: Indicam se o cookie deve ser transmitido apenas por HTTPS e se é acessível somente via JavaScript, respectivamente.

Protocolo do Cookie

O cookie é enviado na requisição do cliente no cabeçalho (header) HTTP.
Mais especificamente, os cookies são incluídos no cabeçalho "Cookie" da requisição.

Quando o servidor envia um cookie para o navegador do cliente, ele inclui um cabeçalho "Set-Cookie".

```
Set-Cookie: username=kenji; Domain=alphaedtech.org.br; Path=/; Expires=Thu, 01 Jan 2025 00:00:00 GMT
```

Quando o cliente faz uma requisição subsequente para o mesmo domínio, ele incluirá automaticamente todos os cookies associados a esse domínio no cabeçalho "Cookie" da requisição:

```
GET /pagina HTTP/1.1
Host: alphaedtech.org.br
Cookie: username=kenji; outroCookie=valor
```

Controle de sessão

Segurança na Utilização de Cookies

Os cookies, por serem armazenados no navegador do usuário, podem apresentar alguns riscos de segurança se não forem utilizados corretamente.
Aqui estão algumas práticas essenciais para garantir a segurança ao lidar com cookies:

- **Atributo Secure:**

O atributo Secure indica que um cookie só deve ser transmitido sobre conexões HTTPS seguras. Isso protege os cookies contra interceptação por parte de terceiros durante a transmissão.

```
Set-Cookie: myCookie=myValue; Secure
```

- **Atributo HttpOnly:**

O atributo HttpOnly restringe o acesso do cookie apenas a solicitações HTTP e impede o acesso via JavaScript.
Isso ajuda a proteger contra ataques de script, como *Cross-Site Scripting (XSS)*, onde um invasor tenta injetar código malicioso no lado do cliente.

```
Set-Cookie: myCookie=myValue; HttpOnly
```

- **Configuração de Domínio e Caminho:**

Ao configurar o domínio e o caminho do cookie, é possível limitar a sua acessibilidade.
Definir o domínio e o caminho corretos evita que cookies sensíveis sejam acessados por páginas que não deveriam tê-los.

```
Set-Cookie: myCookie=myValue; Domain=example.com; Path=/app
```

- **Data de Expiração:**

Definir uma data de expiração para os cookies é crucial.
Isso evita que cookies sensíveis permaneçam no navegador do usuário indefinidamente, reduzindo o tempo de exposição a possíveis ataques.

```
Set-Cookie: myCookie=myValue; Expires=Wed, 21 Mar 2024 07:00:00 GMT
```

- **Segurança contra CSRF:**

Cookies de autenticação devem ser protegidos contra ataques *Cross-Site Request Forgery (CSRF)*.
Isso geralmente é feito incorporando tokens anti-CSRF no formulário e verificando-os no servidor.

```
Set-Cookie: csrfToken=uniqueToken; Secure; HttpOnly
```

Controle de sessão

Vulnerabilidades Comuns na gestão de Sessões

Cross-Site Scripting (XSS):
- **Descrição**: Ataque onde um invasor injeta scripts maliciosos no conteúdo da página, levando o navegador do usuário a executar código não autorizado.
- **Impacto na Sessão**: Pode resultar na captura de cookies de sessão e outras informações sensíveis.

Cross-Site Request Forgery (CSRF):
- **Descrição**: Ataque no qual um invasor induz o usuário a realizar ações indesejadas em uma aplicação na qual o usuário está autenticado.
- **Impacto na Sessão**: Pode causar a execução não autorizada de ações em nome do usuário autenticado.

Session Fixation:
- **Descrição**: Ataque onde um invasor define o ID da sessão antes mesmo do usuário autenticar-se.
- **Impacto na Sessão**: Permite que o invasor assuma a sessão do usuário autenticado.

Session Hijacking **(ou** *Session Sniffing* **):**
- **Descrição**: Interceptação de dados de sessão durante a transmissão não segura, geralmente em redes públicas.
- **Impacto na Sessão**: Possibilidade de obtenção de informações de autenticação.

Expiração de Sessão Inadequada:
- **Descrição**: Falha ao garantir que as sessões expiram após um período inativo ou após a saída do usuário.
- **Impacto na Sessão**: Aumenta o risco de acesso não autorizado se um dispositivo for deixado desatendido.

Práticas recomendadas

Utilizar HTTPS:
- **Descrição**: Transmitir dados de sessão apenas por meio de conexões seguras HTTPS.
- **Razão**: Minimiza a possibilidade de interceptação de dados durante a transmissão.

Configurar Atributos Seguros em Cookies:
- **Descrição**: Utilizar atributos como Secure e HttpOnly nos cookies.
- **Razão**: Secure garante que os cookies são transmitidos apenas por HTTPS, e HttpOnly protege contra ataques XSS.

Gerar IDs de Sessão Aleatórios e Únicos:
- **Descrição**: Utilizar IDs de sessão fortes e aleatórios.
- **Razão**: Dificulta ataques de adivinhação e força bruta.

Implementar Renovação de Sessão:
- **Descrição**: Renovar o ID da sessão após a autenticação ou em intervalos regulares.
- **Razão**: Reduz o risco de ataques de session fixation.

Expiração Adequada de Sessão:
- **Descrição**: Configurar tempos de expiração para sessões inativas e definir limites claros.
- **Razão**: Limita o tempo de exposição a ameaças caso a sessão não seja encerrada corretamente.

Validação e Escapamento de Dados:
- **Descrição**: Validar e escapar corretamente os dados antes de incluí-los em cookies ou em qualquer resposta ao cliente.
- **Razão**: Reduz o risco de ataques XSS.

Controle de sessão com ‘express’

Introdução

Os cookies desempenham um papel crucial no controle de sessão em aplicações web, e o framework Express facilita sua utilização.
Um cookie é uma pequena porção de dados armazenada no navegador do usuário, permitindo que as aplicações persistam informações entre requisições.
Com o Express, é possível manipular cookies de forma eficiente para rastrear sessões, armazenar preferências do usuário e fornecer uma experiência personalizada.

Configuração do middleware ‘cookie-parser’

O middleware ‘**cookie-parser**’ é uma ferramenta essencial para lidar com cookies em aplicações Express.
Para utilizá-lo, é necessário instalá-lo primeiro:

```
npm install cookie-parser --save
```

Abaixo, melhoramos nossa aplicação de API para usuários, criamos uma pasta ‘pages’ na qual colocamos duas páginas:

- **‘./pages/login’** : contém uma função a ser acessada pela rota ‘http://localhost:3000/’ e que contém a nossa página de login solicitando o nome do usuário e senha;
- **‘./pages/app’** : contém uma função que exibe a aplicação e verifica se possui um cookie de sessão válido.

```
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;
const cookieParser = require('cookie-parser');

// Middleware para analisar o corpo das requisições JSON
app.use(express.json());

// Middleware para lidar com cookies
app.use(cookieParser());

// Login Page
const loginPage = require('./pages/login');
app.get('/', loginPage);

// App Page
const appPage = require('./pages/app');
app.get('/app', appPage);

// Rotas
const routes = require('./routes');
app.use('/api', routes);

app.listen(port, () => {
  console.log(`Servidor rodando em http://localhost:${port}`);
});
```

Controle de sessão com ‘express’

‘./pages/login.js’

```
const loginContent = `
    <html>
    <head>
        <title>Login</title>
    </head>
    <body>
        <h1>Login</h1>
        <form action="/login" method="POST">
            <input type="text" name="username" placeholder="Usuário">
            <input type="password" name="password" placeholder="Senha">
            <button type="submit">Entrar</button>
        </form>
    </body>
    <script>
        const form = document.querySelector('form');
        form.addEventListener('submit', async (event) => {
            event.preventDefault();
            const username = form.username.value;
            const password = form.password.value;
            const response = await fetch('/api/login', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ username, password }),
            });
            const data = await response.json();
            if (data.error) {
                alert(data.error);
                document.cookie = "";
            } else {
                window.location.href = '/app';
            }
        });
    </script>
    </html>
`;

function loginPage(req, res) {
    res.setHeader('Content-Type', 'text/html');
    res.send(loginContent)
}

module.exports = loginPage;
```

Controle de sessão com ‘express’

‘./pages/app.js’

```
const appContent = `
    <html>
    <head>
        <title>Aplicação</title>
    </head>
    <body>
        <h1>Autenticado!</h1>
        <p>Seja bem-vindo à aplicação <span id="username"></span>!</p>
        <input id="exit" type="button" value="Sair" />
    </body>
    <script>
        document.addEventListener("DOMContentLoaded", async function() {
            const response = await fetch('/api/login', {
                method: 'GET',
                headers: {
                    'Content-Type': 'application/json',
                },
            });
            const data = await response.json();
            if (data.error) {
                alert(data.error);
                document.cookie = "";
                window.location.href = "/";
            } else {
                document.getElementById('username').textContent = data.name;
            }
            const exitButton = document.querySelector('#exit');
            exitButton.onclick = function() {
                document.cookie = "";
                window.location.href = "/";
            }
        });
    </script>
    </html>
`;

function appPage(req, res) {
    res.setHeader('Content-Type', 'text/html');
    res.send(appContent)
}

module.exports = appPage;
```

Controle de sessão com ‘express’

‘./routes/index.js’

Para que nossa aplicação possa funcionar, foi adicionada uma rota para a api realizar a autenticação do usuário.
Esta rota foi chamada de ‘login’.

```
1   // src/routes/index.js
2   const express = require('express');
3   const router = express.Router();
4   const userRoutes = require('./userRoutes');
5   const loginRoutes = require('./loginRoutes');
6
7   router.use('/users', userRoutes);
8   router.use('/login', loginRoutes);
9
10  module.exports = router;
```

‘./routes/loginRoutes.js’

Contém, neste caso apenas duas rotas:

- **GET ‘/api/login/’** : obtém o nome do usuário e seu nome completo;
- **POST ‘/api/login’** : recebe os dados do usuário e, se autenticado, retorna o cookie de sessão

```
1   // src/routes/userRoutes.js
2   const express = require('express');
3   const router = express.Router();
4   const loginController = require('../controllers/loginController');
5
6   router.get('/', loginController.getLogin);
7   router.post('/', loginController.autenticate);
8
9   module.exports = router;
```

Controle de sessão com ‘express’

‘./controllers/loginController.js’

Na linha 1 foi importada a biblioteca ‘crypto’ responsável pela geração de um ‘hash’ responsável para identificação do usuário em futuras conexões.

A variável ‘loginUsers’ neste caso representa os dados dos usuários que possuem acessos ao sistema. Representado por um array com dados básicos como o nome do usuário ‘username’, o seu nome completo ‘name’, a sua senha ‘password’ (neste caso está em plain text e o ideal seria o armazenamento criptografado), os tipos de permissões que o usuário possui ‘userType’ e a sessão utilizada que o identificará em futuras conexões.

```
1   const crypto = require("crypto");
2
3   let loginUsers = [
4       {
5           "username": "kenji",
6           "name": "Kenji Taniguchi",
7           "password": "123456",
8           "userType": [ "admin" ],
9           "sessionToken": "",
10      },
11      {
12          "username": "samir",
13          "name" : "Samir",
14          "password" : "654321",
15          "userType": [ "user" ],
16          "sessionToken": "",
17      }
18  ];
19
20  const getLogin = (req, res) => {
21      const sessionToken = req.cookies.session_id;
22
23      if (sessionToken) {
24          const foundUser = loginUsers.filter(user => user.sessionToken === sessionToken);
25          if (foundUser.length > 0) {
26              return res.json({ username: foundUser[0].username, name : foundUser[0].name });
27          }
28      }
29      const error = "Falha ao tentar obter dados para o usuário!";
30      res.cookie('session_id', '', { expires: new Date(0) });
31      res.json({error});
32  };
```

A função ‘getLogin’ verifica antes se existe sessionToken obtido pela análise de ‘req.cookies.session_id’ e, caso exista, procura na lista de usuários se ele está cadastrado e retorna se positivo, um objeto com o ‘username’ e ‘name’.
Caso contrário, retorna um objeto com uma mensagem de erro e define a expiração do ‘session_id’ de forma a tentar limpar do navegador do usuário.

Controle de sessão com ‘express’

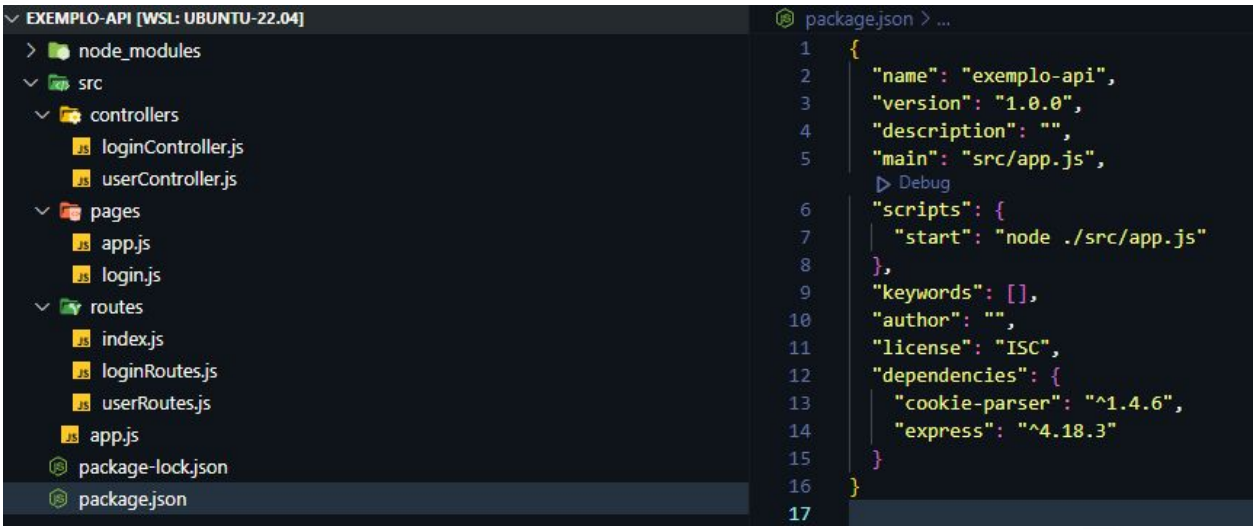
‘./controllers/loginController.js’ (continuação

A função ‘autenticate’ é chamada por uma chamada POST a ‘/api/login’ e verifica se os dados são válidos inicialmente (linhas 37 a 41) e depois se o usuário e senha são válidos (linhas 44 a 48).
Caso não sejam válidos, retorna um objeto que contém um ‘error’.
No caso de usuário encontrado, obtém-se o seu ‘username’ (linha 51).
Cria-se um ‘hash’ baseado no ‘timestamp’ (valor em milissegundos da data atual) (linha 55).
Adiciona-se o valor do ‘hash’ em um atributo ‘sessionToken’ ao usuário que foi autenticado. (linhas 57 e 58).
Define-se o ‘cookie’ chamado ‘session_id’ que será armazenado no navegador do usuário.

```
34  const autenticate = (req, res) => {
35      const { username, password } = req.body;
36
37      const error = "Usuário e/ou senha inválidos!";
38      if (!username || !password) {
39          res.cookie('session_id', '', { expires: new Date(0) });
40          return res.status(400).json({ error });
41      }
42
43      // Procura a existência de um usuário e senha em loginUsers
44      const foundUser = loginUsers.filter(user => user.username === username && user.password === password);
45      if (foundUser.length === 0) {
46          res.cookie('session_id', '', { expires: new Date(0) });
47          return res.status(400).json({ error });
48      }
49
50      // Usuário encontrado!
51      const foundUsername = foundUser[0].username;
52
53      // Criando e associando um sessionToken ao usuário
54      const timestamp = Date.now();
55      const sessionToken = crypto.createHash("sha256").update(`${timestamp}`).digest("hex");
56
57      const index = loginUsers.findIndex(user => user.username === foundUsername);
58      loginUsers[index].sessionToken = sessionToken;
59
60      res.cookie('session_id', sessionToken, { maxAge: 900000, httpOnly: true });
61      res.status(200).json({ success: true });
62  };
63
64  module.exports = {
65      getLogin,
66      autenticate,
67  };
```

Controle de sessão com ‘express’

Estrutura de pastas



Observações importantes

A apresentação deste exemplo de criação de estrutura de pastas, rotas, controllers, páginas, armazenamento de sessões entre outras, trata-se de uma sugestão.

Há diversas formas de organizar um projeto em nodejs e fica a seu critério identificar qual a melhor forma de fazê-lo, ok?

Também, neste exemplo não há verificações ‘fortes’ de estruturas de chamadas e retornos da API que, em produção demandariam maiores cuidados.

O uso de arrays para dados servem para simplificar o entendimento. Porém, os dados usualmente são armazenados em banco de dados (relacionais e não relacionais) dependendo do seu projeto.

Controle de sessão com JWT

Introdução

JSON Web Token (JWT) é um padrão aberto (RFC 7519) para a criação de toke
ns de acesso que podem ser usados para autenticação e troca de informações entre partes confiáveis. Ele é baseado em JSON (*JavaScript Object Notation*), o que o torna fácil de ler e escrever para humanos e de ser interpretado por máquinas.
JWTs são frequentemente utilizados em ambientes web e APIs RESTful para autenticar usuários e autorizar acesso a recursos protegidos.
Eles oferecem uma maneira segura e eficiente de transmitir informações de forma compacta e autenticada entre duas partes.

Estrutura de um token JWT

Um **token JWT** é composto por três partes, separadas por pontos (.), representando respectivamente o cabeçalho (header), o payload e a assinatura.

- ***Header*** (Cabeçalho): Contém informações sobre o tipo do token e o algoritmo de assinatura utilizado para gerar a assinatura. Geralmente, o cabeçalho é representado como um JSON codificado em **Base64**.

- ***Payload*** (Carga Útil): Contém os dados (claims) que deseja-se transmitir, como informações do usuário ou outros metadados relevantes. O payload também é codificado em JSON e codificado em Base64.

- ***Signature*** (Assinatura): É a parte final do token e é usada para verificar a integridade do token e garantir que ele não foi adulterado durante a transmissão. A assinatura é gerada a partir do cabeçalho e do payload, combinados com uma chave secreta, usando o algoritmo especificado no cabeçalho.

Benefícios

- **Simplicidade**: JWTs são fáceis de entender e implementar devido ao seu formato baseado em JSON.

- **Autenticação sem estado (*Stateless*):** Como os tokens JWT contêm todas as informações necessárias para autenticar um usuário, não é necessário armazenar informações de sessão no servidor. Isso torna a autenticação mais escalável e distribuída.

- **Segurança**: JWTs podem ser assinados digitalmente, garantindo que os dados não tenham sido alterados durante a transmissão. Isso garante a integridade dos dados transmitidos.

- **Interoperabilidade**: JWTs são amplamente suportados por várias linguagens de programação e frameworks, tornando-os ideais para comunicação entre sistemas heterogêneos.

Casos de Uso

JWTs são comumente usados para autenticação e autorização em aplicações web e APIs RESTful.
Eles também são úteis para comunicação entre microsserviços e sistemas distribuídos, bem como para implementar single sign-on (SSO) e autenticação federada.

Controle de sessão com JWT

Base64

Introdução

A **Base64** é um método de codificação que converte dados binários em uma representação textual composta por um conjunto de caracteres ASCII.
Esse método é usado principalmente para transmitir dados binários através de meios que lidam apenas com texto, como por exemplo, em sistemas de e-mail ou na transferência de arquivos via HTTP.

Estrutura

A **Base64** é formada por um conjunto de 64 caracteres ASCII, geralmente consistindo nas letras maiúsculas e minúsculas do alfabeto inglês (A-Z, a-z), os números de 0 a 9 e dois caracteres especiais, que podem variar dependendo da implementação.
Esses 64 caracteres são utilizados para representar os 256 valores possíveis de um byte (8 bits).

Codificação

Para converter dados binários em Base64, o processo é relativamente simples:

- Os dados binários são divididos em grupos de 6 bits.
- Cada grupo de 6 bits é então convertido em um caractere da tabela BASE64, usando uma correspondência predefinida.
- Se o número de bits no último grupo não for suficiente para formar um grupo completo de 6 bits, zeros são adicionados à direita para completar o grupo.
- O caractere de preenchimento "=" é adicionado ao final, se necessário, para garantir que o comprimento do texto codificado seja um múltiplo de 4.

```
// Dados que você deseja codificar
const dados = 'Olá, mundo!';

// Codifica os dados em BASE64
const dadosCodificados = Buffer.from(dados).toString('base64');

console.log('Dados codificados em BASE64:', dadosCodificados);
```

Decodificação

Para decodificar dados Base, o processo é reverso:

- Os caracteres Base64 são convertidos de volta em grupos de 6 bits.
- Cada grupo de 6 bits é então combinado para formar os bytes originais.
- O processo continua até que todos os caracteres Base64 tenham sido convertidos de volta para bytes.

```
// Dados codificados em BASE64
const dadosCodificados = 'T2zDoSwgbXVuZG8h';

// Decodifica os dados BASE64
const dadosDecodificados = Buffer.from(dadosCodificados, 'base64').toString('utf-8');

console.log('Dados decodificados:', dadosDecodificados);
```

Casos de Uso

A codificação Base64 é usada em muitas situações onde é necessário representar dados binários em formato de texto.
Isso inclui anexos de e-mail, armazenamento de dados binários em bancos de dados que aceitam apenas texto, e em URLs quando caracteres especiais não são permitidos.

Controle de sessão com JWT

Implementando a Geração de Tokens JWT

Vamos aprender a implementar a geração de tokens JWT em um servidor Node.js.
Isso nos permitirá criar tokens JWT que podem ser utilizados para autenticar usuários em nossa aplicação.

Iniciando o projeto

```
kenji@DESKTOP-0EELV37:~$ pwd
/home/kenji
kenji@DESKTOP-0EELV37:~$ mkdir exemplo-jwt
kenji@DESKTOP-0EELV37:~$ cd exemplo-jwt
kenji@DESKTOP-0EELV37:~/exemplo-jwt$ npm init -y
Wrote to /home/kenji/exemplo-jwt/package.json:

{
  "name": "exemplo-jwt",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


kenji@DESKTOP-0EELV37:~/exemplo-jwt$ npm install express --save

added 64 packages, and audited 65 packages in 3s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
kenji@DESKTOP-0EELV37:~/exemplo-jwt$ npm install jsonwebtoken

added 16 packages, and audited 81 packages in 3s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
kenji@DESKTOP-0EELV37:~/exemplo-jwt$ code .
```

- Crie uma pasta: 'mkdir exemplo-jwt'
- Acesse a pasta: 'cd exemplo-jwt'
- Inicie um projeto: 'npm init –y'
- Instale o pacote do express e salve os dados: 'npm install express –save'
- Instale o pacote do jsonwebtoken e salve os dados: 'npm install jsonwebtoken –save'
- Abra o vscode: 'code .'
- Instale também a biblioteca cookie-parser: 'npm install cookie-parser –save'

```
kenji@DESKTOP-0EELV37:~/exemplo-jwt$ npm install cookie-parser --save

added 2 packages, and audited 83 packages in 482ms

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Controle de sessão com JWT

package.json

Após instalação das bibliotecas, o arquivo package.json foi ajustado como abaixo:

```
package.json > ...
{
  "name": "exemplo-jwt",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  ▷ Debug
  "scripts": {
    "start": "node ./src/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cookie-parser": "^1.4.6",
    "express": "^4.18.3",
    "jsonwebtoken": "^9.0.2"
  }
}
```

app.js

```
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;
const cookieParser = require('cookie-parser');

// Middleware para analisar o corpo das requisições JSON
app.use(express.json());

// Middleware para lidar com cookies
app.use(cookieParser());

// Login Page
const loginPage = require('./pages/login');
app.get('/', loginPage);

// App Page
const appPage = require('./pages/app');
app.get('/app', appPage);

// Rotas
const routes = require('./routes');
app.use('/api', routes);

app.listen(port, () => {
    console.log(`Servidor rodando em http://localhost:${port}`);
});
```

Controle de sessão com JWT

./pages/app.js

src > pages > app.js > ...

```
const appContent = `
    <html>
    <head>
        <title>Aplicação</title>
    </head>
    <body>
        <h1>Autenticado!</h1>
        <p>Seja bem-vindo à aplicação <span id="username"></span>!</p>
        <input id="exit" type="button" value="Sair" />
    </body>
    <script>
        document.addEventListener("DOMContentLoaded", async function() {
            const response = await fetch('/api/login', {
                method: 'GET',
                headers: {
                    'Content-Type': 'application/json',
                },
            });
            const data = await response.json();
            if (data.error) {
                alert(data.error);
                document.cookie = "";
                window.location.href = "/";
            } else {
                document.getElementById('username').textContent = data.name;
            }
            const exitButton = document.querySelector('#exit');
            exitButton.onclick = function() {
                document.cookie = "";
                window.location.href = "/";
            }
        });
    </script>
    </html>
`;

function appPage(req, res) {
    res.setHeader('Content-Type', 'text/html');
    res.send(appContent)
}

module.exports = appPage;
```

Controle de sessão com JWT

./pages/login.js

src > pages > login.js > ...

```
const loginContent = `
    <html>
    <head>
        <title>Login</title>
    </head>
    <body>
        <h1>Login</h1>
        <form action="/login" method="POST">
            <input type="text" name="username" placeholder="Usuário">
            <input type="password" name="password" placeholder="Senha">
            <button type="submit">Entrar</button>
        </form>
    </body>
    <script>
        const form = document.querySelector('form');
        form.addEventListener('submit', async (event) => {
            event.preventDefault();
            const username = form.username.value;
            const password = form.password.value;
            const response = await fetch('/api/login', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({ username, password }),
            });
            const data = await response.json();
            if (data.error) {
                alert(data.error);
                document.cookie = "";
            } else {
                window.location.href = '/app';
            }
        });
    </script>
    </html>
`;

function loginPage(req, res) {
    res.setHeader('Content-Type', 'text/html');
    res.send(loginContent)
}

module.exports = loginPage;
```

Controle de sessão com JWT

./routes/index.js

```
1   // src/routes/index.js
2   const express = require('express');
3   const router = express.Router();
4   const userRoutes = require('./userRoutes');
5   const loginRoutes = require('./loginRoutes');
6
7   router.use('/users', userRoutes);
8   router.use('/login', loginRoutes);
9
10  module.exports = router;
```

./routes/loginRouter.js

Aqui começam as diferenças da implementação da solução da aula passada!
Neste caso foi criado um 'middleware' chamado 'permissionVerify' que cuidará de verificar se há um 'session_id' no cookie da requisição.
Veja que este middleware foi colocado apenas na rota GET de '/', isto é, para esta rota e método, deve-se aplicar o 'permissionVerify' middleware.

```
src > routes > JS loginRoutes.js > ...
1   // src/routes/userRoutes.js
2   const express = require('express');
3   const router = express.Router();
4   const loginController = require('../controllers/loginController');
5   const permissionVerify = require('./permissionVerify');
6
7   router.get('/', permissionVerify, loginController.getLogin);
8   router.post('/', loginController.autenticate);
9
10  module.exports = router;
```

Já, a rota POST '/' não passa pela verificação de permissão pois qualquer usuário pode enviar os dados de 'username' e 'password' para poder receber o cookie 'session_id'.

Controle de sessão com JWT

./routes/permissionVerify.js

Este é o arquivo do middleware.
Inicia-se com o carregamento da constante 'jwt' com a biblioteca anteriormente instalada 'jsonwebtoken'
Obteve-se o valor de 'SECRET_KEY' do arquivo em '../config/index.js' que contém uma string que deve ser mantida em segredo pois qualquer pessoa que tiver a 'SECRET_KEY' poderá gerar sessionToken que seriam considerados válidos, por isso, mantenha esta chave em segurança!

Como a função deste middleware é verificar se há o cookie 'session_id', caso este seja inexistente, retorna erro de 'Token JWT ausente'.

Verifica-se então se o 'session_id' que está em 'sessionToken', mais a 'SECRET_KEY' é válida.
Caso seja válida, adiciona-se o valor decodificado do atributo 'user' do sessionToken decodificado.
Caso contrário retorna um JSON com erro 'Token JWT inválido!'.

src > routes > permissionVerify.js > ...

```
1   const { SECRET_KEY } = require("../config");
2   const jwt = require('jsonwebtoken');
3
4   function permissionVerify(req, res, next) {
5     // Verifica se o cookie de session_id está presente na requisição
6     const sessionToken = req.cookies.session_id;
7
8     if (!sessionToken) {
9       return res.status(401).json({ error: 'Token JWT ausente' });
10    }
11
12    // Verifica e decodifica o token JWT
13    jwt.verify(sessionToken, SECRET_KEY, (err, decoded) => {
14      if (err) {
15          return res.status(403).json({ error: 'Token JWT inválido' });
16      } else {
17          // O token é válido, podemos acessar as informações decodificadas
18          req.user = decoded.user; // Armazena as informações do usuário decodificadas no objeto req
19          next(); // Passa a requisição para o próximo middleware ou rota
20      }
21    });
22  }
23
24  module.exports = permissionVerify;
```

./config/index.js

src > config > index.js > ...

```
1   const config = {
2       SECRET_KEY: "esta_eh_uma_chave_que_deve_ser_mantida_em_segredo"
3   }
4
5   module.exports = config;
```

Controle de sessão com JWT

./controllers/loginController.js

src > controllers > loginController.js > ...

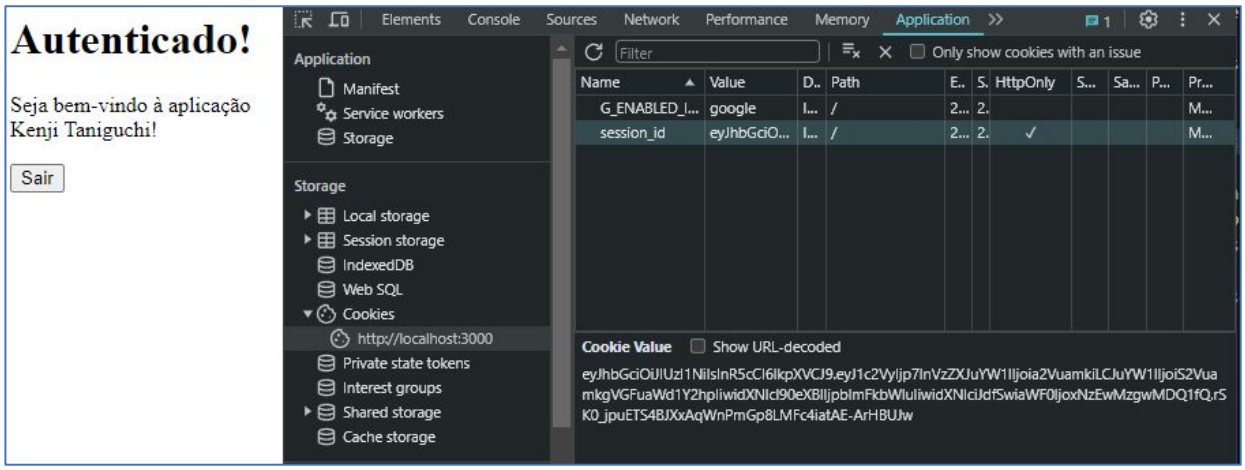
```
const { SECRET_KEY } = require("../config");
const jwt = require('jsonwebtoken');

let loginUsers = [
    {
        "username": "kenji",
        "name": "Kenji Taniguchi",
        "password": "123456",
        "userType": [ "admin", "user" ],
    },
    {
        "username": "samir",
        "name" : "Samir",
        "password" : "654321",
        "userType": [ "user" ],
    }
];

const getLogin = async (req, res) => {
    const user = req.user;
    return res.json(user);
};

const autenticate = async (req, res) => {
    const { username, password } = req.body;

    const error = "Usuário e/ou senha inválidos!";
    if (!username || !password) {
        res.cookie('session_id', '', { expires: new Date(0) });
        return res.status(400).json({ error });
    }

    // Procura a existência de um usuário e senha em loginUsers
    const foundUser = loginUsers.filter(user => user.username === username && user.password === password);
    if (foundUser.length === 0) {
        res.cookie('session_id', '', { expires: new Date(0) });
        return res.status(400).json({ error });
    }

    // Usuário encontrado!
    const user = {
        username: foundUser[0].username,
        name: foundUser[0].name,
        user_type: foundUser[0].userType,
    };

    // Gerando token JWT com informações personalizadas e enviando como cookie session_id
    try {
        const sessionToken = await jwt.sign({ user }, SECRET_KEY);
        res.cookie('session_id', sessionToken, { maxAge: 900000, httpOnly: true });
        res.json({ success: true });
    } catch (err) {
        res.status(500).json({ error: 'Erro ao gerar token JWT' });
    }

};

module.exports = {
    getLogin,
    autenticate,
};
```

Controle de sessão com JWT

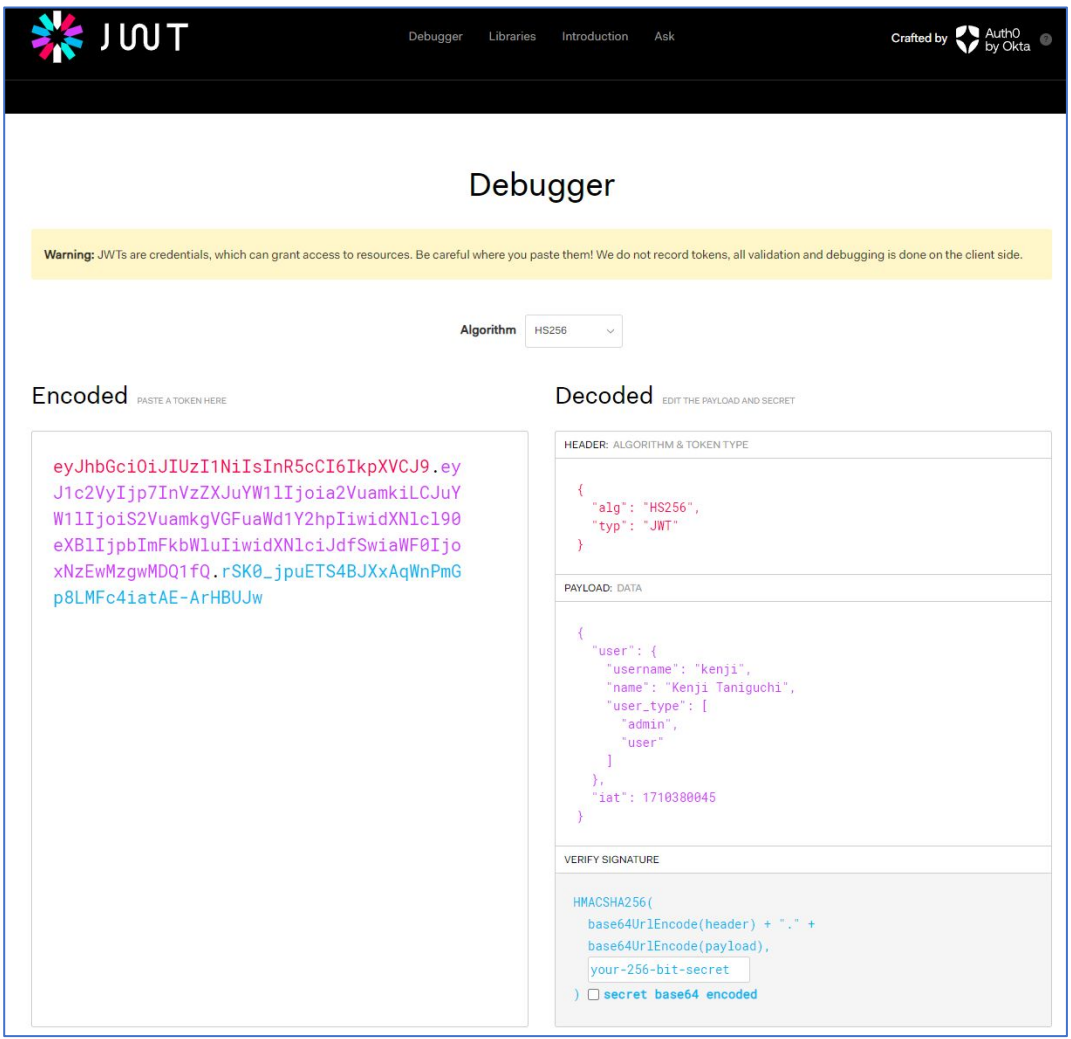
./controllers/loginController.js

Note que o '**getLogin**' agora obtém apenas o 'req.user' que foi passado pelo middleware e devolve ao usuário a sua identificação já decodificada.

Também, o '**autenticate**' agora não precisa ficar armazenando o valor junto aos dados, apenas devolve um cookie 'session_id' com o valor do 'sessionToken' gerado pelo 'jwt.sign' que recebe como parâmetros o objeto contendo o 'user' que neste caso é um objeto com os valores que serão convertidos em base64 e a chave criptográfica.



Veja na imagem acima que o cookie de 'session_id' contém um conjunto de letras e números.
Para saber o conteúdo, basta visitar o site **jwt.io**



Controle de sessão com JWT

./routes/userRoutes.js

Este arquivo, diferentemente do 'loginRoutes.js' inclui o middleware em 'TODAS' as rotas utilizando-se o '**router.use(permissionVerify)**'.

src > routes > userRoutes.js > ...

```
// src/routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
const permissionVerify = require('./permissionVerify');

// Aplica o middleware permissionVerify em todas as rotas definidas em userRoutes.js
router.use(permissionVerify);

router.get('/', userController.getAllUsers);
router.post('/', userController.createUser);
router.put('/:id', userController.updateUser);
router.delete('/:id', userController.deleteUser);

module.exports = router;
```

./controllers/userController.js

src > controllers > userController.js > ...

```
let users = [
    { id: 1, username: 'usuario1', email: 'usuario1@example.com' },
    { id: 2, username: 'usuario2', email: 'usuario2@example.com' },
    // Adicione outros usuários conforme necessário
];

let nextUserId = users.length + 1;

const error = "Sem permissão para realizar esta requisição!";

const getAllUsers = (req, res) => {
    const admin = req.user.user_type.includes("admin");
    if (!admin) {
        res.status(403).json({error})
    }
    res.json(users);
};

const createUser = (req, res) => {
    const admin = req.user.user_type.includes("admin");
    if (!admin) {
        res.status(403).json({error})
    }

    const { username, email } = req.body;
    // Validação simples
    if (!username || !email) {
        return res.status(400).json({ error: 'O username e o email são obrigatórios' });
    }

    const newUser = { id: nextUserId++, username, email };
    users.push(newUser);
    res.status(201).json(newUser);
};
```

Controle de sessão com JWT

./controllers/userController.js

Verifique que todas as rotas fazem análise se o usuário é do tipo 'admin' e só permitem o acesso se o usuário tiver 'admin' como 'userType'.
Neste exemplo de dois usuários, 'kenji' e 'samir', apenas 'kenji' é do tipo 'admin'. Assim, o usuário 'kenji' conseguirá acessar o endpoint para 'user' mas o usuário 'samir' não.
Obviamente, como passa pelo middleware, um usuário sem um jwt válido já não tem acesso a estas rotas.

```
35
36  const updateUser = (req, res) => {
37      const admin = req.user.user_type.includes("admin");
38      if (!admin) {
39          res.status(403).json({error})
40      }
41
42      const userId = parseInt(req.params.id);
43      const { username, email } = req.body;
44
45      const userIndex = users.findIndex((user) => user.id === userId);
46
47      // Verifica se o usuário existe
48      if (userIndex === -1) {
49          return res.status(404).json({ error: 'Usuário não encontrado' });
50      }
51
52      // Atualiza os dados do usuário
53      users[userIndex] = { ...users[userIndex], username, email };
54      res.json({ message: `Usuário com ID ${userId} atualizado` });
55  };
56
57  const deleteUser = (req, res) => {
58      const admin = req.user.user_type.includes("admin");
59      if (!admin) {
60          res.status(403).json({error})
61      }
62
63      const userId = parseInt(req.params.id);
64
65      users = users.filter((user) => user.id !== userId);
66
67      res.json({ message: `Usuário com ID ${userId} excluído` });
68  };
69
70  module.exports = {
71      getAllUsers,
72      createUser,
73      updateUser,
74      deleteUser,
75  };
```

Controle de sessão com JWT

Estrutura de pastas

Para entender a estrutura de pastas criadas, segue a imagem abaixo: