**JSON**

**JSON =** JavaScript Object Notation;

Suportada via texto e pode ser interpretada por outras linguagens, logo “Universal”;

É similar a síntaxe dos objetos JavaScript;

Mais simples q o XML e utilizado para fins parecidos;

Formato de chave e valor;

Leve para ser enviado por requisições;

Muito utilizado para APIs e arquivos de config.

**Tipos de dados**

Strings – “blabla”;

Números – 1, 22.33;

Arrays – [1, 2, 3];

Objetos –

{

“nome”: “Matheus”,

“idade”:12

};

Dados nulos – null.

coments

A chave precisa ser compostas com aspas duplas

**API**

**A**pplication **P**rogramming **I**nterface (Interface de Programação de Aplicações);

Uma requisição e uma resposta;

Simplifica o desenvolvimento, não necessita de telas;

Antes, eram feitas telas cada funcionalidade, desde a do cliente, até a do banco de dados. Com a API, não são necessárias telas, mas sim, funcionalidades, recursos, melhorando essa questão de desenvolvimento;

Fácil integração com o sistema, apenas dando respostas, por ex.: se necessário ligar uma api a um sistema, só preciso saber se ela funciona, fazer as requisições para lá e receber as respostas que quero;

Padrão de desenvolvimento, permitindo trocas de API;

Performance por causa das requisições e respostas rápidas, ex. prático: cotação de frete.

Basicamente uma interface onde será enviada uma requisição e esperada uma resposta.

**Tipos de API**

**API pública:** disponibilizada para qualquer um acessar, sem necessidade de autenticação, um exemplo seria o GitHub;

**API privada:** quando uma API é criada para utilização de sistemas de uma empresa que visa em privacidade de dados, requer autenticação, sistemas internos de uma empresa;

**API de terceiros:** quando a API é um produto, podendo ser integrada em um sistema, requer autenticação. Um ex.: desenvolve um sistema mas só acessa ou usufrui quem paga.

**Comunicação com API**

Normalmente feita através de XML ou JSON devido a simplicidade destes tipos de dados;

Requisições e respostas são feitas através do HTTP, basicamente para condicionar os verbos e status utilizados;

Onde podemos seguir um padrão, para criação de APIs robustas e que serão de fácil utilização.

**Verbos do HTTP**

Alterando os verbos podemos realizar requisições diferentes, simplificando a construção da nossa API e seus endpoints;

**GET:** verbo para resgatar algum recurso, por exemplo, resgatar usuários;

**POST:** enviar dados da API, por exemplo, adicionar usuários;

**DELETE:** remover algum registro do banco, por exemplo, remover usuários;

**PUT:** atualização(completamente ) de registros, por exemplo, uma atualização completa do usuário;

**PATCH:** atualização parcial de registros, por exemplo, a atualização de um dado do usuário, o nome, a idade.

**O que é endpoint?**

A URL que acessamos da API para extrair uma resposta;

Ex.: /users/1 – GET

Obviamente é também acompanhado do domínio da API, poder ser o ip do servidor, o domínio que a impresa colocou para o acesso da API, etc;

Mesmo endpoints com verbos diferentes podem ter resultados diferentes;

Uma das técnicas para reduzir a complexidade e aplicar o padrão REST.

**O que é REST?**

Um padrão/conceito de arquitetura ligado a APIs e o protocolo HTTP, convenções a serem seguidas e ser criada uma API RESTful;

Podemos implementar este padrão, que é composto de 6 partes para tornar a API em **RESTful**;

São eles:

Uniformidade – requisições vindas de locais diferentes, se forem para o mesmo endpoint, obtém as mesmas respostas;

Desacoplação – API deve ser independente do Client, API tem que receber uma requisição e enviar uma resposta independente do sistema dela;

Stateless – requisição única, não há a ligação com a API, cada requisição vai ser independente;

Cache – responder requisições diferentes com as mesmas respostas, não teria repetições;

Arquitetura de Camadas – hierarquias no código;

Code on demand – opcional, código roda por demanda.

**Postman**

Software que permite testar as APIs que criamos;

Pode-se inserir um endpoint, método HTTP, inserir dados no corpo da requisição, entre outras coisas;

Muito utilizado na indústria de software.

**Promises**

Como promessas na realidade, ela está esperando algum retorno;

Para a criação, instancia-se a **classe Promise**;

Leva dois argumentos **resolve**(solução) e **reject**(erro);

Para encadear mais processos utilizamos o **método then**;

Alguns recursos de JS (**Fetch API**) e bibliotecas retornam Promises.

**.then**

//.then - esta função possui dois parâmetros para lidar com o sucesso ou rejeição da promessa

//podem ser encadeados diversos then's

**toLowerCase**

toLowerCase()//conversao para letra minúscula

})

**fetch**

//fetch - retorna uma promessa

**.catch**

//catch - trata de erros

//para processos geralmente utiliza-se .then e .catch juntos

**all**

//all - recebe array de promessas e entrega resposta apenas quando todas estão resolvidas

**race**

//race - recebe array de promessa, retorna mensagem que deu fim na promessa para primeira promessa resolvida

**Fetch API**

Um meio nativo de realizar requisições a APIs com JS;

Utilizado com Promise ou asyn/await, com aync/await é mais intuitivo;

Se a requisição for GET é preciso passar apenas a URL do recurso resgatado da API;

Caso seja outro verbo, pode-se configurar a requisição com: **method**, **body** e **headers**;

Muito utilizado com frameworks/libs, como: **React**, **Vue** e **Angular**.

**REQUEST E RESPONSE**

Método de comunicação entre computadores em rede;

Esse método de comunicação é síncrono – depois que a requisição é feita ele fica aguardando a resposta;

Existe um tempo limite para isso;

Request (Requisição) – Indica a comunicação do cliente para o servidor