## Questões Teóricas

1. **Explique o que é JWT e quais são suas vantagens.**

JWT (JSON Web Token) é um padrão aberto para transmissão segura de informações entre duas partes como um objeto JSON. Ele é amplamente utilizado para autenticação e autorização em aplicações web. Um JWT é composto por três partes codificadas em Base64: o header, o payload e a assinatura. Suas vantagens incluem ser stateless (não exige armazenamento de sessão no servidor), fácil de transportar entre cliente e servidor (via headers, por exemplo), e permitir a verificação da integridade e autenticidade dos dados por meio da assinatura digital.

2. **Qual a diferença entre sessões e cookies?**

Sessões armazenam informações do usuário no servidor, enquanto cookies armazenam dados no navegador do cliente. As sessões são mais seguras, pois o conteúdo fica no servidor, sendo que o cliente apenas guarda um identificador. Já os cookies podem conter informações acessíveis via JavaScript e são enviados automaticamente em todas as requisições para o domínio. Enquanto as sessões expiram conforme configurado no servidor, os cookies podem ter um tempo de vida definido no lado do cliente.

3. **Defina CORS e por que é importante configurá-lo corretamente.**

CORS (Cross-Origin Resource Sharing) é um mecanismo de segurança que controla como os recursos de um servidor podem ser acessados por páginas web de origens diferentes. Ele é essencial para evitar que aplicações maliciosas hospedadas em outros domínios façam requisições indevidas para APIs ou backends. Configurar o CORS corretamente garante que apenas origens confiáveis possam acessar os recursos, ajudando a evitar vazamentos de dados ou ações não autorizadas.

4. **O que é CSRF e como prevenir este tipo de ataque?**

CSRF (Cross-Site Request Forgery) é um tipo de ataque em que um usuário autenticado é induzido a executar ações indesejadas em uma aplicação web, sem seu conhecimento, geralmente através de um link ou formulário malicioso. Para prevenir esse tipo de ataque, é comum o uso de tokens CSRF únicos por sessão, adicionados a formulários e verificados no servidor. Outras medidas incluem o uso do atributo SameSite nos cookies e exigir autenticação adicional para ações sensíveis.

5. **Explique o conceito de SQL Injection e seus riscos.**

SQL Injection é uma técnica de ataque que explora vulnerabilidades em instruções SQL, permitindo que um invasor injete comandos maliciosos no banco de dados. Isso pode levar ao vazamento, alteração ou exclusão de dados, e até ao comprometimento total do sistema. Os riscos incluem acesso não autorizado a informações sensíveis, destruição de dados, e controle administrativo sobre a aplicação.

6. **Como o atributo HttpOnly ajuda a proteger cookies?**

O atributo HttpOnly, quando adicionado a um cookie, impede que ele seja acessado via JavaScript no navegador. Isso ajuda a proteger os cookies contra ataques de cross-site scripting (XSS), evitando que scripts maliciosos roubem informações de sessão armazenadas nos cookies.

7. **Quais são as principais partes de um token JWT?**

Um token JWT possui três partes: o header, que especifica o tipo de token e o algoritmo de assinatura; o payload, que contém as informações (claims) a serem transmitidas, como ID do usuário e permissões; e a signature, que é a assinatura digital usada para verificar a integridade e autenticidade do token.

8. **Liste três boas práticas para evitar SQL Injection em aplicações Node.js.**

As boas práticas incluem o uso de queries parametrizadas ou prepared statements, evitando concatenar strings diretamente nas instruções SQL; a validação e sanitização rigorosa dos dados de entrada; e o uso de bibliotecas de ORM ou query builders confiáveis, como Sequelize ou Knex, que abstraem a construção das consultas de forma segura.

9. **O que é autenticação baseada em roles e quais são seus benefícios?**

Autenticação baseada em roles (RBAC – Role-Based Access Control) é um modelo de controle de acesso no qual permissões são atribuídas a diferentes perfis ou funções de usuários, como "admin", "editor" ou "leitor". Os benefícios incluem a centralização da gestão de permissões, facilidade de manutenção, maior segurança e maior clareza sobre quem pode fazer o quê na aplicação.

**10. Como o middleware csrf auxilia na proteção de aplicações web?**

O middleware csrf fornece automaticamente tokens CSRF únicos e os associa a sessões ou requisições. Ele garante que, para uma ação ser executada, o token presente no formulário ou cabeçalho da requisição seja válido, protegendo a aplicação contra ataques em que comandos maliciosos são enviados de forma automatizada ou de sites terceiros tentando se passar por requisições legítimas do usuário.

## Questões Teóricas

**11. Configure um servidor Express para gerar e validar tokens JWT.**

Primeiro foi criado um servidor utilizando o framework Express.js e configurado o middleware body-parser para interpretar requisições com dados em formato JSON. Em seguida, implementou-se uma rota de login que valida credenciais fornecidas pelo usuário e, em caso de sucesso, gera um token JWT com um payload contendo informações básicas e prazo de expiração. Esse token é assinado com uma chave secreta e retornado ao cliente. Para proteger recursos sensíveis, foi criado um middleware que extrai o token do cabeçalho da requisição, verifica sua validade por meio da função jwt.verify e, se for válido, permite o acesso à rota protegida; caso contrário, bloqueia a requisição com a resposta adequada.

**Arquivo:**

index.js

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;
const SECRET_KEY = 'minha_chave_secreta_super_segura';
app.use(bodyParser.json());

app.post('/login', (req, res) => {
  const { username, password } = req.body;
```

```
  if (username === 'admin' && password === 'senha123') {
    // Criação do payload do token
    const payload = {
      username: username,
      role: 'admin'
    };
    const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
    return res.json({ token });
  }

  res.status(401).json({ error: 'Credenciais inválidas' });
});

function autenticarToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Formato: Bearer
token

  if (!token) {
    return res.status(401).json({ error: 'Token não fornecido' });
  }

  jwt.verify(token, SECRET_KEY, (err, usuario) => {
    if (err) {
      return res.status(403).json({ error: 'Token inválido ou expirado' });
    }

    req.usuario = usuario;
    next();
  });
}

// Rota protegida
app.get('/protegido', autenticarToken, (req, res) => {
```

```
  res.json({ mensagem: 'Acesso concedido à rota protegida!', usuario:
req.usuario });
});

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

**12. Implemente um sistema de sessão que armazene informações temporárias de um usuário.**

Para a implementação do sistema de sessões, utilizou-se o framework Express.js juntamente com o middleware express-session, que permite armazenar informações temporárias de usuários por meio de cookies de sessão. Inicialmente, o servidor foi configurado para interpretar requisições JSON e manter sessões com uma chave secreta, definindo um tempo de expiração para os cookies. Em seguida, foi criada uma rota de login que, ao receber um nome de usuário, armazena essas informações na sessão ativa do cliente. Também foi implementada uma rota de verificação que retorna os dados da sessão caso o usuário esteja autenticado, além de uma rota de logout responsável por destruir a sessão e invalidar o cookie correspondente. Esse fluxo permite a manutenção de estado entre requisições HTTP, garantindo que os dados temporários do usuário fiquem acessíveis durante o período de validade da sessão.

**Arquivo:**

index.js

```
const express = require('express');

const session = require('express-session');

const bodyParser = require('body-parser');


const app = express();

const PORT = 3000;
```

```
app.use(bodyParser.json());

app.use(session({
  secret: 'chave_secreta_para_assinar_cookie', // use variável de ambiente em produção
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 60000 } // 1 minuto de duração
}));

app.post('/login', (req, res) => {
  const { username } = req.body;

  if (!username) {
    return res.status(400).json({ error: 'Usuário não fornecido' });
  }

  req.session.usuario = { username };
  res.json({ mensagem: `Usuário ${username} logado com sucesso.` });
});

app.get('/perfil', (req, res) => {
  if (req.session.usuario) {
    res.json({ mensagem: 'Sessão ativa', usuario: req.session.usuario });
  } else {
```

```
    res.status(401).json({ mensagem: 'Nenhum usuário logado' });
  }
});

app.post('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.status(500).json({ erro: 'Erro ao encerrar a sessão' });
    }
    res.json({ mensagem: 'Sessão encerrada com sucesso' });
  });
});

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

**13. Configure o CORS para permitir apenas requisições de um domínio específico.**

Foi instalado o pacote cors por meio do gerenciador de pacotes npm para habilitar o controle de acesso entre origens distintas no servidor Express. Em seguida, foi importado o módulo e configurado o middleware cors para aceitar requisições apenas do domínio https://meusite.com, definindo esta URL no parâmetro origin das opções do middleware. Por fim, o middleware foi aplicado globalmente à aplicação Express, permitindo que apenas requisições originadas desse domínio fossem aceitas pelo servidor.

**Arquivo:**

index.js

```
const express = require('express');

const cors = require('cors');


const app = express();

const PORT = 3000;


const corsOptions = {

  origin: 'https://meusite.com', // Domínio de exemplo

  optionsSuccessStatus: 200

};


app.use(cors(corsOptions));


app.get('/dados', (req, res) => {

  res.json({ mensagem: 'Requisição autorizada via CORS' });

});


app.listen(PORT, () => {

  console.log(`Servidor rodando em http://localhost:${PORT}`);

});
```

## 14. Crie uma rota protegida que utilize tokens CSRF para validação.

Inicialmente, configurou-se o middleware para gerar tokens CSRF e armazená-los em cookies de forma automática. Em seguida, criou-se uma rota GET responsável por fornecer ao cliente um token válido, necessário para futuras requisições seguras. Por fim, foi desenvolvida uma rota POST protegida, que exige a presença e a validade do token CSRF para aceitar a requisição, garantindo que apenas origens autorizadas e legítimas possam submeter dados ao servidor.

Para testar a aplicação, o primeiro passo consiste em realizar uma requisição GET à rota /form, armazenando o cookie de sessão e extraindo o token CSRF fornecido. Em um segundo momento, o cliente deve enviar uma requisição POST à rota /dados, incluindo o token no cabeçalho csrf-token e reenviando os cookies da sessão obtidos anteriormente. Esse procedimento simula o comportamento de um formulário protegido por CSRF e assegura que apenas requisições deliberadas, iniciadas pelo próprio usuário e dentro do contexto da aplicação, sejam aceitas pelo servidor.

**Arquivo:**

index.js

```
const express = require('express');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const csrf = require('csurf');

const app = express();
const PORT = 3000;

app.use(cookieParser());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

const csrfProtection = csrf({ cookie: true });
```

```
app.get('/form', csrfProtection, (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});


app.post('/dados', csrfProtection, (req, res) => {
  res.json({ mensagem: 'Requisição POST protegida por token CSRF recebida com sucesso.' });
});

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

**15. Desenvolva uma função que previna SQL Injection utilizando prepared statements.**

A conexão com o banco de dados foi estabelecida de forma convencional, e a consulta foi estruturada com o uso de marcadores posicionais (?), nos quais os valores fornecidos pelo usuário são inseridos de forma segura. Essa abordagem impede que dados maliciosos sejam interpretados como parte da instrução SQL, uma vez que o driver do MySQL realiza o escaping automático dos parâmetros. A função criada recebe um nome como entrada e executa uma consulta parametrizada para buscar registros na tabela de usuários, retornando os resultados por meio de um callback. Dessa forma, assegura-se que a interação com o banco de dados ocorra de maneira robusta e resistente a injeções de código.

**Arquivo:**

index.js

```
const mysql = require('mysql2');


const connection = mysql.createConnection({
```

```
  host: 'localhost',
  user: 'seu_usuario',
  password: 'sua_senha',
  database: 'seu_banco'
});

function buscarUsuarioPorNome(nome, callback) {
  const sql = 'SELECT * FROM usuarios WHERE nome = ?';
  connection.execute(sql, [nome], (err, results) => {
    if (err) return callback(err);
    callback(null, results);
  });
}

buscarUsuarioPorNome('João', (err, resultados) => {
  if (err) {
    console.error('Erro na consulta:', err);
  } else {
    console.log('Resultados:', resultados);
  }
});
```

**16. Configure cookies seguros com os atributos HttpOnly e Secure.**

**Arquivo:**

Neste exercício, foi configurado um cookie seguro em uma aplicação desenvolvida com o framework Express, utilizando o middleware cookie-parser para facilitar o gerenciamento de cookies. A configuração adotada inclui os atributos HttpOnly, que impede o acesso ao cookie via JavaScript e, consequentemente, mitiga ataques do tipo Cross-Site Scripting (XSS), e Secure, que garante que o cookie seja transmitido apenas por conexões HTTPS, protegendo os dados contra interceptações em canais inseguros. Além disso, foi definido o atributo SameSite como Strict, restringindo o envio do cookie a requisições originadas no mesmo domínio, o que contribui para a prevenção de ataques de Cross-Site Request Forgery (CSRF). Essa configuração promove uma camada adicional de segurança no armazenamento e na transmissão de informações sensíveis entre cliente e servidor.

index.js

```
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();
const PORT = 3000;

app.use(cookieParser());

// Rota que define um cookie seguro
app.get('/set-cookie', (req, res) => {
  res.cookie('token', 'valor-seguro-do-token', {
    httpOnly: true,   // Impede acesso via JavaScript (protege contra XSS)
    secure: true,     // Garante envio apenas via HTTPS
```

```
    sameSite: 'Strict', // Impede envio em requisições cross-site (protege contra CSRF)
    maxAge: 60 * 60 * 1000 // 1 hora
  });

  res.send('Cookie seguro configurado com sucesso.');
});

app.get('/get-cookie', (req, res) => {
  const token = req.cookies.token;
  res.send(`Token recebido: ${token || 'Nenhum token encontrado'}`);
});

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

**17. Crie uma tabela de usuários com roles e insira dados fictícios.**

**SQL:**

```
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    senha VARCHAR(255) NOT NULL,
    role ENUM('admin', 'editor', 'leitor') NOT NULL DEFAULT 'leitor',
    criado_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
INSERT INTO usuarios (nome, email, senha, role) VALUES
('Alice Souza', 'alice@example.com', 'senha123', 'admin'),
('Bruno Lima', 'bruno@example.com', 'senha123', 'editor'),
('Carla Mendes', 'carla@example.com', 'senha123', 'leitor'),
('Daniel Rocha', 'daniel@example.com', 'senha123', 'editor'),
('Eduarda Reis', 'eduarda@example.com', 'senha123', 'leitor'),
('Fernando Alves', 'fernando@example.com', 'senha123', 'admin');
```

**18. Implemente middleware para restringir acesso com base em roles.**

**Código:**

```
/**
 * Middleware para restringir acesso com base em roles.
 * @param {...string} rolesPermitidas - Lista de roles que podem acessar a rota.
 */
function restringirPorRole(...rolesPermitidas) {
  return (req, res, next) => {
    const usuario = req.usuario;

    if (!usuario || !usuario.role) {
      return res.status(403).json({ mensagem: 'Acesso negado: usuário não autenticado ou role não definida.' });
    }

    if (!rolesPermitidas.includes(usuario.role)) {
      return res.status(403).json({ mensagem: 'Acesso negado: permissão insuficiente.' });
    }
```

```
    next();
  };
}

module.exports = restringirPorRole;
```

**19. Simule um ataque CSRF e implemente a solução para preveni-lo.**

Neste exercício, foi simulada uma situação comum de ataque chamada Cross-Site Request Forgery (CSRF), que acontece quando um usuário, mesmo estando autenticado, é levado a executar uma ação no site sem querer, geralmente por meio de um site malicioso. Para exemplificar essa vulnerabilidade, foi criada uma rota simples que realiza uma transferência financeira sem exigir nenhuma proteção extra, o que permite que alguém mal-intencionado envie comandos em nome do usuário sem que ele perceba.

Para evitar esse tipo de problema, foi implementada uma solução usando o middleware csurf no Express, que cria tokens únicos para cada sessão do usuário. Esses tokens precisam ser enviados junto com as requisições que modificam dados no servidor, funcionando como uma espécie de “senha temporária” que confirma que a ação veio de uma fonte confiável. Também foi criada uma rota para fornecer esse token ao cliente, para que ele possa incluí-lo em formulários ou cabeçalhos das requisições.

Assim, a rota que realiza a transferência só aceita comandos que venham acompanhados desse token válido, bloqueando qualquer tentativa de ação forjada que não tenha a autorização explícita do usuário. Dessa forma, a aplicação ganha uma camada extra de segurança, garantindo que somente ações legítimas e intencionais sejam processadas, protegendo os usuários contra fraudes causadas por sites maliciosos.

**Código:**

```
// Exemplo de rota vulnerável (sem proteção CSRF)
app.post('/transferir', (req, res) => {
  const { valor, destino } = req.body;
  // Aqui faria uma lógica de transferência de saldo, por exemplo
  res.send(`Transferência de R$${valor} para ${destino} realizada.`);
});
```

Solução com CSRF

```
const express = require('express');
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;

pp.use(cookieParser());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// Ativa o csurf para proteção com token via cookie
const csrfProtection = csrf({ cookie: true });
```

```
// Rota para fornecer o token ao cliente (ex: via front-end)
app.get('/formulario', csrfProtection, (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});

// Rota protegida que exige token CSRF válido
app.post('/transferir', csrfProtection, (req, res) => {
  const { valor, destino } = req.body;
  res.send(`Transferência segura de R$${valor} para ${destino} realizada.`);
});

app.listen(PORT, () => {
  console.log(`Servidor com proteção CSRF em http://localhost:${PORT}`);
});
```

**20. Crie um sistema de login simples que emita um JWT e proteja rotas baseadas no token.**

Primeiro, implementou-se uma rota de login que valida as credenciais fornecidas pelo usuário contra dados simulados e, em caso de sucesso, gera um token JWT contendo informações essenciais do usuário, como identificador, nome e papel (role), com prazo de validade configurado para uma hora. Para proteger recursos sensíveis, foi criado um middleware que intercepta requisições às rotas protegidas, extrai o token do cabeçalho de autorização, verifica sua validade e, caso seja válido, permite o acesso, armazenando as informações decodificadas no objeto de requisição para uso posterior. Essa abordagem assegura que apenas usuários autenticados e autorizados possam acessar determinadas áreas da aplicação, garantindo uma camada eficaz de segurança.

**Código:**

```
const express = require('express');

const jwt = require('jsonwebtoken');

const bodyParser = require('body-parser');


const app = express();

const PORT = 3000;

const SECRET_KEY = 'minha_chave_secreta_supersegura';


app.use(bodyParser.json());


// Usuário fictício para demonstração

const usuarioMock = {

  id: 1,

  username: 'usuario1',

  password: 'senha123', // Nunca armazene senha em texto plano na vida real!

  role: 'admin'

};

// Rota de login - gera e retorna token JWT

app.post('/login', (req, res) => {

  const { username, password } = req.body;


  if (username === usuarioMock.username && password ===
usuarioMock.password) {

    const payload = {
```

```
      id: usuarioMock.id,
      username: usuarioMock.username,
      role: usuarioMock.role
    };

    const token = jwt.sign(payload, SECRET_KEY, { expiresIn: '1h' });
    return res.json({ token });
  }

  res.status(401).json({ error: 'Usuário ou senha inválidos' });
});

// Middleware para proteger rotas e validar token
function verificarToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  if (!authHeader) return res.status(401).json({ error: 'Token não fornecido' });

  const token = authHeader.split(' ')[1]; // Espera formato "Bearer TOKEN"
  if (!token) return res.status(401).json({ error: 'Token inválido' });

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ error: 'Token inválido ou expirado' });
    req.user = user;
    next();
```

```
  });
}

// Rota protegida
app.get('/dashboard', verificarToken, (req, res) => {
  res.json({ mensagem: `Bem-vindo, ${req.user.username}!`, usuario: req.user });
});

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