Aula 3 - JS, Node.js e Express

Tópicos da aula

- Javascript
  - conhecendo a linguagem: conceitos básicos
  - ES6+
- Node.js
  - vantagens e uso
  - pq usar javascript no backend?
  - ambiente de desenvolvimento
- Express

Javascript

- Linguagem de programação interpretada, de alto nível e dinâmica
- Lado do cliente e do servidor (**node.js**)
- Dinamicamente tipada
- Baseada em protótipos
- Funcional e Orientada a objetos

Conceitos básicos

- variáveis: var, let e const;
- tipos de dados: number, string, boolean, null, undefined, **array, funções**…
- operadores:
  - aritméticos
  - comparação
  - lógicos
- estruturas de controle
  - if, else if, else e switch
  - for, while, do…while
- funções

```
function soma(a, b) {
  return a + b;
}
```

```
const saudacao = function (nome) {
  return `Olá, ${nome}!`;
};
```

```
const multiplicar = (a, b) => a * b;
```

O que o ES6+ trouxe de mais importante?

- Classes
  - sintaxe para criação de objetos e herança baseada em protótipos
- Modules
  - permitiu a exportação e importação de código entre arquivos;

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  message() {
    return `My name is ${this.name} and i have ${this.age} old.`;
  }
}

const p1 = new Person("Alice", 25);
console.log(p1.message());
```

```
export const name = "Alice";

export function message() {
  console.log("Hello world!");
}

import { name, message } from './modulo.js';
message();
```

Classes

- Forma de implementar OOP em Javascript;

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  message() {
    return `My name is ${this.name} and i have ${this.age} old.`;
  }
}

const p1 = new Person("Alice", 25);
console.log(p1.message());
```

- Introduziu os conceitos de herança

```
class Student extends Person {
  constructor(name, age, course) {
    super(name, age);
    this.course = course;
  }

  StudentInfo() {
    return `${this.message()}. I'm studying ${this.course}.`;
  }
}

const Student1 = new Student("Bob", 22, "Prog web");
console.log(Student1.StudentInfo());
```

- Promises:
  - um objeto que representa a eventual conclusão de uma operação assíncrona e seu valor resultante;
  - pendente, cumprida e rejeitada
  - then(onFulfilled, onRejected), catch(onRejected), finally(onFinally)

```
const myPromise = new Promise((resolve, reject) => {
  let sucesso = true;

  if (sucesso) {
    resolve("Successfull!");
  } else {
    reject("Error");
  }
});
```

```
myPromise
  .then((message) => {
    console.log(message);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    console.log('Successfull!');
  });
```

```
const url = "https://api.exemplo.com/data";

function searchData(url) {
  return new Promise((resolve, reject) => {
    fetch(url)
      .then((response) => {
        if (!response.ok) {
          throw new Error("Network error");
        }
        return response.json();
      })
      .then((data) => resolve(data))
      .catch((error) => reject(error));
  });
}

searchData(url)
  .then((data) => {
    console.log("response data:", data);
  })
  .catch((error) => {
    console.error("Error to get data:", error);
  });
```

Promise.all
Promise.race
Promise.allSettled
Promise.any

Qual foi o verdadeiro ganho das Promises em Javascript?

- Código assíncrono era frequentemente gerenciado por meio de callbacks;
- Complexidade do código e a difícil manutenção

```
doSomething(function (result) {
  doSomethingElse(
    result,
    function (newResult) {
      doThirdThing(
        newResult,
        function (finalResult) {
          console.log(finalResult);
        },
        failureCallback
      );
    },
    failureCallback
  );
}, failureCallback);
```

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => console.log(finalResult))
  .catch((error) => console.error(error));
```

- Melhor tratamento de erros
- Encandeamento mais simples;
- Operações paralelas;
- Compatibilidade com **async/await**

Async/await

- Introduzida no ES8;
- Permitiu que código assíncrono fosse escrito de maneira síncrona;

```
async function fetchData() {
  try {
    const response1 = await fetch("url1");
    const data1 = await response1.json();

    const response2 = await fetch("url2");
    const data2 = await response2.json();

    console.log("Data:", data1, data2);
  } catch (error) {
    console.error("Erro to get data info:", error);
  }
}

fetchData();
```

- Grandes benefícios ao gerenciamento de operações assíncronas em JavaScript;
- Evitar o callback hell;

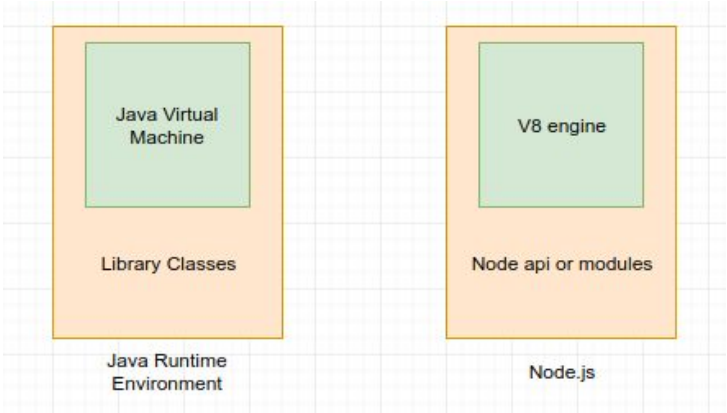
```
const fs = require("fs");

fs.readFile("file1.txt", "utf8", function (err, data1) {
  if (err) {
    console.error(err);
    return;
  }
  processFile(data1, function (err, processedData) {
    if (err) {
      console.error(err);
      return;
    }
    fs.writeFile("file2.txt", processedData, function (err) {
      if (err) {
        console.error(err);
        return;
      }
      console.log("File written successfully");
    });
  });
});
```

- Legibilidade
- Manutenção
- Tratamento de erros
- Reusabilidade

```
const fs = require('fs').promises;

async function processFiles() {
  try {
    const data1 = await fs.readFile('file1.txt', 'utf8');
    const processedData = await processFile(data1);
    await fs.writeFile('file2.txt', processedData);
    console.log('File written successfully');
  } catch (err) {
    console.error(err);
  }
}

async function processFile(data) {
  return new Promise((resolve, reject) => {
    const processedData = data.toUpperCase();
    resolve(processedData);
  });
}

await processFiles();
```

Node.js

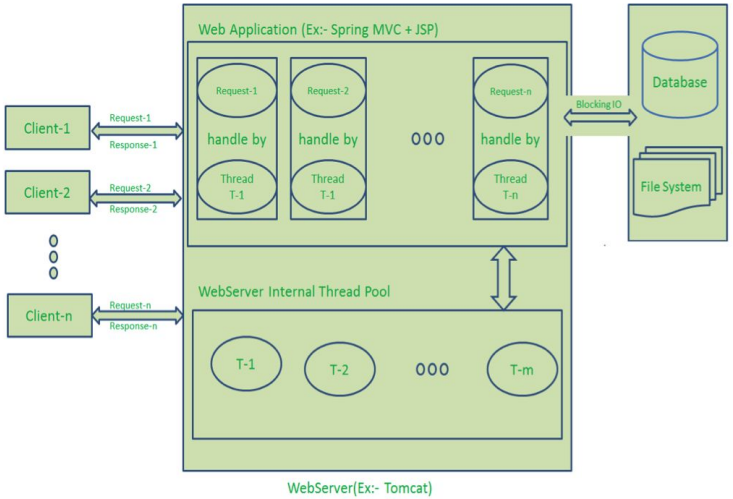
- Ambiente de execução JavaScript assíncrono baseado no V8 do Google Chrome;
- Sair do navegador e executar em máquina próprias, como uma aplicação autônoma;



- V8 engine transmit código baixo nível para o computador, que não tem a necessidade de interpretá-lo primeiramente;

Node.js

- Single-threaded, Event-driven - Modelo de Loop de Eventos com Thread Única
- Request-Response Multi-threaded;



1. cliente envia requisições para o servidor web (SW);
2. SW mantém um pool de threads limitados;
3. SW em loop infinito esperando;
4. SW recebe a requisição e a atribui a uma thread;

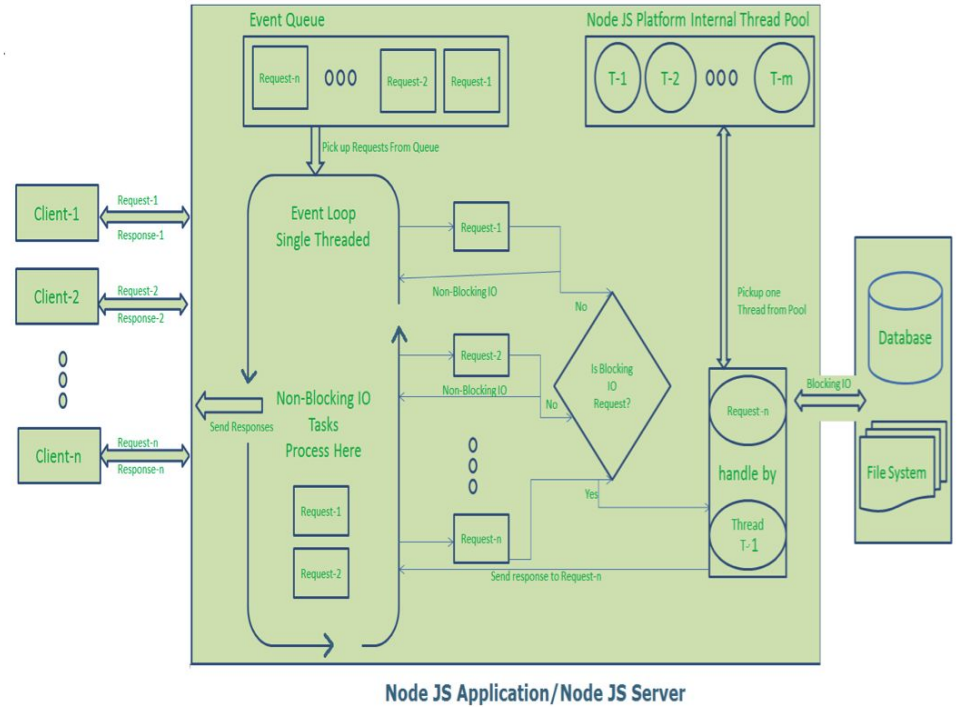
desvantagem: cria uma req por cliente, ocupa as threads.

Node.js

- Single-threaded, Event-driven - Modelo de Loop de Eventos com Thread Única

baseado no modelo de eventos do Javascript;

1. Clientes enviam req para o SW;
2. SW mantém um pool de threads;
3. Event Queue;
4. Event Loop (EL);
   a. utiliza apenas uma thread;
5. Se tiver req na Queue;
   a. processa e se não houver IO bloqueante atribui a uma thread;
   b. a thread processa e retorna ao EL;
6. EL devolve ao cliente;



Node.js - NPM (node package manager)

- Maior ecossistema de bibliotecas de código aberto do mundo;
- Facilita a instalação e gerenciamento de pacotes;

```
{
  "name": "my_project",
  "version": "1.0.0",
  "description": "A little description of my project",
  "main": "index.js",
  ▷ Depurar
  "scripts": {
    "test": "echo \"Error: no test specified.\" && exit 1"
  },
  "author": "Glauber",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Node.js - vantagens

- Alta performance devido ao V8 e ao modelo non-blocking I/0
- Escalabilidade
- Linguagem server-side e client-side;
- Javascript é uma das linguagens mais utilizadas atualmente;

Express

- Framework web para Node.js e fornece um conjunto robusto de recursos;
- Abstrai funcionalidades como:
  - roteamento e middleware

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

app.post("/submit", (req, res) => {
  res.send("Dados enviados");
});

app.put("/update/:id", (req, res) => {
  res.send(`Atualizar item com ID: ${req.params.id}`);
});

app.delete("/delete/:id", (req, res) => {
  res.send(`Deletar item com ID: ${req.params.id}`);
});
```

```
1  const express = require('express');
2  const app = express();
3
4  // Aplication Middleware
5  app.use((req, res, next) => {
6    console.log(`${req.method} ${req.url}`);
7    next();
8  });
9
10 // Route Middleware
11 app.get('/usuario/:id', (req, res, next) => {
12   const userId = req.params.id;
13   if (isNaN(userId)) {
14     res.status(400).send('UserId invalid');
15   } else {
16     next();
17   }
18 }, (req, res) => {
19   const userId = req.params.id;
20   res.send(`Usuário ID: ${userId}`);
21 });
22
23 // Manipulate error Middleware
24 app.use((err, req, res, next) => {
25   console.error(err.stack);
26   res.status(500).send('Something wrong!');
27 });
```

Express

- Trabalhando com dados:
  - O Express facilita a manipulação de dados em requisições HTTP usando middlewares como body-parser

```
1  const bodyParser = require("body-parser");
2  app.use(bodyParser.json());
3
4  app.post("/users", (req, res) => {
5    const user = req.body;
6    res.send(`User ${user.name} was created!`);
7  });
8
```

Lab 01 -

link:

https://github.com/joseglauberbo/ProgWeb

Referências

- FLANAGAN, David. **JavaScript: o guia definitivo**. Bookman Editora, 2012.
- https://www.digitalocean.com/community/tutorials/node-js-architecture-single-threaded-event-loop
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Node_server_without_framework
- https://docs.npmjs.com/