INTRODUÇÃO AO NODE.JS

Prof. Me. Hélio Esperidião

O que é Node.js?

Node.js não é uma linguagem de programação

Você programa utilizando a linguagem JavaScript.

Possui semelhanças com linguagens compiladas, uma vez que uma máquina virtual faz etapas de pré-compilação e otimização antes do código entrar em operação.

O que é Node.js?

Node.js não é um framework Javascript

É uma aplicação na qual você escreve seus programas com Javascript, estes serão compilados, otimizados e interpretados pela **máquina virtual V8**.

A VM é a mesma que o Google utiliza para executar Javascript no Chrome, e foi a partir dela que surgiu do Node.js

Para que serve Node.js?

Pode substituir java, c#, php, etc no back end de aplicações.

Node.js serve para fazer APIs.

Esse talvez seja o principal uso da tecnologia, uma vez que por default ela apenas sabe processar requisições.

Não apenas por essa limitação, mas também porque seu modelo não bloqueante de tratar as requisições o torna excelente para essa tarefa consumindo pouquíssimo hardware.

Para que serve Node.js?

Node.js serve para fazer backend de jogos, IoT e apps de mensagens.

Node.js para APIs nestas circunstâncias (backend-as-a-service) devido ao alto volume de requisições que esse tipo de aplicações efetuam.

Quais as desvantagens do Javascript no backend?

- Não implementa orientação objeto a objeto com todos os recursos de linguagens como java, c# ou php.

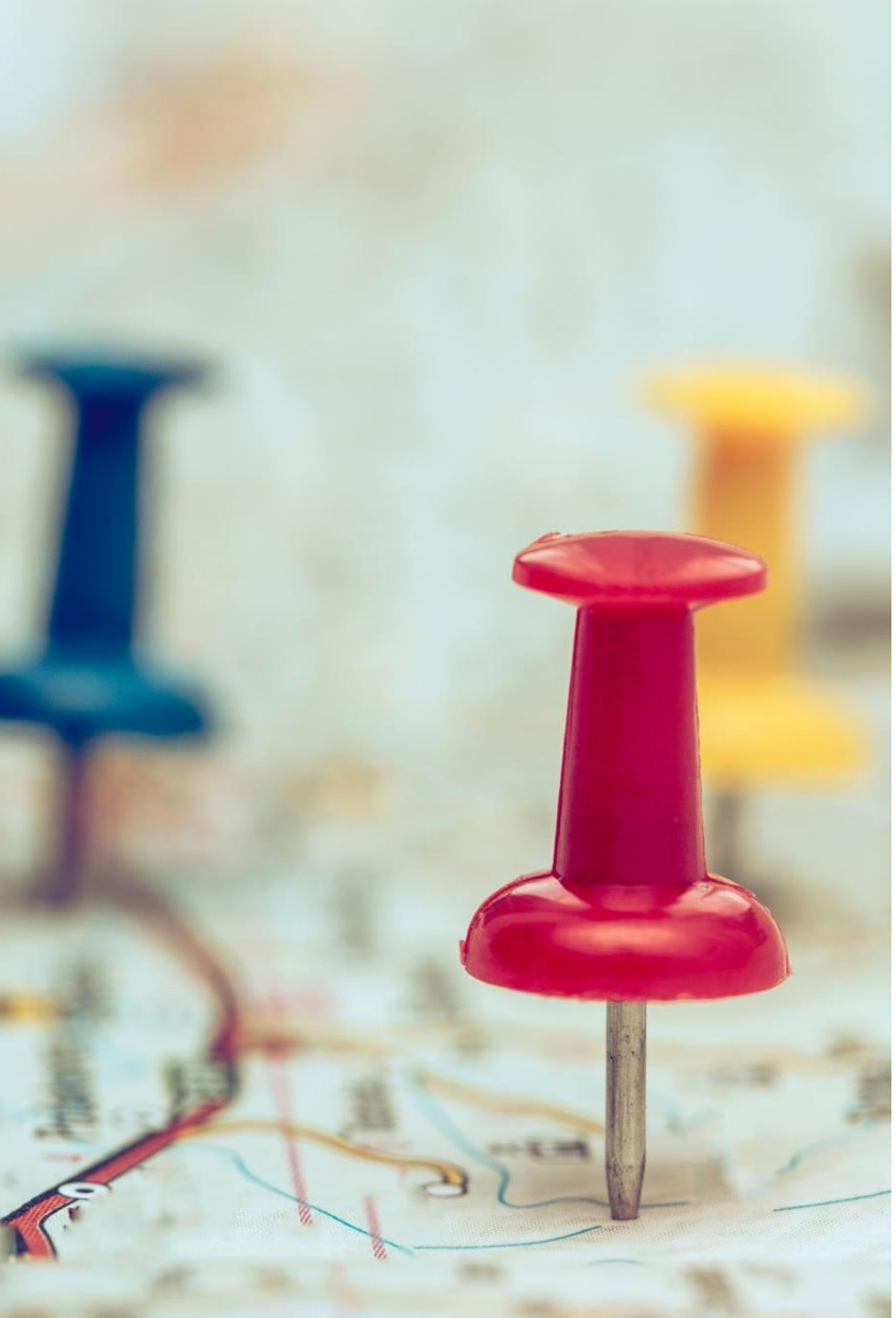
Suporte a banco de dados

Bancos de Dados: você pode utilizar tanto com bancos relacionais quanto com não-relacionais:

- MySQL
- PostgreSQL
- MS SQL Server
- MongoDB
- Redis
- SQLite

Quem usa Node.js?

- Netflix
- Linkedin
- Walmart
- Trello
- Uber
- PayPal
- eBay
- NASA



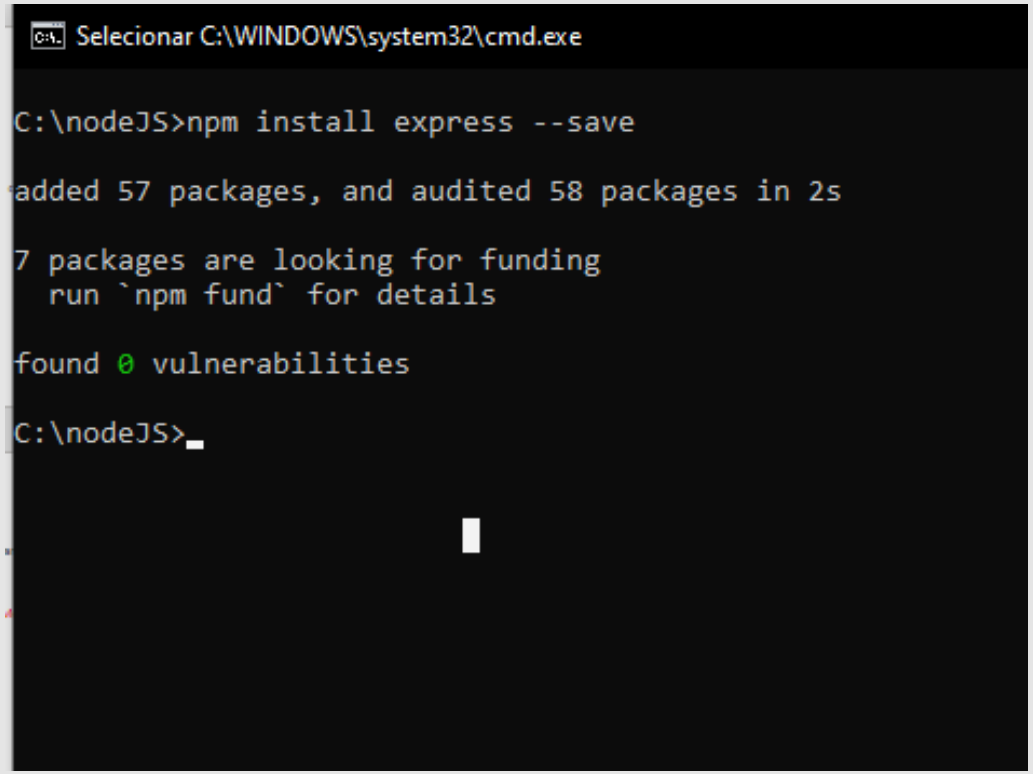
Download

- https://nodejs.org/en/download/

Express

- O Express é um framework para aplicativo da web do Node.js mínimo e flexível que fornece um conjunto robusto de recursos para aplicações web.

Instalando o express



- Acesse o diretório da sua aplicação
  - Digite: npm install express --save

app.js

```
var express = require('express'); //importa o express
var app = express();          //recupera uma instancia de express
app.use(express.static('view'))  // pasta view é estática não é preciso criar rotas para essa pasta
//rota: GET: /
app.get('/',  (request, response) => {
    response.send('Ola mundo');
});
//rota: GET: /rota2
app.get('/rota2',  (request, response) => {
    response.send('Ola mundo2');
});
//inicia a espera por requisições http
app.listen(3000);
```

Rodando a aplicação

- Digite no console:
  - node app.js
- Toda vez que modificar o código é necessário parar o programa atual e reiniciar a aplicação.
  - Para parar a aplicação digite no console:
    - control+c
    - digite novamente: node app.js

Pouco produtivo

- Parar a aplicação e rodar novamente toda vez que ocorrer uma modificação no código é extremamente improdutivo.

nodemon

- O nodemon é uma biblioteca que ajuda no desenvolvimento de sistemas com o Node.js reiniciando automaticamente o servido
- Instale:
  - npm install nodemon –g
- Inicie a aplicação utilizando no nodemoon:
  - nodemon app.js
- Sempre que houver uma modificação no código automaticamente o servidor será reiniciado.

Rotas

```
var express = require('express');
var app = express();
app.get(‘/rota1', function (req, res) {
      res.send(“r1”);
});
app.get(‘/rota2', function (req, res) {
      res.send(“r2”);
});
app.get(‘/rotan', function (req, res) {
      res.send(“rn”);
});

app.listen(8080);
```

Fácil visualizar e organizar rotas

req, res

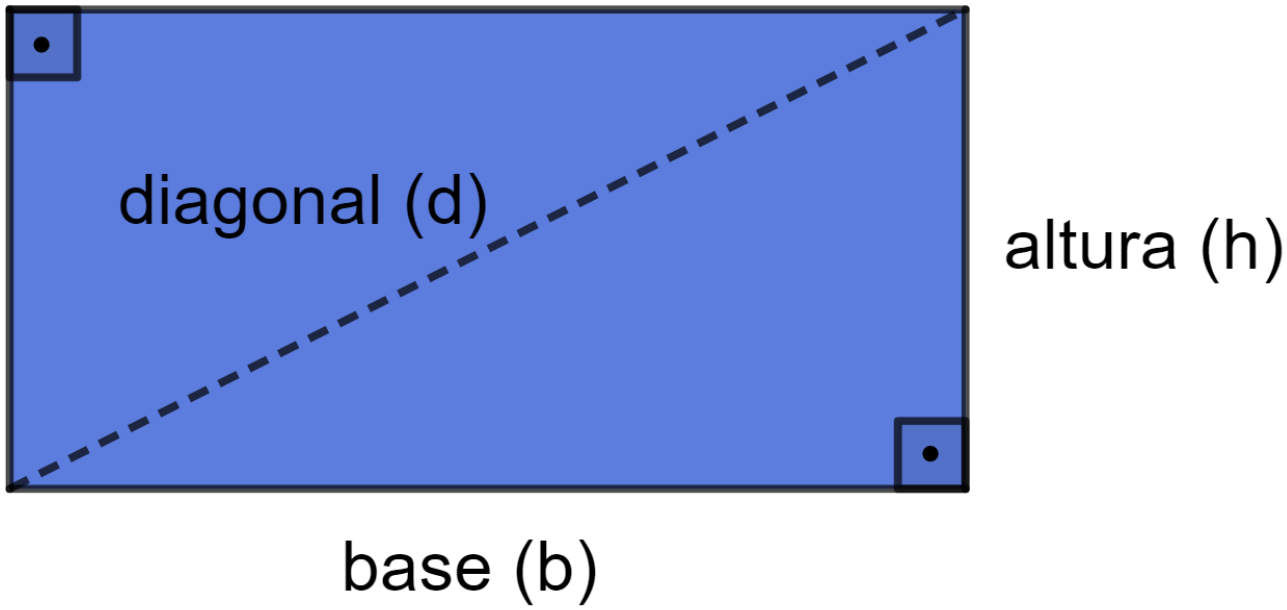
```
app.get('/', (request, response) => {
  res.send('Ola mundo');
});
```

- request é um objeto que possui dados da requisição
- response é um objeto que possui dados referente a respota.
- get: verbo http pode ser substituído por: post, put, delete, etc.

Classe (Retangulo)

- Atributos (Características)
  - base
  - altura

- Métodos
  - CalcularArea()
  - CalcularDiagonal()
  - CalcularPerimetro()

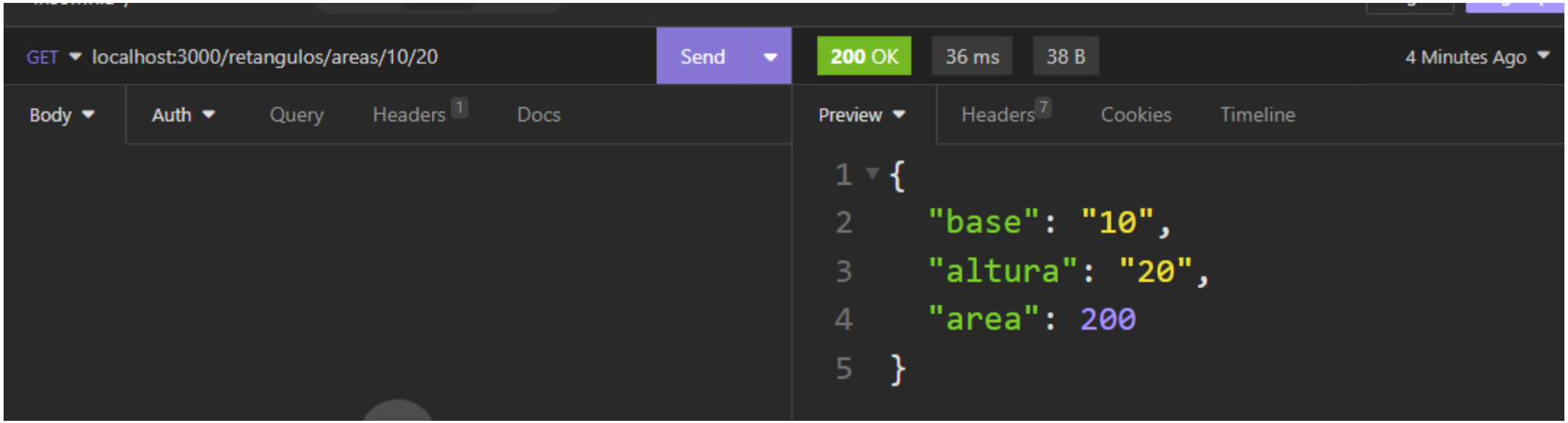


```
module.exports = class Retangulo { // arquivo Retangulo.js
    base = null;
    altura =null;
    constructor(base,altura) {
        this.base = base;
        this.altura = altura;
    }
    calcularArea(){
        const area = this.base * this.altura;
        return area;
    }
    calcularPerimetro(){
        const perimetro = this.base*2 +  this.altura*2;
        return perimetro;
    }
    calcularDiagonal(){
        const diagonal = Math.sqrt(this.base*this.base +
this.altura*this.altura);
        return diagonal;
    }
    setBase(base) {
        this.base = base;
    }
    getBase() {
        return this.base;
    }
    setAltura(altura) {
        this.altura = altura;
    }
    getAltura() {
        return this.altura;
    }
}
```

POO

```
//como utilizar a classe?
let Retangulo = require('./modelo/Retangulo');//importa arquivo
app.get('/retangulos/areas/:base/:altura',  (request, response) => {
    const base = request.params.base;
    const altura = request.params.altura;
    const ret1 = new Retangulo(base,altura);
    const calculo = ret1.calcularArea();
    const resposta ={
        base: ret1.getBase(),
        altura: ret1.getAltura(),
        area:calculo
    }
    response.send(resposta);
});
```

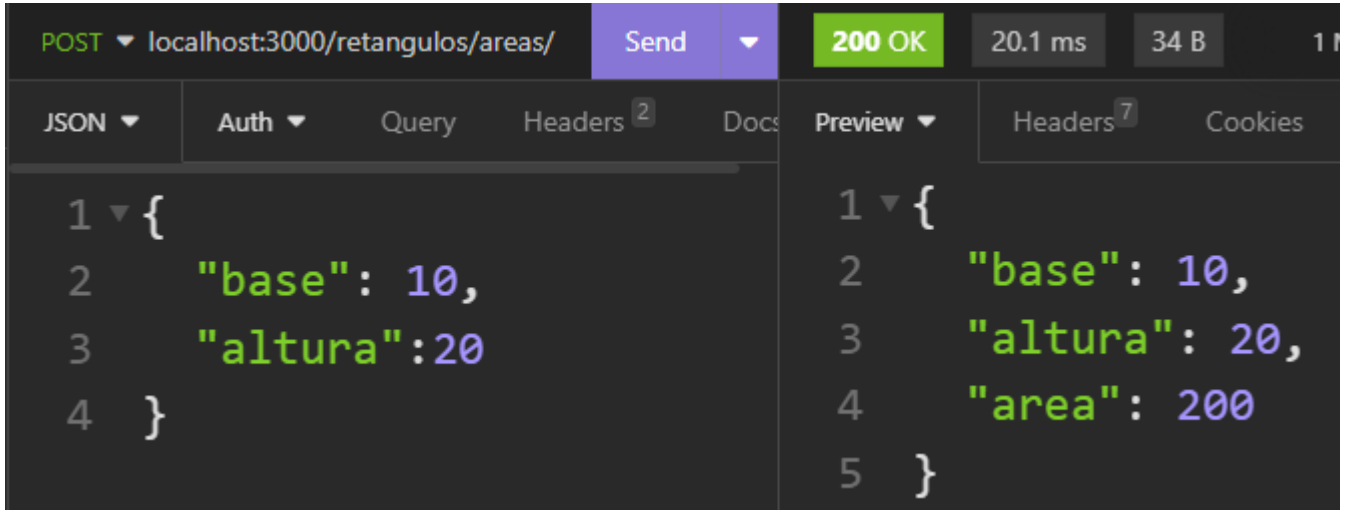
Teste


GET
localhost:3000/retangulos/areas/10/20
Send
200 OK
36 ms
38 B
4 Minutes Ago
Body
Auth
Query
Headers 1
Docs
Preview
Headers 7
Cookies
Timeline
1 {
2 "base": "10",
3 "altura": "20",
4 "area": 200
5 }

Recuperando dados no corpo(body) da requisição

```
let Retangulo = require('./modelo/Retangulo’);
//importe: app.use(express.json())
app.post('/retangulos/areas/',  (request, response) => {
    const base = request.body.base;
    const altura = request.body.altura;
    const ret1 = new Retangulo(base,altura);
    const calculo = ret1.calcularArea();

    const resposta ={
        base: ret1.getBase(),
        altura: ret1.getAltura(),
        area:calculo
    }
    response.send(resposta);
});
```



Promise

- Em JavaScript, uma "Promise" (promessa) é um objeto que representa um valor que pode estar disponível agora, no futuro ou nunca.
- Promises são usadas para lidar com operações assíncronas, como fazer uma solicitação de rede, ler um arquivo do disco ou executar uma consulta de banco de dados, onde o resultado não está imediatamente disponível.

Estados de uma promise.

- O objetivo principal das Promises é tornar o código assíncrono mais fácil de ler, manter e depurar, evitando o chamado "callback hell" (aninhamento excessivo de callbacks).
- Uma Promise tem três estados possíveis:
  - Pending (Pendente): O estado inicial, quando a promessa está em execução e o resultado ainda não está disponível.
  - Fulfilled (Cumprida): A promessa foi resolvida com sucesso e o valor desejado está disponível.
  - Rejected (Rejeitada): A promessa foi rejeitada devido a um erro e não foi cumprida com sucesso. O motivo do erro geralmente é fornecido.

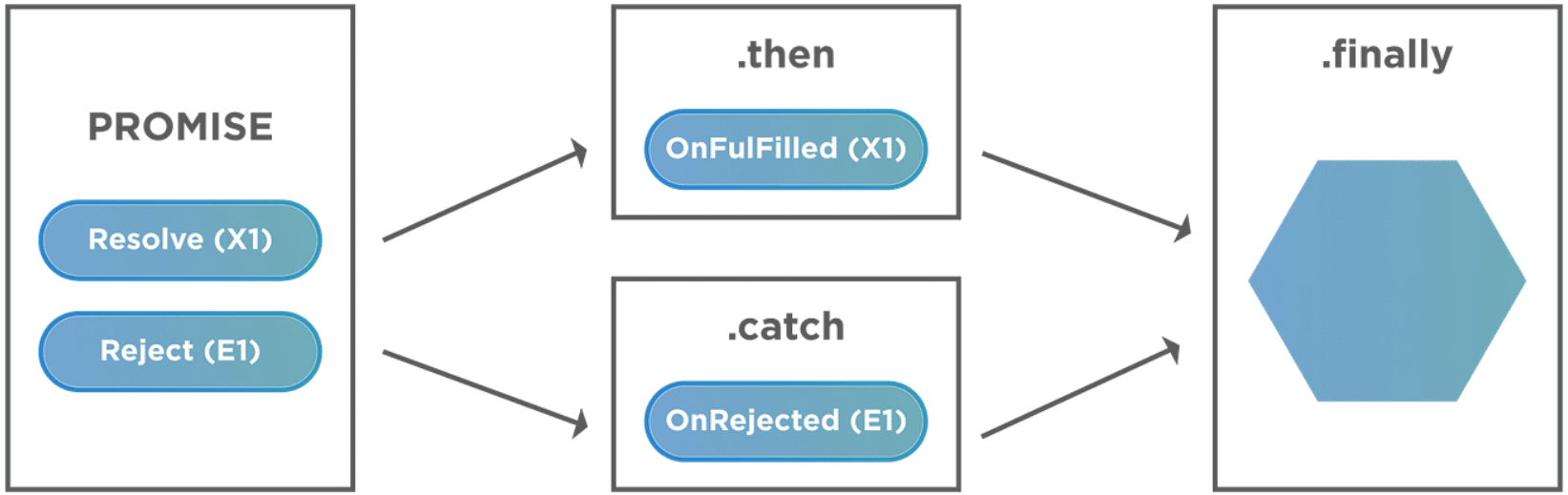
fetch retorna uma Promise

- fetch é uma função em JavaScript que retorna uma Promise.
  - A função fetch é usada para fazer solicitações HTTP assíncronas.
- A Promise retornada pelo fetch tem dois possíveis estados:
  - Pending (Pendente): Quando a solicitação está em andamento, e o resultado ainda não está disponível.
  - Fulfilled (Cumprida): Quando a solicitação foi concluída com sucesso e a resposta HTTP está disponível.
- Você pode usar os métodos .then() e .catch() da Promise retornada pelo fetch para lidar com a resposta da solicitação ou para capturar erros.

exemplo

```
fetch(“uri”)
  .then((response) => {
      return response.text(); // Converte a resposta em texto
  }).then((data) => {
    console.log(data); // Processa os dados da resposta
  }).catch((error) => {
    console.error('Ocorreu um erro:', error);
  });
```

promise


PROMISE
Resolve (X1)
Reject (E1)
.then
OnFulFilled (X1)
.catch
OnRejected (E1)
.finally

É possível criar suas próprias promises

- Quando trabalhos com o fech estamos consumindo promises.
- Ou seja, estamos utilizando promises que foram construídas por outros programadores.
- É possível construir suas próprias promises.

```
<script>
   //criando sua propria Promise
   const minhaPromise = new Promise((resolve, reject) => {
   //realize a operação paralela aqui dentro.
    const sucesso = true; // Altere para false para simular um erro
   if (sucesso) {
     resolve("Operação bem-sucedida!");
   } else {
     reject("Ocorreu um erro na operação!");
   }
});
// Usando a Promise
minhaPromise.then((resultado) => {
  //executado apenas quando for chamado o resolve("Operação bem-sucedida!");
  console.log(resultado); // "Operação bem-sucedida!"
}).catch((erro) => {
   //executado apenas quando for chamado o reject("Ocorreu um erro na operação!");;
   console.error(erro); // "Ocorreu um erro!"
});
</script>
```

Connection pool

- Um piscina de conexões (ou "connection pool", em inglês) é um mecanismo de gerenciamento de conexões de banco de dados em um aplicativo.

- Consiste em um conjunto de conexões de banco de dados pré-criadas, mantidas em um estado ativo e pronto para uso.

- A principal finalidade de um pool de conexões é melhorar o desempenho e a eficiência ao interagir com um banco de dados.

Connection pool
Conexões pré-criadas

- Quando o aplicativo é inicializado, um número definido de conexões de banco de dados é criado e estabelecido com o servidor do banco de dados.
- Essas conexões estão prontas para uso imediato.

Reutilização de conexões:

- Em vez de criar uma nova conexão toda vez que uma operação de banco de dados é necessária, o aplicativo utiliza uma das conexões disponíveis no pool.
- Após o uso, a conexão não é fechada imediatamente; ela é liberada de volta para o pool para ser reutilizada em futuras operações.

Limite de conexões

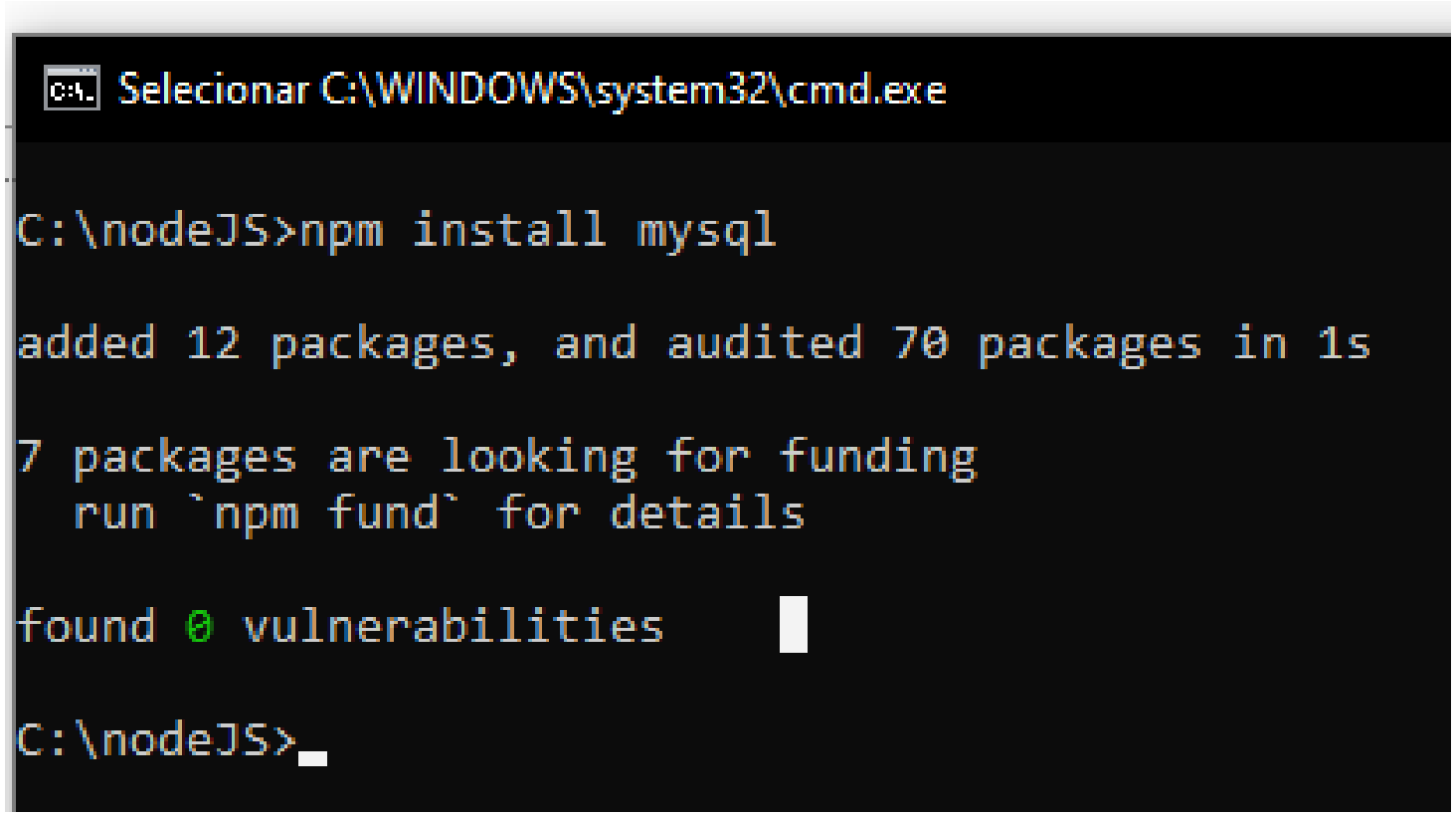
- O pool de conexões tem um limite máximo de conexões simultâneas que podem estar ativas ao mesmo tempo.
- Isso impede que o aplicativo sobrecarregue o servidor do banco de dados com um número excessivo de conexões.

Eficiência e desempenho

- Usar um pool de conexões ajuda a evitar o custo de criar e fechar conexões de banco de dados repetidamente, o que pode ser uma operação demorada e intensiva em recursos.
- Isso melhora a eficiência e o desempenho geral do aplicativo.

Mysql

- Configure o mysql na sua aplicação
- Digite na pasta do seu projeto:
  - C:\nodejs\npm install mysql --save



Criar um pool.

```
var mysql = require('mysql'); //instale antes: npm install mysql --save:
var banco = mysql.createPool({
  connectionLimit: 128, //quantidade de conexões asd
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'paw007',

});
```

Exemplo Completo

Arquitetura

- Modelo
  - Cargo.js
  - Funcionario.js
  - JwtToken
- Rotas
  - rotas_cargos.js
  - rotas_funcionarios
- Views
  - Views
    - Js
      - cargos.js
      - funcionarios.js
      - login.js
      - sessao.js
    - Cargos.html
    - Funcionarios.html
    - Login.html
    - PainelAdm.html
- app.js

Classe: Cargo.js

```
module.exports =
    class Cargo {
        constructor(banco) {
            this.banco = banco;
            this.idCargo = null;
            this.nomeCargo = null;
        }
```

```
async create() {
        const operacaoAssincrona = new Promise((resolve, reject) => {
            const nomeCargo = this.getNomeCargo();
            let parametros = [nomeCargo];
            let sql = "INSERT INTO cargo (nomeCargo) VALUES (?);";
            this.banco.query(sql, parametros, function (error, result) {
                if (error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            });
        });
        return operacaoAssincrona;
    }
```

```
async read() {
        const operacaoAssincrona = new Promise((resolve, reject) => {
            const id = this.getIdCargo();
            let parametros = [id];  let sql = "";
            if (id == null) {
                sql = "SELECT * FROM cargo order by nomeCargo";
            } else {
                sql = "SELECT * FROM cargo where idCargo=?";
            }
            this.banco.query(sql, parametros, function (error, result) {
                if (error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            });
        });
        return operacaoAssincrona;
    }
```

```
async update() {
        const operacaoAssincrona = new Promise((resolve, reject) => {
            const id = this.getIdCargo();
            const nomeCargo = this.getNomeCargo();
            console.log(nomeCargo);
            let parametros = [nomeCargo, id];
            let sql = "update cargo set nomeCargo=? where idCargo = ?";
            this.banco.query(sql, parametros, function (error, result) {
                if (error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            });
        });
        return operacaoAssincrona;
    }
```

```
async delete() {
        const operacaoAssincrona = new Promise((resolve, reject) => {
            let parametros = [this.idCargo];
            let sql = "delete from cargo where idCargo=?";
            this.banco.query(sql, parametros, function (error, result) {
                if (error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            });
        });
        return operacaoAssincrona;
    }
```

```
setIdCargo(idCargo) {
    this.idCargo = idCargo;
}
getIdCargo() {
    return this.idCargo;
}
setNomeCargo(nomeCargo) {
    this.nomeCargo = nomeCargo;
}
getNomeCargo() {
    return this.nomeCargo;
}
```

Classe: Funcionario.js

```
module.exports = class Funcionario {

    constructor(banco) {
        this.banco = banco;
        this.id = null;
        this.nome = null;
        this.email = null;
        this.senha = null;
        this.recebeValeTransporte = null;
        this.cargo = {
            idCargo: null,
            nomeCargo: null
        };
    }
```

```
async create() {
    const operacaoAssincrona = new Promise((resolve, reject) => {
        const nome = this.getNome();
        const email = this.getEmail();
        const senha = md5(this.getSenha());
        const recebeValeTransporte = this.getRecebeValeTransporte();
        const cargo = this.getCargo();
        const Cargo_idCargo = cargo.idCargo;
        const params = [nome, email, senha, recebeValeTransporte, Cargo_idCargo];
        let sql = "INSERT INTO funcionario (nome, email,senha, recebeValeTransporte, Cargo_idCargo) VALUES (?, ?, ?, ?, ?);";
        this.banco.query(sql, params, function (error, result) {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
        });
    });
    return operacaoAssincrona;
}
```

```
async read() {
      const operacaoAssincrona = new Promise((resolve, reject) => {
        const id = this.getIdFuncionario();
         let params = [id];
         let SQL = "";
         if (id == null) {
            SQL = "SELECT idFuncionario,nome,email,recebeValeTransporte,idCargo,nomeCargo FROM funcionario JOIN
cargo ON cargo.idCargo = funcionario.Cargo_idCargo  ORDER BY nome, nomeCargo";
         } else {
            SQL = "SELECT idFuncionario,nome,email,recebeValeTransporte,idCargo,nomeCargo FROM funcionario JOIN
cargo ON cargo.idCargo = funcionario.Cargo_idCargo where idFuncionario=? ORDER BY nome, nomeCargo ";
         }
         this.banco.query(SQL, params, function (error, result) {
            if (error) {
               console.log(error);
               reject(error);
            } else {
               resolve(result);
            }
         });
      });
      return operacaoAssincrona;
   }
```

```
async update() {
    const operacaoAssincrona = new Promise((resolve, reject) => {
        const nome = this.getNome();
        const email = this.getEmail();
        const senha = md5(this.getSenha());
        const recebeValeTransporte = this.getRecebeValeTransporte();
        const Cargo_idCargo = this.getCargo().idCargo;
        const idFuncionario = this.getIdFuncionario();
        let parametros = [nome, email, senha, recebeValeTransporte, Cargo_idCargo, idFuncionario];
        let sql = "update funcionario set nome=?,email=?,senha=?,recebeValeTransporte=?,Cargo_idCargo =? where
idFuncionario = ?";
        this.banco.query(sql, parametros, function (error, result) {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
        });
    });
    return operacaoAssincrona;
}
```

```
async delete() {
    const operacaoAssincrona = new Promise((resolve, reject) => {
        const idFuncionario = this.getIdFuncionario();
        let parametros = [idFuncionario];
        let sql = "delete from funcionario where idFuncionario = ?";
        this.banco.query(sql, parametros, function (error, result) {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
        });
    });
    return operacaoAssincrona;
}
```

```
async login() {
    const operacaoAssincrona = new Promise((resolve, reject) => {
        const email = this.getEmail();
        const senha = md5(this.getSenha());
        let parametros = [email, senha];
        let sql = "SELECT COUNT(*) AS qtd ,nome,email FROM funcionario WHERE email =? AND senha =?";
        const result = this.banco.query(sql, parametros, (error, result) => {
            if (error) {
                reject(error);
            } else {
                if (result[0].qtd == 1) {
                    const resposta = {status: true, nome: result[0].nome, email: result[0].email };
                    resolve(resposta);
                } else {
                    const resposta = { status: false }
                    resolve(resposta);
                }
            }
        });
    });
    return operacaoAssincrona;
}
```

App.js

```
let express = require('express');
var mysql = require('mysql');
var rotas_cargos = require('./rotas/rotas_cargos');  //module you want to include
var rotas_funcionarios = require('./rotas/rotas_funcionarios');  //module you want to include

var app = express();
app.use(express.static('js'));
app.use(express.json())
app.use('/view', express.static(__dirname + '/view'));

let porta = 3000;

var banco = mysql.createPool({
  connectionLimit: 128,
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'paw007'});

rotas_cargos(app,banco);
rotas_funcionarios(app,banco);

app.listen(porta, function () {
  console.log('Servidor Ativo na porta: ' + porta + '!');
});
module.exports = app;
```

rotas_cargos.js

```
app.get('/cargos', (request, response) => {
    console.log("rota: GET: /cargos");
    const dadosAutorizacao = request.headers.authorization;
    const jwt = new JwtToken();
    if (jwt.validarToken(dadosAutorizacao).status == true) {
        const cargo = new Cargo(banco);
        cargo.read().then((resultadosBanco) => {
            const resposta = { status: true, msg: 'Executado com sucesso', codigo: 'cargo_002', dados: resultadosBanco
            }
            response.status(200).send(resposta);
        }).catch((erro) => {
            const resposta = {status: false,msg: 'erro ao executar', codigo: 'cargo_E002',dados: erro}
            response.status(200).send(resposta);
        });;
    } else {
        const resposta = {status: false, msg: 'Usuário não logado',codigo: 401}
        response.status(200).send(resposta);

    }

});
```