Aula 02 - Aplicações Server-side vs Client side; Introdução ao express.js

1. Aplicações Server-side vs Client-side

1.1 Definições

Server-side (Lado do Servidor):

- O processamento ocorre principalmente no servidor.
- O servidor gera o HTML completo que é enviado ao cliente.
- Exemplo: Aplicações construídas com PHP, Ruby on Rails, ou Node.js com renderização no servidor.

Client-side (Lado do Cliente):

- O processamento ocorre no navegador do usuário.
- O servidor fornece dados (geralmente em formato JSON) e o cliente renderiza a interface.
- Exemplo: Aplicações SPA (Single Page Application) usando frameworks como React, Angular ou Vue.js.

1.2 Vantagens e Desvantagens

Server-side:

- **Vantagens:**
    - **SEO Melhorado:** Conteúdo renderizado no servidor é facilmente indexado por motores de busca.
    - **Compatibilidade:** Funciona mesmo em navegadores com JavaScript desabilitado.
    - **Segurança:** Lógica de negócios protegida no servidor.
- **Desvantagens:**
    - **Carga no Servidor:** Mais processamento pode levar a tempos de resposta mais lentos.
    - **Menos Interatividade:** Atualizações da página requerem recargas completas.

Client-side:

- **Vantagens:**
    - **Experiência de Usuário Rica:** Interfaces dinâmicas e responsivas.
    - **Menor Carga no Servidor:** O servidor fornece apenas dados, não interfaces completas.
    - **Atualizações em Tempo Real:** Sem necessidade de recarregar a página.
- **Desvantagens:**
    - **SEO Desafiador:** Pode ser difícil para motores de busca indexar conteúdo dinâmico.
    - **Dependência de JavaScript:** Usuários com JS desabilitado terão problemas.
    - **Complexidade Adicional:** Mais responsabilidade no lado do cliente.

2. Monólitos vs Microsserviços

2.1 Definições

Monólitos:

- Aplicação única e unificada.
- Todos os componentes (UI, lógica de negócios, acesso a dados) estão integrados.
- Exemplo: Aplicações tradicionais onde todo o código reside em um único projeto.

Microsserviços:

- Arquitetura composta por serviços pequenos e independentes.
- Cada serviço executa um processo e se comunica através de APIs.
- Exemplo: Serviços separados para autenticação, processamento de pagamentos, gerenciamento de produtos, etc.

2.2 Vantagens e Desvantagens

Monólitos:

- **Vantagens:**
  - **Simplicidade:** Mais fácil de desenvolver e implementar inicialmente.
  - **Depuração Simplificada:** Ambiente único facilita o rastreamento de erros.
  - **Desenvolvimento Rápido:** Menos sobrecarga de arquitetura.
- **Desvantagens:**
  - **Escalabilidade Limitada:** Dificuldade em escalar componentes individuais.
  - **Atualizações Arriscadas:** Alterações podem impactar todo o sistema.
  - **Barreira Tecnológica:** Difícil adotar novas tecnologias em partes do sistema.

Microsserviços:

- **Vantagens:**
  - **Escalabilidade Flexível:** Serviços podem ser escalados independentemente.
  - **Manutenibilidade:** Código mais modular facilita atualizações.
  - **Diversidade Tecnológica:** Possibilidade de usar tecnologias diferentes para cada serviço.
- **Desvantagens:**
  - **Complexidade Operacional:** Gestão de múltiplos serviços é mais complexa.
  - **Comunicação Entre Serviços:** Pode introduzir latência e necessidade de gerenciamento de APIs.
  - **Depuração Complexa:** Rastreamento de erros através de múltiplos serviços pode ser difícil.

---

3. Desempenho

Considerações de Desempenho

- **Latência:** Aplicações client-side podem ter tempos de carregamento iniciais maiores devido ao download de scripts.
- **Processamento:** Aplicações server-side colocam mais carga no servidor; client-side distribui o processamento para os clientes.
- **Escalabilidade:** Microsserviços permitem escalabilidade horizontal mais granular.
- **Carga de Rede:** Comunicação entre microsserviços pode aumentar o tráfego interno.

---

4. Outros Tópicos Relacionados

Tendências e Tecnologias Emergentes

- **Arquitetura Serverless:** Executa código sem gerenciar servidores, escalando automaticamente.
- **Progressive Web Apps (PWA):** Aplicações web com funcionalidades de aplicativos nativos.
- **Edge Computing:** Processamento de dados próximo à fonte para reduzir latência.
- **GraphQL:** Linguagem de consulta para APIs que permite solicitar exatamente os dados necessários.

---

5. Introdução ao Express.js

5.1 O que é o Express.js?

- **Express.js** é um framework web para Node.js que facilita a criação de aplicações web e APIs robustas.
- Fornece uma camada minimalista de funcionalidades, permitindo extensibilidade através de middleware.

5.2 Configuração

Passo 1: Instalar o Node.js

- Baixe e instale o Node.js do site oficial: nodejs.org

Passo 2: Criar um Projeto

```
mkdir minha-aplicacao
cd minha-aplicacao
npm init -y
```

Passo 3: Instalar o Express.js

```
npm install express
```

5.3 Criando um Servidor Básico

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

```
const port = 3000;

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

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

5.4 Rotas

Definindo Rotas Simples

- **Métodos HTTP Comuns:** `GET`, `POST`, `PUT`, `DELETE`

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

Rotas com Parâmetros

```
app.get('/usuarios/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`Perfil do usuário ${userId}`);
});
```

Rotas com Query Parameters

```
app.get('/buscar', (req, res) => {
  const termo = req.query.termo;
  res.send(`Resultados da busca para: ${termo}`);
});
```

5.5 Wildcards e Expressões Regulares

Usando Wildcards

- Captura rotas que seguem um padrão.

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

Expressões Regulares em Rotas

```
app.get(/.*fly$/, (req, res) => {
  res.send('Rota que termina com "fly"');
});
```

5.6 Manipulação do Corpo da Requisição (Body)

Middleware para Parsing

- **JSON:**

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

- **URL-Encoded:**

```
app.use(express.urlencoded({ extended: true }));
```

Manipulando Dados de Formulários

```
app.post('/login', (req, res) => {
  const { usuario, senha } = req.body;
  // Lógica de autenticação aqui
  res.send(`Usuário ${usuario} autenticado!`);
});
```

5.7 Middleware

Definição de Middleware

- Funções que têm acesso ao objeto de requisição (`req`), resposta (`res`) e ao próximo middleware na cadeia (`next`).

Exemplo de Middleware Global

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

Middleware em Rotas Específicas

```
const verificaAutenticacao = (req, res, next) => {
  // Lógica de verificação
  next();
};

app.get('/area-segura', verificaAutenticacao, (req, res) => {
  res.send('Bem-vindo à área segura!');
});
```

5.8 Tratamento de Erros

Middleware de Tratamento de Erros

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Algo deu errado!');
});
```

5.9 Modularizando a Aplicação

Criando Rotas em Arquivos Separados

- **Arquivo** `routes/usuarios.js`**:**

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.send('Lista de usuários');
});

router.get('/:id', (req, res) => {
  res.send(`Detalhes do usuário ${req.params.id}`);
});

module.exports = router;
```

- **Integrando no App Principal:**

```
const usuariosRouter = require('./routes/usuarios');
app.use('/usuarios', usuariosRouter);
```

Referências

- Node.js Documentation

- Documentação Oficial do Express.js
- Artigo sobre Arquiteturas Monolíticas vs Microsserviços
- Métodos HTTP