# Node.js

Profª Simone Dominico

# Servidores

- Exemplo com http e node.js
- Nesse exemplo, estamos utilizando o módulo **`http`** nativo do Node.js para criar um servidor HTTP simples.
- Quando o servidor receber uma requisição, ele definirá o código de status HTTP como 200, o cabeçalho de resposta como "text/plain" e o corpo da resposta como "Hello World".
- Em seguida, o servidor enviará a resposta de volta para o cliente.

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(3000, () => {
  console.log('Servidor rodando em http://localhost:3000/');
});
```

# Servidores

- utilizando o framework Express.js para criar um servidor web.
- O método **`app.get()`** é utilizado para definir uma rota que responde a requisições GET para o caminho raiz ("/").
- Quando o servidor receber uma requisição para essa rota, ele enviará de volta uma resposta com a mensagem "Hello World!".

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

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Servidor rodando em http://localhost:3000/');
});
```

# Protocolo http e node.js

- O protocolo HTTP (Hypertext Transfer Protocol) é um protocolo de comunicação que permite a transferência de dados pela internet.
- O Node.js é capaz de lidar com requisições e respostas HTTP de forma eficiente, utilizando o módulo **`http`** nativo do Node.js. Esse módulo fornece uma API para a criação de servidores HTTP e o processamento de requisições e respostas.
- Ao criar um servidor HTTP com o Node.js, é necessário definir uma função de callback que será executada sempre que o servidor receber uma requisição. Essa função de callback recebe dois parâmetros: uma requisição (objeto **`http.IncomingMessage`**) e uma resposta (objeto **`http.ServerResponse`**).
- A partir desses objetos, é possível acessar informações sobre a requisição (como o método HTTP utilizado, os cabeçalhos da requisição e o corpo da mensagem, se houver) e definir a resposta que será enviada de volta para o cliente (como o código de status HTTP, os cabeçalhos da resposta e o corpo da mensagem).

# Express - rotas, middlewares e templates

- O sistema de rotas e middlewares é um dos principais recursos do Express.js.
- Ele permite que os desenvolvedores possam definir rotas para as requisições HTTP e executar funções intermediárias (middlewares) antes ou depois do tratamento das rotas.
- As rotas são definidas no objeto **app** do Express.js.
- Para criar uma rota, utiliza-se um método HTTP do objeto **app** (como **`get()`**, **`post()`**, **`put()`**, **`delete()`**, entre outros) e especifica-se o caminho para a rota e uma função de callback que será executada sempre que a rota for acessada.
- Por exemplo, para definir u

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

# Exemplo

Exemplo, para criar um middleware que loga as requisições recebidas pelo servidor

A função de callback recebe três parâmetros: a requisição (**`req`**), a resposta (**`res`**) e a próxima função de middleware (**`next`**).

Nesse caso, estamos apenas imprimindo o método HTTP e a URL da requisição no console e em seguida chamando a próxima função de middleware.

Os middlewares também podem ser utilizados para definir rotas para grupos de requisições ou para lidar com erros e exceções.

# Utilizando middlewares para processar as requisições

O middleware é uma função que é executada antes ou depois do tratamento das rotas em uma aplicação Express.js.

Isso significa que os middlewares podem ser utilizados para realizar tarefas genéricas, como validação de dados, autenticação, log, entre outras.

Para utilizar middlewares em uma aplicação Express.js, basta utilizar o método **use()** do objeto **app**. Esse método permite que você defina um middleware que será executado sempre que uma requisição for recebida pelo servidor.

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

# Exemplo

- Nesse código, estamos utilizando a função de callback do **`use()`** para definir um middleware que loga todas as requisições recebidas pelo servidor.
- A função de callback recebe três parâmetros: a requisição (**`req`**), a resposta (**`res`**) e a próxima função de middleware (**`next`**).
- Nesse caso, estamos apenas imprimindo o método HTTP e a URL da requisição no console e em seguida chamando a próxima função de middleware.

# Implementando autenticação e autorização em uma aplicação Express.js

- O Express.js também oferece recursos para implementar autenticação e autorização em uma aplicação.
- Esses recursos são geralmente implementados utilizando middlewares.
- Para implementar a autenticação em uma aplicação, podemos utilizar o middleware **`passport`**.
- **`passport`** é uma biblioteca que oferece uma maneira fácil de implementar autenticação em uma aplicação Express.js.

# Exemplo

- Por exemplo, podemos utilizar o **`passport`** para implementar autenticação baseada em tokens JWT (JSON Web Token). Para isso, precisamos instalar as seguintes dependências:

```
npm install passport passport-jwt jsonwebtoken
```

```
// Configuração do middleware de autenticação
const secretKey = 'mysecretkey';
passport.use(new JWTStrategy({
    jwtFromRequest: ExtractJWT.fromAuthHeaderAsBearerToken(),
    secretOrKey: secretKey
  },
  (payload, done) => {
    // Aqui você pode realizar a verificação do token e retornar um usuário ou
    // Se o usuário for válido, você pode chamar o callback done com o usuário
    // Se o token for inválido, você pode chamar o callback done com um erro co

    const user = { id: payload.id, username: payload.username };

    // Aqui você pode realizar a verificação do token
    if (user) {
      done(null, user);
    } else {
      done(new Error('Token inválido'), null);
    }
  }
));
```

```
// Rota de autenticação que retorna um token JWT
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Aqui você pode realizar a verificação do usuário e senha
  // Se o usuário e senha forem válidos, você pode gerar um token JWT com
  // e retorná-lo como resposta da requisição
  const token = jwt.sign({ username }, secretKey);
  res.json({ token });
});

// Inicia o servidor
const port = 3000;
app.listen(port, () => {
  console.log(`Servidor rodando em http://localhost:${port}/`);
});
```

- Nesse código, estamos criando duas rotas: **`/protected`**, que requer autenticação, e **`/login`**, que gera um token JWT.
- A rota **`/protected`** é protegida pelo middleware de autenticação do **`passport`**. Isso significa que para acessar essa rota, o usuário precisa estar autenticado e enviar um token JWT válido no cabeçalho **`Authorization`** da requisição.
- A rota **`/login`** é responsável por gerar um token JWT válido para o usuário. Nesse exemplo, estamos utilizando a função **`jwt.sign()`** para gerar o token com base no nome de usuário. Depois de gerar o token, ele é retornado como resposta da requisição.

# Rotas

Para criar rotas em Express.js, basta utilizar os métodos HTTP do objeto **app**.

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


app.get('/', (req, res) => {
  res.send('Página inicial');
});
```

# Trabalhando com templates e views em Express.js

- Os templates são arquivos HTML que podem conter marcações específicas para a inserção de dados dinâmicos. Para utilizar templates em uma aplicação Express.js, podemos utilizar um motor de templates, como o **`ejs`** ou o **`pug`** (antigo **`jade`**).
- Por exemplo, para utilizar o **`ejs`** em uma aplicação Express.js, precisamos instalar a seguinte dependência:

```
npm install ejs
```

# Trabalhando com templates e views em Express.js

Em seguida, podemos configurar o **ejs** na aplicação .

Nesse código, estamos utilizando o método **`set()`** do objeto **`app`** para configurar o motor de templates **`ejs`**.

O primeiro parâmetro indica que estamos configurando o motor de templates e o segundo parâmetro indica qual é o nome do motor de templates que estamos utilizando.

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

// Configuração do motor de templates EJS
app.set('view engine', 'ejs');
```

# Trabalhando com templates

- Crie uma pasta chamada **`views`** na raiz do seu projeto. Nessa pasta, crie um arquivo chamado **`index.ejs`** com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Minha aplicação</title>
  </head>
  <body>
    <h1>Bem-vindo, <%= nome %>!</h1>
  </body>
</html>
```

# Trabalhando com templates

- Por fim, podemos utilizar o método **`render()`** do objeto **`res`** para renderizar o template com os dados dinâmicos. Por exemplo:

```
app.get('/', (req, res) => {
  res.render('index', { nome: 'Fulano' });
});
```

# Usando

- Crie um arquivo **`app.js`** com o seguinte conteúdo

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

// Configuração do diretório de views
app.set('views', './views');
app.set('view engine', 'ejs');

// Rota para a página inicial
app.get('/', (req, res) => {
  const nome = 'Fulano';
  res.render('index', { nome });
});

// Inicia o servidor
app.listen(3000, () => {
  console.log('Servidor iniciado na porta 3000');
});
```

# Implementando autenticação e autorização em uma aplicação Express.js

```
npm install express passport passport-local bcryptjs express-session
```

# Implementando autenticação e autorização em uma aplicação Express.js

- Crie uma pasta chamada **`controllers`** na raiz do seu projeto. Nessa pasta, crie um arquivo chamado **`authController.js`** com o seguinte conteúdo.
- Nesse código, estamos utilizando o **`bcrypt`** para criptografar a senha do usuário antes de armazená-la no banco de dados. Estamos também utilizando o **`passport`** para lidar com a autenticação do usuário, a partir da estratégia de autenticação local (**`passport-local`**).
- Estamos exportando os métodos **`getSignup`**, **`postSignup`**, **`getLogin`**, **`postLogin`** e **`getLogout`** que serão usados para renderizar e manipular as páginas de autenticação e autorização.

```
const bcrypt = require('bcryptjs');
const passport = require('passport');

const User = require('../models/user');

exports.getSignup = (req, res) => {
  res.render('auth/signup');
};

exports.postSignup = async (req, res) => {
  const { name, email, password } = req.body;

  const hash = await bcrypt.hash(password, 10);

  const user = new User({
    name,
    email,
    password: hash,
  });

  await user.save();

  res.redirect('/login');
};

exports.getLogin = (req, res) => {
  res.render('auth/login');
};

exports.postLogin = passport.authenticate('local', {
  successRedirect: '/',
  failureRedirect: '/login',
});

exports.getLogout = (req, res) => {
  req.logout();
  res.redirect('/');
};
```

- Crie uma pasta chamada **`models`** na raiz do seu projeto. Nessa pasta, crie um arquivo chamado **`user.js`** com o seguinte conteúdo:

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

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
});

module.exports = mongoose.model('User', userSchema);
```

- Crie uma pasta chamada **`config`** na raiz do seu projeto. Nessa pasta, crie um arquivo chamado **`passport.js`** com o seguinte conteúdo:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const bcrypt = require('bcryptjs');

const User = require('../models/user');

passport.use(
  new LocalStrategy({ usernameField: 'email' }, async (email, password, done) => {
    try {
      const user = await User.findOne({ email });

      if (!user) {
        return done(null, false, { message: 'Usuário não encontrado' });
      }

      const match = await bcrypt.compare(password, user.password);

      if (!match) {
        return done(null, false, { message: 'Senha incorreta' });
      }

      return done(null, user);
    } catch (error) {
      return done(error);
    }
  })
);

passport.serializeUser((user, done) => {
  done(null,
```

# Rotas para acesso

```
const authController = require('./controllers/authController');

// Rotas de autenticação e autorização
app.get('/signup', authController.getSignup);
app.post('/signup', authController.postSignup);
app.get('/login', authController.getLogin);
app.post('/login', authController.postLogin);
app.get('/logout', authController.getLogout);

// Rota de conteúdo restrito
app.get('/dashboard', isAuthenticated, (req, res) => {
  res.render('dashboard');
});
```

- Nesse código, estamos utilizando os métodos **`getSignup`**, **`postSignup`**, **`getLogin`**, **`postLogin`** e **`getLogout`** definidos no **`authController`** para manipular as páginas de autenticação e autorização.
- Estamos também definindo uma rota para a página de conteúdo restrito (**`/dashboard`**) que só pode ser acessada por usuários autenticados. Para isso, estamos utilizando o middleware **`isAuthenticated`**, que verifica se o usuário está autenticado antes de permitir o acesso à página.

# middleware **isAuthenticated**

- Esse middleware verifica se o usuário está autenticado utilizando o método **`isAuthenticated()`** do **`passport`**. Se o usuário estiver autenticado, o middleware permite o acesso à página. Se o usuário não estiver autenticado, o middleware redireciona o usuário para a página de login.
- Para testar a autenticação e autorização em sua aplicação, basta executá-la com o comando **`node app.js`** e acessar as rotas criadas em seu navegador. Lembre-se de criar um usuário de teste utilizando a página de registro (**`/signup`**) antes de tentar fazer login.

```
const isAuthenticated = (req, res, next) => {
  if (req.isAuthenticated()) {
    return next();
  }

  res.redirect('/login');
};
```

# Layout

- Crie uma pasta chamada view
- Um arquivo layout ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Minha aplicação</title>
  </head>
  <body>
    <header>
      <nav>
        <ul>
          <% if (user) { %>
            <li><a href="/dashboard">Dashboard</a></li>
            <li><a href="/logout">Sair</a></li>
          <% } else { %>
            <li><a href="/signup">Registrar</a></li>
            <li><a href="/login">Entrar</a></li>
          <% } %>
        </ul>
      </nav>
    </header>
    <main>
      <%- body %>
    </main>
  </body>
</html>
```

# Layout

- Crie um arquivo chamado **`signup.ejs`** na pasta **`views`** com o seguinte conteúdo:

```
<h1>Registrar</h1>
<form action="/signup" method="post">
  <div>
    <label for="name">Nome:</label>
    <input type="text" name="name" id="name" required>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" name="email" id="email" required>
  </div>
  <div>
    <label for="password">Senha:</label>
    <input type="password" name="password" id="password" required>
  </div>
  <div>
    <button type="submit">Registrar</button>
  </div>
</form>
```

# Layout

- Crie umCrie um arquivo chamado **`login.ejs`** na pasta **`views`** com o seguinte conteúdo:
- arquivo chamado **`signup.ejs`** na pasta **`views`** com o seguinte conteúdo:

```
<h1>Entrar</h1>
<form action="/login" method="post">
  <div>
    <label for="email">Email:</label>
    <input type="email" name="email" id="email" required>
  </div>
  <div>
    <label for="password">Senha:</label>
    <input type="password" name="password" id="password" required>
  </div>
  <div>
    <button type="submit">Entrar</button>
  </div>
</form>
```

# Layout

- Crie um arquivo chamado **`dashboard.ejs`** na pasta **`views`** com o seguinte conteúdo:

```
<h1>Dashboard</h1>
<p>Bem-vindo, <%= user.name %>!</p>
```

renderizar as páginas utilizando o motor de templates EJS no **`authController`**:

```
const bcrypt = require('bcryptjs');
const passport = require('passport');

const User = require('../models/user');

exports.getSignup = (req, res) => {
  res.render('auth/signup', { layout: 'layout' });
};

exports.postSignup = async (req, res) => {
  const { name, email, password } = req.body;

  const hash = await bcrypt.hash(password, 10);

  const user = new User({
    name,
    email,
    password: hash,
  });

  await user.save();

  res.redirect('/login');
};

exports.getLogin = (req, res) => {
  res.render('auth/login', { layout: 'layout' });
};

exports.postLogin = passport.authenticate('local', {
  successRedirect: '/dashboard',
  failureRedirect: '/login',
});

exports.getLogout = (req, res) => {
  req.logout();
  res.redirect('/');
};

exports.getDashboard = (req, res) => {
  res.render('auth/dashboard', { layout: 'layout', user: req.user });
};
```