UniSENAI

# Módulo 4: Desenvolvimento Web com Node.js

**Módulo 4: Desenvolvimento Web com Node.js**

---

## Objetivo:

Capacitar os alunos a desenvolver servidores web robustos com Node.js, utilizando o framework Express.js, integrando banco de dados MySQL e implementando boas práticas de segurança e manutenção.

---

## Conteúdo Detalhado:

### Aula 4.1: Introdução ao Express.js e criação de servidores básicos

#### O que é o Express.js?

- Um framework minimalista para Node.js que simplifica a criação de servidores e APIs.
- Oferece suporte para middleware, roteamento e integração com bancos de dados.

#### Instalação do Express.js

```
npm install express
```

#### Exemplo de Servidor Básico

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

app.get('/', (req, res) => {
  res.send('Bem-vindo ao servidor Express.js!');
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
});
```

---

## Aula 4.2: Definição de rotas: GET, POST, PUT e DELETE

### Tipos de Rotas

1. **GET:** Usado para buscar dados.
2. **POST:** Usado para enviar dados ao servidor.
3. **PUT:** Atualiza dados existentes.
4. **DELETE:** Remove dados.

### Exemplo Prático

```
app.get('/users', (req, res) => {
  res.json({ message: 'Listando usuários' });
});

app.post('/users', (req, res) => {
  res.json({ message: 'Usuário criado' });
});

app.put('/users/:id', (req, res) => {
  res.json({ message: `Usuário ${req.params.id} atualizado` });
});

app.delete('/users/:id', (req, res) => {
  res.json({ message: `Usuário ${req.params.id} excluído` });
});
```

---

## Aula 4.3: Middleware no Express.js (conceitos e práticas)

### O que é Middleware?

- Função executada antes de enviar a resposta ao cliente.
- Pode manipular requisições, respostas e próximas funções na cadeia de execução.

### Tipos de Middleware

1. **Global:** Aplica-se a todas as rotas.
2. **Específico:** Aplica-se a rotas selecionadas.

**Exemplo Prático**

```
// Middleware global
app.use((req, res, next) => {
  console.log(`Método: ${req.method}, URL: ${req.url}`);
  next();
});

// Middleware específico
app.use('/users', (req, res, next) => {
  console.log('Middleware aplicado apenas às rotas de /users');
  next();
});
```

---

### Aula 4.4: Integração com MySQL: Conexão e consultas básicas

#### Conexão com MySQL

- Use o pacote `mysql2`.
- Configuração:

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

const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'sua_senha',
  database: 'projeto_node'
});

connection.connect((err) => {
  if (err) throw err;
  console.log('Conectado ao MySQL!');
});
```

**Consulta Básica**

```
app.get('/users', (req, res) => {
  connection.query('SELECT * FROM usuarios', (err, results) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.json(results);
  });
});
```

---

## Aula 4.5: Gerenciamento de dados com CRUD completo no MySQL

**CRUD Completo**

1. **Create:** Inserção de dados no banco.
2. **Read:** Consulta de dados.
3. **Update:** Atualização de registros.
4. **Delete:** Exclusão de registros.

**Exemplo de CRUD**

```
// Create
app.post('/users', (req, res) => {
  const { nome, email } = req.body;
  connection.query('INSERT INTO usuarios (nome, email) VALUES (?, ?)', [nome, email], (err) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.status(201).send('Usuário criado com sucesso!');
  });
});

// Read
app.get('/users', (req, res) => {
  connection.query('SELECT * FROM usuarios', (err, results) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.json(results);
  });
});

// Update
app.put('/users/:id', (req, res) => {
  const { nome, email } = req.body;
  connection.query('UPDATE usuarios SET nome = ?, email = ? WHERE id = ?', [nome, email, req.params.id], (err) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.send('Usuário atualizado com sucesso!');
  });
});

// Delete
app.delete('/users/:id', (req, res) => {
  connection.query('DELETE FROM usuarios WHERE id = ?', [req.params.id], (err) => {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.send('Usuário excluído com sucesso!');
  });
});
```

---

## Aula 4.6: Boas práticas de manipulação de dados e proteção contra SQL Injection

**Boas Práticas**

1. **Use prepared statements:** Evita SQL Injection.
2. **Validação de entrada:** Certifique-se de que os dados recebidos sejam seguros.
3. **Use variáveis de ambiente:** Não exponha credenciais no código.

**Exemplo de Proteção**

```
app.get('/users', (req, res) => {
  const nome = req.query.nome;
  connection.query('SELECT * FROM usuarios WHERE nome = ?', [nome], (err, results)
=> {
    if (err) {
      res.status(500).send(err);
      return;
    }
    res.json(results);
  });
});
```

# O que é EventEmitter?

O `EventEmitter` é uma classe central no Node.js que fornece uma forma de lidar com eventos. Ele é usado para criar e gerenciar sistemas baseados em eventos, nos quais objetos podem emitir eventos e outras partes do código podem ouvir e reagir a esses eventos. Essa abordagem é fundamental em muitas aplicações Node.js, especialmente aquelas que requerem interação assíncrona.

A classe `EventEmitter` faz parte do módulo `events` do Node.js e segue o padrão de publicação/assinatura, onde um "emissor" emite um evento, e "ouvintes" (listeners) reagem a esse evento.

---

# Importando EventEmitter

Antes de usar o `EventEmitter`, você precisa importá-lo do módulo `events`:

```
const EventEmitter = require('events');
```

# Principais Métodos do EventEmitter

## 1. `on(event, listener)`

Adiciona um listener para um evento específico.

## 2. `emit(event, [arg1], [arg2], [...])`

Emite um evento, chamando todos os listeners registrados para ele.

### 3. `once(event, listener)`

Adiciona um listener que será executado apenas uma vez para o evento.

### 4. `removeListener(event, listener)` ou `off(event, listener)`

Remove um listener específico de um evento.

### 5. `removeAllListeners([event])`

Remove todos os listeners de um evento ou de todos os eventos.

### 6. `listenerCount(event)`

Retorna o número de listeners registrados para um evento específico.

---

## Exemplo Básico de Uso

Aqui está um exemplo básico para entender como o `EventEmitter` funciona:

```
const EventEmitter = require('events');

// Criando uma instância de EventEmitter
const myEmitter = new EventEmitter();

// Adicionando um listener para o evento 'greet'
myEmitter.on('greet', (name) => {
    console.log(`Olá, ${name}!`);
});

// Emitindo o evento 'greet'
myEmitter.emit('greet', 'João'); // Saída: Olá, João!
```

## Exemplo Prático: Sistema de Login

Imagine um sistema de login onde queremos emitir eventos baseados no sucesso ou falha de autenticação:

```
const EventEmitter = require('events');

// Criando a classe Auth, que herda de EventEmitter
class Auth extends EventEmitter {
    login(username, password) {
        // Lógica simulada de login
        if (username === 'admin' && password === '1234') {
            this.emit('loginSuccess', username);
        } else {
            this.emit('loginFailure', username);
        }
    }
}

const auth = new Auth();

// Registrando listeners
auth.on('loginSuccess', (user) => {
    console.log(`Usuário ${user} logado com sucesso!`);
});

auth.on('loginFailure', (user) => {
    console.log(`Falha no login para o usuário: ${user}`);
});

// Testando o sistema de login
auth.login('admin', '1234'); // Saída: Usuário admin logado com sucesso!
auth.login('user', 'wrongpass'); // Saída: Falha no login para o usuário: user
```

## Exemplo Avançado: Chat em Tempo Real

No caso de um sistema de chat, você pode emitir eventos como `message` ou `userJoined`:

```
const EventEmitter = require('events');

class ChatRoom extends EventEmitter {
    join(user) {
        this.emit('userJoined', user);
    }

    sendMessage(user, message) {
        this.emit('message', { user, message });
    }
}

const chatRoom = new ChatRoom();

// Listeners
chatRoom.on('userJoined', (user) => {
    console.log(`${user} entrou na sala de chat.`);
});

chatRoom.on('message', ({ user, message }) => {
    console.log(`${user} diz: ${message}`);
});

// Simulando eventos
chatRoom.join('Alice');
chatRoom.sendMessage('Alice', 'Olá a todos!');
chatRoom.join('Bob');
chatRoom.sendMessage('Bob', 'Oi, Alice!');
```

Saída:

```
Alice entrou na sala de chat.
Alice diz: Olá a todos!
Bob entrou na sala de chat.
Bob diz: Oi, Alice!
```

# Vantagens do EventEmitter

1. **Facilidade na implementação de eventos**: Com poucos métodos, é possível criar sistemas robustos.
2. **Flexibilidade**: Útil para cenários como sistemas de notificação, streams de dados ou interfaces interativas.
3. **Reutilização**: Eventos podem ser facilmente gerenciados e reutilizados em diferentes partes de um aplicativo.

---

# Boas Práticas

1. **Evitar vazamentos de memória**:
    - O Node.js exibe um aviso se mais de 10 listeners forem adicionados ao mesmo evento. Use `emitter.setMaxListeners(n)` para ajustar o limite.
2. **Remover listeners desnecessários**:
    - Listeners que não são mais úteis devem ser removidos para liberar recursos.
3. **Erro centralizado**:
    - Sempre adicione um listener para eventos de erro:

```
myEmitter.on('error', (err) => {
    console.error('Erro capturado:', err);
});
```

## Atividades de Fixação

### Atividades Teóricas

1. **Exploração da Documentação Oficial**
    - **Objetivo**: Familiarizar-se com a documentação do módulo `events` no Node.js.
    - **Atividade**: Pesquise a documentação oficial e responda:
        - Quais métodos principais a classe `EventEmitter` oferece?
        - Qual é a diferença entre os métodos `on` e `once`?
        - O que acontece se você tentar emitir um evento sem um listener?
2. **Diagrama de Fluxo de Eventos**
    - **Objetivo**: Entender o fluxo de um evento em um sistema baseado em eventos.
    - **Atividade**: Crie um diagrama de fluxo que mostre o processo de:
        - Registro de listeners.

        - Emissão de eventos.
        - Execução de callbacks.
3. **Questionário de Múltipla Escolha**
    - **Objetivo**: Testar o conhecimento básico sobre o `EventEmitter`.
    - **Atividade**: Responda perguntas como:
        - Qual método é usado para emitir eventos?
        - O que faz o método `removeListener`?
        - É possível registrar vários listeners para o mesmo evento?
4. **Análise de Código**
    - **Objetivo**: Identificar e explicar como o `EventEmitter` está sendo usado.
    - **Atividade**: Analise o seguinte código e explique cada parte:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('data', (info) => {
    console.log('Dado recebido:', info);
});
emitter.emit('data', { id: 1, message: 'Olá!' });
```

**5. Estudo de Caso**

- **Objetivo**: Relacionar o uso de eventos com casos reais.
- **Atividade**: Dê exemplos de aplicações práticas que poderiam usar o `EventEmitter`, como:
    - Streaming de áudio/vídeo.
    - Notificações em tempo real.
    - Controle de fluxos de dados em APIs.

---

## Atividades Práticas

6. **Criação de Eventos Simples**
    - **Objetivo**: Implementar e testar eventos básicos.
    - **Atividade**:
        - Crie um programa que emite um evento chamado `hello` e responde com "Hello, World!".
        - Modifique para receber o nome de uma pessoa e dizer "Hello, [nome]!".
7. **Sistema de Registro de Log**
    - **Objetivo**: Implementar um sistema de logging usando o `EventEmitter`.
    - **Atividade**:
        - Crie um evento `log` que registra mensagens no console.
        - Adicione diferentes níveis de log: `info`, `warn` e `error`.
8. **Simulação de Chat**

- **Objetivo**: Simular um sistema de chat com eventos.
- **Atividade**:
  - Crie uma classe `ChatRoom` que emite eventos quando usuários entram na sala e enviam mensagens.
  - Teste emitindo eventos como `userJoined` e `message`.

9. **Eventos com Múltiplos Listeners**
   - **Objetivo**: Demonstrar como vários listeners podem ser registrados para o mesmo evento.
   - **Atividade**:
     - Crie um evento `dataReceived` com dois listeners:
       1. Um listener que salva os dados em um arquivo.
       2. Outro que exibe os dados no console.
10. **Gerenciamento de Erros com Eventos**
    - **Objetivo**: Praticar o uso do evento `error`.
    - **Atividade**:
      - Implemente um `EventEmitter` que emite um evento `error` quando ocorre um problema.
      - Crie um listener para tratar esses erros e exibir mensagens apropriadas no console

---

## Extras

- **Desafio**: Combine as atividades práticas 6 e 7 para criar um programa que registra e gerencia logs baseados em eventos.
- **Feedback em Grupo**: Divida os alunos em grupos para discutir como poderiam melhorar os exemplos criados nas atividades práticas.

## Lista de Exercícios

### Questões Teóricas

1. O que é o Express.js e quais suas principais vantagens?
2. Quais são os quatro principais tipos de rotas HTTP?
3. Explique o conceito de middleware no Express.js.
4. Como criar uma conexão básica entre Node.js e MySQL?
5. O que é um prepared statement e como ele protege contra SQL Injection?
6. Qual é a estrutura básica de um servidor Express.js?
7. Explique a diferença entre rotas globais e rotas específicas.
8. Liste três boas práticas ao manipular dados em bancos de dados.
9. Por que é importante usar variáveis de ambiente em aplicações Node.js?
10. Como tratar erros em consultas SQL dentro do Node.js?

### Questões Práticas

1. Crie um servidor Express.js que responda à rota `/` com "Hello, World!".
2. Implemente uma rota GET que liste todos os usuários de um banco MySQL.
3. Adicione um middleware que registre o método HTTP e a URL de cada requisição.
4. Crie uma rota POST que insira um novo usuário no banco de dados.
5. Implemente uma rota PUT para atualizar o nome de um usuário pelo ID.
6. Crie uma rota DELETE que exclua um usuário pelo ID.
7. Use prepared statements para consultar um usuário pelo nome de forma segura.
8. Configure variáveis de ambiente para conexão com o banco.
9. Adicione tratamento de erros para uma consulta SQL que pode falhar.
10. Estruture um CRUD completo com separação de módulos em arquivos diferentes.

**Instruções para a Entrega das Atividades**

1. **Elaboração e Envio do Arquivo**
   - Responda todas as questões de forma clara e objetiva.
   - Gere um arquivo no formato **.PDF** contendo as respostas de cada questão.
   - Envie o arquivo para os e-mails dos professores responsáveis.
2. **Validação da Atividade**
   - Após o envio do arquivo, procure o(s) professor(es) para realizar a validação da atividade.
   - **Não inicie a próxima atividade sem antes validar a anterior com o professor.**
3. **Forma de Validação**
   - **Explicação Verbal**: Explique cada resposta verbalmente ao(s) professor(es).
   - **Perguntas e Respostas**: Esteja preparado para responder aos questionamentos do(s) professor(es) sobre o conteúdo das respostas.
   - **Orientação**: Receba orientações sobre a apresentação do(s) tema(s).