Criando e consumindo API REST com PHP

Repositório: https://github.com/estherpeixoto/php-api-rest

O que é uma API?

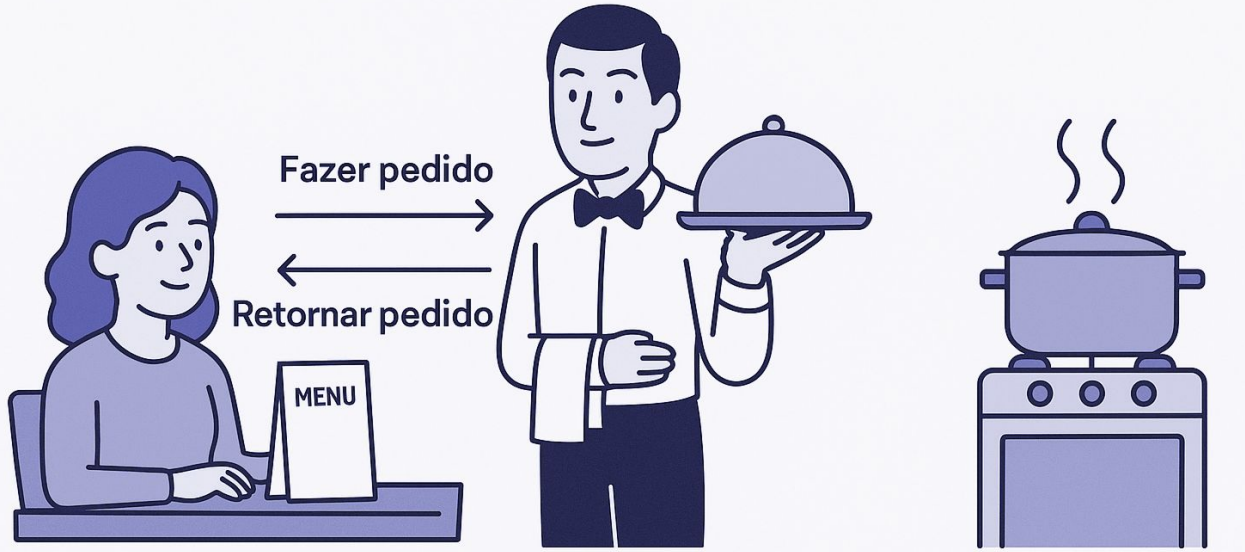
Definição: API (Application Programming Interface*)

Uma API é um sistema que serve de ponte entre outros sistemas externos, permitindo a troca de informações.

Exemplo real: os Correios oferecem uma API para calcular frete. Ela recebe dados como peso, dimensões, origem e destino do pacote, e responde com o valor e prazo de entrega. Essa API se comunica com o sistema principal dos Correios sem que o desenvolvedor precise conhecer sua lógica interna.

* Interface de Programação de Aplicação

Analogia do restaurante



E o que é REST?

Definição: REST (Representational State Transfer*)

É um conjunto de princípios de arquitetura usados para criar APIs simples, escaláveis e compatíveis com a web.

Não é um protocolo, é um estilo de arquitetura.

Usa padrões amplamente conhecidos, como HTTP, URLs e JSON.

Uma API RESTful é uma API que segue os princípios do REST.

* Transferência de Estado Representacional

Princípios do REST

Cliente-Servidor: separação clara entre quem consome e quem fornece os dados.

Stateless (Sem estado): cada requisição carrega todas as informações necessárias. O servidor não guarda contexto (Session no PHP).

Cacheável: respostas podem ser armazenadas em cache para melhorar desempenho e custo.

Interface Uniforme: as APIs seguem convenções previsíveis (ex: usar métodos HTTP).

Sistema em Camadas: a API pode usar camadas intermediárias (como proxies e gateways) sem o cliente saber.

Código sob Demanda (opcional): o servidor pode enviar scripts que o próprio cliente executa.

O que são recursos?

Definição: Um recurso é qualquer entidade que pode ser identificada e manipulada dentro de uma API. Isso inclui objetos tangíveis, como "usuário" ou "produto", e conceitos abstratos, como "sessão" ou "relatório".

Identificação: Cada recurso é acessado por um URI único, por exemplo, `/usuarios/123` para o usuário com ID 123.

Manipulação: Os recursos são manipulados usando métodos HTTP (próximo slide).

Coleções e Sub-recursos: Recursos podem ser agrupados em coleções, como `/usuarios`, e podem ter sub-recursos, como `/usuarios/123/pedidos`, representando os pedidos do usuário com ID 123.

Métodos HTTP

Método/Verbo	Operação	Descrição
`GET`	Ler (Read)	Recupera dados de um recurso.
`POST`	Criar (Create)	Envia dados para criar um novo recurso.
`PUT`	Atualizar (Update)	Substitui completamente os dados de um recurso existente.
`PATCH`	Atualização parcial	Altera parcialmente um recurso (ex: atualizar só o nome).
`DELETE`	Excluir (Delete)	Remove um recurso existente.
`OPTIONS`	Descoberta	Retorna os métodos HTTP suportados por um endpoint.
`HEAD`	Recupera cabeçalhos	Igual ao GET, mas retorna apenas os headers da resposta.

Representações (JSON)

Como os dados são trocados?

- Através de "representações" do recurso.
- O cliente e servidor negociam o formato em JSON.
- JSON (JavaScript Object Notation*) é o mais comum para APIs REST hoje.
    - Leve, fácil de ler por humanos, fácil de interpretar por máquinas.

No nosso projeto:

- Exemplo de JSON: `{"cnpj":"60742855000110"}`

* Notação de Objetos JavaScript

Códigos de status HTTP

Essenciais para o cliente entender se a requisição foi bem-sucedida, falhou, ou precisa de ação adicional. Exemplos que vamos usar:

Código	Quando usar	Exemplo
`200`	Deu certo	Empresa encontrada ✅
`201`	Criou algo novo	Empresa cadastrada ✅
`400`	Cliente errou	JSON malformado ❌
`404`	Não existe	Empresa não encontrada ❌
`500`	Erro no servidor	API offline ❌

Boas práticas

- Use URIs descritivas e padronizadas:
  - **Use substantivos no plural:** `/usuarios` `/produtos`
  - **Evite verbos na URI:** prefira `/usuarios` a `/getUsuarios`
  - **Utilize padrões previsíveis:** `/recurso/:id` para acessar itens específicos
- Utilize os métodos HTTP corretamente.
- Retorne códigos de status HTTP adequados.
- Use JSON como formato padrão.
- Mantenha a API stateless (sem estado).
- Padronize respostas de erro.

Por que usar APIs REST?

Simplicidade: Usa padrões HTTP bem conhecidos.

Flexibilidade: Diferentes tipos de clientes podem consumir.

Escalabilidade: Devido à natureza stateless.

Separação de Interesses: Permite que frontend e backend sejam desenvolvidos e implantados independentemente.

Interoperabilidade: Permite que diversos sistemas trabalhem em conjunto.

REST API na prática

Vamos construir uma API de empresas que:

✅ Cadastra empresa (pega dados da Receita via API externa)

✅ Lista empresas

✅ Busca por ID

✅ Atualiza dados

✅ Remove empresa

Ferramentas:

- PHP 8+
- MySQL
- Composer (gerenciador de pacotes)
- Extensão do VS Code: REST Client (para testar a API)

Vamos ao código

Banco de dados

```
CREATE DATABASE IF NOT EXISTS rest_api;
USE rest_api;
DROP TABLE IF EXISTS empresas;
CREATE TABLE empresas (
    id int NOT NULL AUTO_INCREMENT,
    cnpj varchar(14) NOT NULL,
    razao_social varchar(100) NOT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Criar o projeto com Composer

Crie uma pasta para o projeto e dentro dela execute os comandos em sequência:

```
composer init --name="fasm/mini-curso" --no-interaction
```

```
composer require vlucas/phpdotenv bramus/router
```

Cole o bloco autoload no arquivo **`composer.json`**:

```
"autoload": {
    "psr-4": {
        "App\\": "src/"
    }
}
```

Criar o projeto com PHP

Por fim rode o comando:

```
composer dump-autoload
```

Com isso, todas as nossas classes dentro da pasta `src` ficarão disponíveis através do namespace `App/`

```
{
    "name": "fasm/mini-curso",
    "require": {
        "vlucas/phpdotenv": "^5.6",
        "bramus/router": "^1.6"
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Estrutura de pastas

Nossa API irá seguir uma estrutura baseada no padrão MVC (Model-View-Controller), adaptada para aplicações sem interface visual (View).

Fluxo da aplicação: [Cliente] → [public/index.php] → [Controller] → [Model] → [Banco de dados]

E a resposta JSON volta no caminho inverso.

```
pasta/
├── public/              # Ponto de entrada da aplicação (index.php)
├── src/
│   ├── Controllers/     # Controladores: recebem requisições e coordenam respostas
│   ├── Core/            # Código de suporte (ex: banco de dados, curl e status HTTP)
│   └── Models/          # Modelos: lidam com os dados e a lógica de acesso ao banco
├── vendor/              # Dependências gerenciadas pelo Composer
├── composer.json        # Configuração do projeto e autoload
└── .env                 # Variáveis de ambiente
```

Variáveis de ambiente

Essas variáveis são definidas no arquivo .env e carregadas pela biblioteca Dotenv. Elas permitem separar informações sensíveis e de configuração do código-fonte.

```
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=rest_api
DB_USER=root
DB_PASSWORD=
```

✅ Criar o arquivo `.env` na raiz do projeto

O que é o index.php?

Esse arquivo funciona como o ponto de entrada da aplicação. Ele carrega o ambiente, define rotas HTTP e executa o roteador, que direciona as requisições para o que for definido.

✅ Criar o arquivo `public/index.php`

```php
<?php

use Dotenv\Dotenv;
use Bramus\Router\Router;

// Cria a constante ROOT_PATH com o caminho da pasta raiz do projeto
define('ROOT_PATH', dirname(__DIR__));

// Carrega o autoload do Composer
require ROOT_PATH . '/vendor/autoload.php';

// Carrega as variáveis do arquivo .env
// E as armazenam no $_ENV (ex: $_ENV['DB_HOST'])
$dotenv = Dotenv::createImmutable(ROOT_PATH);
$dotenv->load();

// Cria uma nova instância do roteador
// A partir daqui, podemos definir as rotas da API
$router = new Router();

// Define uma rota do tipo GET para a URL "/"
// Quando acessada, ela executa a função anônima que imprime 'Hello World!'
$router->get('/', function() {
    echo 'Hello World!';
});

// Inicia o roteador e processa a requisição atual
$router->run();
```

Retornando uma resposta JSON

Vamos retornar nosso Hello World como um JSON, definindo o código de status e os headers adequados.

✅ Boas práticas aplicadas:

- Cabeçalho Content-Type garante que o cliente saiba que a resposta é JSON.
- `http_response_code(200)` torna o retorno mais semântico e padronizado.
- Usar `json_encode()` para formatar a resposta.
- `exit;` evita que outros códigos sejam executados após a resposta.

```php
$router->get('/', function() {
    // Define o tipo de conteúdo da resposta
    header('Content-Type: application/json; charset=UTF-8');

    // Define o código de status HTTP (200 = OK)
    http_response_code(200);

    // Envia a resposta no formato JSON
    echo json_encode([
        'mensagem' => 'Hello World!'
    ]);

    // Encerra a execução
    exit;
});
```

Controller Empresa

Este método simula a criação de uma empresa, apenas recebendo os dados enviados via POST e retornando-os para confirmar que chegaram corretamente até o controller.

✅ Criar a rota POST

✅ Criar o arquivo `App/Controllers/EmpresaController.php`

✅ Testando a entrada de dados no controller

```php
$router->post('empresas', '\App\Controllers\EmpresaController@criar');
```

```php
<?php

namespace App\Controllers;

use App\Core\Response;

class EmpresaController
{
    // POST /empresas
    public function criar()
    {
        // Lê e converte o corpo da requisição (JSON) em array associativo
        $dadosJson = json_decode(file_get_contents('php://input'), true);

        // Retorna esses dados para o cliente em formato JSON
        return Response::json(['dadosRecebidos' => $dadosJson]);
    }
}
```

Classe Response

Responsável por enviar a resposta da API em formato JSON, definindo o código de status HTTP e os headers adequados.

Garante que todas as respostas da API sejam consistentes, com encoding e formatação legíveis.

✅ Criar o arquivo `App/Core/Response.php`

```php
<?php

namespace App\Core;

class Response
{
    // Envia uma resposta JSON padronizada
    public static function json(array $response)
    {
        // Define o código de status HTTP
        http_response_code($response['status']);

        // Define o tipo de conteúdo da resposta
        header('Content-Type: application/json; charset=UTF-8');

        // Converte os dados para JSON e envia a resposta
        echo json_encode(
            $response,
            JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE
        );

        // Encerra a execução
        exit;
    }
}
```

Classe Request

Responsável por ler e processar os dados enviados na requisição HTTP.

Utiliza `php://input` para acessar o corpo da requisição e converte o conteúdo JSON em um array associativo.

✅ Criar o arquivo `App/Core/Request.php`

```php
<?php

namespace App\Core;

class Request
{
    // Método responsável por ler o corpo da requisição e retornar os dados como array
    public static function getJson(): array
    {
        // Lê os dados brutos enviados no corpo da requisição
        $input = file_get_contents('php://input');

        // Converte o JSON em um array associativo
        $data = json_decode($input, true);

        // Se a conversão deu certo, retorna os dados
        // Caso contrário, retorna um array vazio
        return is_array($data) ? $data : [];
    }
}
```

Consumindo uma API REST externa

Este método consome a API pública da BrasilAPI para buscar informações de um CNPJ enviado via POST.

Utiliza cURL para fazer uma requisição HTTP externa e retorna a resposta junto com os dados recebidos do cliente.

✅ Testando o endpoint POST /empresas

JSON: `{"cnpj": "60742855000110"}`

```php
// POST /empresas
public function criar()
{
    // Lê e converte o corpo da requisição (JSON) em array associativo
    $dadosJson = Request::getJson();

    // Consumindo uma API REST
    $url = "https://brasilapi.com.br/api/cnpj/v1/{$dadosJson['cnpj']}";

    $ch = curl_init(); // Inicia sessão cURL

    curl_setopt_array($ch, [
        CURLOPT_URL => $url,             // Define a URL
        CURLOPT_RETURNTRANSFER => true,  // Captura a resposta como string
        CURLOPT_SSL_VERIFYPEER => false, // Desativa SSL (útil para localhost)
    ]);

    $response = curl_exec($ch); // Executa a requisição
    curl_close($ch);            // Encerra a sessão cURL

    // Retorna esses dados para o cliente em formato JSON
    return Response::json([
        'dadosRecebidos' => $dadosJson,
        'responseAPI' => json_decode($response, true),
    ]);
}
```

Classe Database

Esta classe gerencia a conexão com o banco de dados de forma centralizada e reutilizável, utilizando PDO.

- Utiliza PDO por ser seguro, flexível e compatível com múltiplos bancos.
- Carrega as configurações do banco a partir das variáveis de ambiente.
- Usa o padrão Singleton, garantindo que apenas uma conexão seja criada e reutilizada durante a execução.

✅ Criar o arquivo
`App/Core/Database.php`

```php
<?php

namespace App\Core;

use PDO;

class Database
{
    // Atributo estático que guarda a instância da conexão com o banco
    // O "?" indica que pode ser null
    private static ?PDO $instance = null;

    // Função pública que retorna a instância única de conexão com o banco
    public static function getInstance(): PDO
    {
        // Se ainda não existe conexão, criamos uma
        if (self::$instance === null) {
            // Pegamos as configurações do banco definidas no .env
            $host     = $_ENV['DB_HOST'];
            $port     = $_ENV['DB_PORT'];
            $dbName   = $_ENV['DB_DATABASE'];
            $username = $_ENV['DB_USER'];
            $password = $_ENV['DB_PASSWORD'];

            // Monta o DSN (Data Source Name) para o MySQL
            $dsn = "mysql:host=$host;port=$port;dbname=$dbName;charset=utf8mb4";

            // Tenta criar a conexão com o banco usando o PDO
            self::$instance = new PDO($dsn, $username, $password, [
                PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, // Erros lançam exceção
                PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC, // Retorna resultados como array associativo
                PDO::ATTR_EMULATE_PREPARES => false, // Usa queries preparadas do MySQL
            ]);
        }

        // Retorna a instância da conexão
        return self::$instance;
    }
}
```

Model Empresa

Esta classe representa a entidade Empresa e é responsável por gerir as informações no banco de dados com segurança usando PDO.

✅ Criar o arquivo
`App/Models/EmpresaModel.php`

```php
<?php

namespace App\Models;

use App\Core\Database;
use Exception;

class EmpresaModel
{
    // Atributo público que armazena uma mensagem de erro, se ocorrer
    // Pode ser null
    public ?string $error = null;

    public function insert(array $data)
    {
        try {
            // Obtém a instância da conexão com o banco (PDO)
            $db = Database::getInstance();

            // Prepara uma query SQL com parâmetros nomeados
            $stmt = $db->prepare('INSERT INTO empresas (cnpj, razao_social) VALUES (:cnpj, :razao_social)');

            // Substitui os parâmetros da query pelos dados fornecidos
            $stmt->bindParam(':cnpj', $data['cnpj']);
            $stmt->bindParam(':razao_social', $data['razao_social']);

            if ($stmt->execute()) {
                // Se a inserção foi bem-sucedida, retorna o ID gerado
                return $db->lastInsertId();
            }
        } catch (Exception $exception) {
            // Em caso de erro, armazena a mensagem para depuração
            $this->error = $exception->getMessage();
        }

        // Se algo falhou, retorna false
        return false;
    }
}
```

Inserir Empresa

Este trecho finaliza o método criar do controller:

✅ Validar a resposta da API externa

✅ Inserir empresa no banco de dados

✅ Retornar um status adequado ao cliente (created ou server error)

```php
// Converte a resposta da API de string para array associativo
$response = json_decode($response, true);

if (!isset($response['razao_social'])) {
    // Se a resposta não contiver a razão social, retorna erro
    return Response::json([
        'status' => 500,
        'message' => "Não foi possível buscar a razão social da empresa {{$dadosJson['cnpj']}}",
    ]);
}

// Instancia o model responsável por lidar com o banco de dados
$model = new EmpresaModel();

// Insere a nova empresa no banco com o CNPJ enviado e a razão social retornada da API
$empresaId = $model->insert([
    'cnpj' => $dadosJson['cnpj'],
    'razao_social' => $response['razao_social'],
]);

if ($empresaId === false) {
    // Se falhar ao inserir no banco, retorna erro do PDO
    return Response::json([
        'status' => 500,
        'message' => $model->error,
    ]);
}

// Retorna sucesso com o ID da nova empresa inserida
return Response::json([
    'status' => 201,
    'message' => 'Empresa criada com sucesso',
    'data' => [
        'id' => $empresaId,
    ],
]);
```

Listar Empresa

Este método busca até 10 empresas cadastradas, ordenadas por razão social.

✅ Criar método findAll() no model

✅ Criar a rota GET

```php
// src/Models/EmpresaModel.php
public function findAll()
{
    try {
        // Obtém a instância da conexão com o banco (PDO)
        $db = Database::getInstance();

        // Executa a consulta SQL que seleciona as empresas ordenadas por razão social
        // Limita o resultado a no máximo 10 registros
        $stmt = $db->query('SELECT id, razao_social, cnpj FROM empresas ORDER BY razao_social LIMIT 10');

        // Retorna todos os resultados como array associativo
        return $stmt->fetchAll();
    } catch (Exception $exception) {
        // Em caso de erro, armazena a mensagem para depuração
        $this->error = $exception->getMessage();
    }

    // Se algo falhou, retorna false
    return false;
}
```

```php
// public/index.php
$router->get('empresas', '\App\Controllers\EmpresaController@listar');
```

Listar Empresa

Este método chama o model para buscar as empresas no banco de dados.

Em caso de erro, retorna uma resposta com status 500. Se der certo, responde com a lista de empresas e status 200.

✅ Criar o método listar no Controller

```php
// src/Controllers/EmpresaController.php
public function listar()
{
    // Instancia o model responsável por lidar com o banco de dados
    $model = new EmpresaModel();

    // Chama o método do model que retorna todas as empresas
    $empresas = $model->findAll();

    // Verifica se houve erro na consulta
    if ($empresas === false) {
        return Response::json([
            'status' => 500,
            'message' => $model->error,
        ]);
    }

    // Retorna as empresas com status 200 e mensagem de sucesso
    return Response::json([
        'status' => 200,
        'message' => 'Sucesso',
        'data' => $empresas,
    ]);
}
```

Buscar Empresa Por ID

Este método busca uma empresa específica pelo ID no banco de dados.

✅ Criar método find() no Model

✅ Criar a rota GET

```php
// src/Models/EmpresaModel.php
public function find(int $id)
{
    try {
        // Obtém a instância da conexão com o banco (PDO)
        $db = Database::getInstance();

        // Prepara uma consulta SQL para buscar a empresa com base no ID
        $stmt = $db->prepare('SELECT id, razao_social, cnpj FROM empresas WHERE id = :id');

        // Substitui o parâmetro :id pelo valor recebido, garantindo segurança contra SQL Injection
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);

        // Executa a consulta no banco
        $stmt->execute();

        // Retorna os dados da empresa como um array associativo
        return $stmt->fetch();
    } catch (Exception $exception) {
        // Em caso de erro, armazena a mensagem para depuração
        $this->error = $exception->getMessage();
    }

    // Se algo falhou, retorna false
    return false;
}
```

```php
// public/index.php
$router->get('empresas/(\d+)', '\App\Controllers\EmpresaController@buscarPorId');
```

Buscar Empresa Por ID

Este método busca uma empresa específica pelo ID.

Retorna 404 se não encontrar e 200 com os dados caso exista.

✅ Criar o método buscarPorId() no Controller

```php
// src/Controllers/EmpresaController.php
public function buscarPorId(int $id): array
{
    // Instancia o model responsável por lidar com o banco de dados
    $model = new EmpresaModel();

    // Busca uma empresa específica pelo ID fornecido
    $empresa = $model->find($id);

    if ($empresa === false) {
        // Se não encontrar a empresa, retorna status 404 (não encontrado)
        return Response::json([
            'status' => 404,
            'message' => 'Empresa não existe',
        ]);
    }

    // Se a empresa for encontrada, retorna os dados com status 200
    return Response::json([
        'status' => 200,
        'message' => 'Sucesso',
        'data' => $empresa,
    ]);
}
```

Atualizar Empresa

Este método atualiza o CNPJ e razão social de uma empresa.

✅ Criar método update() no Model

✅ Criar a rota PUT

```php
// src/Models/EmpresaModel.php
public function update(int $id, array $data)
{
    try {
        // Obtém a instância da conexão com o banco (PDO)
        $db = Database::getInstance();

        // Prepara uma consulta SQL para atualizar a empresa
        $stmt = $db->prepare('UPDATE empresas SET cnpj = :cnpj, razao_social = :razao_social WHERE id = :id');

        // Substitui os parâmetros de forma segura contra SQL Injection
        $stmt->bindParam(':cnpj', $data['cnpj']);
        $stmt->bindParam(':razao_social', $data['razao_social']);
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);

        // Executa o update no banco
        return $stmt->execute();
    } catch (Exception $exception) {
        // Em caso de erro, armazena a mensagem para depuração
        $this->error = $exception->getMessage();
    }

    // Se algo falhou, retorna false
    return false;
}
```

```php
// public/index.php
$router->put('empresas/(\d+)', '\App\Controllers\EmpresaController@atualizar');
```

Atualizar Empresa

Este método verifica se a empresa existe e atualiza o CNPJ e a razão social.

✅ Criar método atualizar() no Controller

```php
// src/Controllers/EmpresaController.php
public function atualizar(int $id)
{
    // Lê e converte o corpo da requisição (JSON) em array associativo
    $dadosJson = Request::getJson();

    // Instancia o model responsável por lidar com o banco de dados
    $model = new EmpresaModel();

    // Verifica se existe uma empresa com o ID fornecido
    $empresa = $model->find($id);

    if ($empresa === false) {
        // Se não encontrar a empresa, retorna status 404 (não encontrado)
        return Response::json([
            'status' => 404,
            'message' => 'Empresa não existe',
        ]);
    }

    // Retorna mensagem de sucesso e status 200
    if ($model->update($id, $dadosJson)) {
        return Response::json([
            'status' => 200,
            'message' => 'Empresa atualizada com sucesso',
        ]);
    }

    // Se falhar ao atualizar no banco, retorna erro do PDO
    return Response::json([
        'status' => 500,
        'message' => $model->error,
    ]);
}
```

Excluir Empresa

Este método exclui uma empresa do banco de dados.

✅ Criar método delete() no Model

✅ Criar a rota DELETE

```php
// src/Models/EmpresaModel.php
public function delete(int $id)
{
    try {
        // Obtém a instância da conexão com o banco (PDO)
        $db = Database::getInstance();

        // Prepara uma consulta SQL para excluir a empresa
        $stmt = $db->prepare('DELETE FROM empresas WHERE id = :id');

        // Substitui o parâmetro :id de forma segura contra SQL Injection
        $stmt->bindParam(':id', $id, PDO::PARAM_INT);

        // Executa o delete no banco
        return $stmt->execute();
    } catch (Exception $exception) {
        // Em caso de erro, armazena a mensagem para depuração
        $this->error = $exception->getMessage();
    }

    // Se algo falhou, retorna false
    return false;
}
```

```php
// public/index.php
$router->delete('empresas/(\d+)', '\App\Controllers\EmpresaController@excluir');
```

Excluir Empresa

Este método verifica se a empresa existe e exclui ela do banco de dados.

✅ Criar método excluir() no Controller

```php
// src/Controllers/EmpresaController.php
public function excluir(int $id): array
{
    // Instancia o model responsável por lidar com o banco de dados
    $model = new EmpresaModel();

    // Verifica se existe uma empresa com o ID fornecido
    $empresa = $model->find($id);

    if ($empresa === false) {
        // Se não encontrar a empresa, retorna status 404 (não encontrado)
        return Response::json([
            'status' => 404,
            'message' => 'Empresa não existe',
        ]);
    }

    if ($model->delete($id)) {
        // Retorna mensagem de sucesso e status 200
        return Response::json([
            'status' => 200,
            'message' => 'Empresa excluída com sucesso',
        ]);
    }

    // Se falhar ao excluir no banco, retorna erro do PDO
    return Response::json([
        'status' => 500,
        'message' => $model->error,
    ]);
}
```

Sugestões de evolução

📖 Documentação da API

- Criar coleções no Postman ou Insomnia para testes.
- Gerar documentação com OpenAPI/Swagger.

🛠️ Ambientes e Deploy

- Adicionar suporte a Docker para facilitar a implantação.

🔐 Autenticação com JWT (JSON Web Token).

- Detalhes no próximo slide.

🔐 Autenticação com JWT (JSON Web Token)

1. Fluxo de autenticação: O usuário envia e-mail e senha para /login. Se as credenciais forem válidas, a API retorna um token JWT com informações do usuário e data de expiração.
2. Requisições autenticadas: O cliente envia o token JWT no cabeçalho da requisição (Authorization: Bearer <token>). A API valida a assinatura e a validade do token para permitir o acesso ao recurso.
3. Para gerar JWT com PHP, utilize a biblioteca firebase/php-jwt, definindo os dados do usuário, adicionando uma data de expiração e assinando o token com uma chave secreta.

Obrigada por acompanhar até aqui!

Agora é hora de praticar e explorar por conta própria 😉

Repositório: https://github.com/estherpeixoto/php-api-rest