



otterwise



# Módulo WEB Developer



# JSON



otterwise

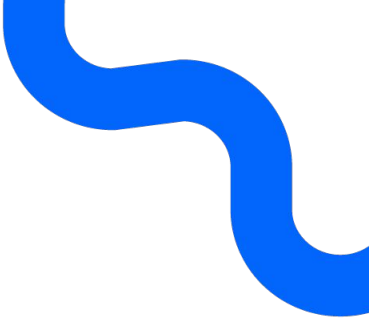
# JSON



- JavaScript Object Notation.
- Formato de dados para interpretação e comunicação.
- Fácil entendimento tanto para pessoas quanto para máquinas.
- **Semelhante** ao Object do JS.
- Tipos permitidos:
  - string.
  - number.
  - boolean.
  - null.
  - object.
  - array.

Fonte: <https://pt.wikipedia.org/wiki/JSON>

# JSON



```
testes - data.json
1  {
2    "id": 1,
3    "name": "Biblioteca do Juca",
4    "books": [
5      { "id": 1, "title": "Comunicação Não-Violenta" },
6      { "id": 2, "title": "Obrigado pelo Feedback" },
7      { "id": 3, "title": "Mindset" }
8    ]
9  }
10
```



# MÓDULOS

# Trabalhando com módulos

- Import e export em JS;
- Separar códigos e funções por contexto para reutilizá-los;
- Melhor legibilidade e manutenibilidade do projeto.

Exemplo de import/export nomeado ->

```
// hello.js  
  
export const hello = () => console.log('Hello World')
```

```
// main.js  
  
import { hello } from './hello.js'  
  
hello() // "Hello World"
```

# Trabalhando com módulos



Exemplo de import/export padrão

```
// hello.js

const hello = () => console.log('Hello World')

export default hello
```

```
// main.js

import helloFunction from './hello.js'

helloFunction() // "Hello World"
```





# GERENCIADORES DE PACOTES

# Gerenciadores de pacotes



- Ferramentas que irão gerir os pacotes que temos no nosso projeto ou no nosso workspace (global).
- Utilizados para instalar, remover e atualizar nossos pacotes, em conjunto ou individualmente.
- Podem ser utilizados para executar scripts como build, start, etc...
- Exemplos: yarn, npm, etc...

# package.json



- Arquivo responsável por informações pertinentes do projeto além de pacotes, dependências, scripts.
- Criando um arquivo **package.json** no seu projeto:
  - Utilizando o gerenciador **Yarn**: use o comando **yarn init** e siga os passos no prompt.
  - Utilizando o gerenciador **NPM**: use o comando **npm init** e siga os passos no prompt.



# EXERCÍCIOS

## Exercícios



1. Crie um arquivo chamado `helpers.js` e nele crie uma função que multiplique 2 números qualquer. Faça o `export` dessa função e em outro arquivo chamado `index.js` importe a função e a execute.



# GERENCIAMENTO DE PACOTES

# Gerenciamento de pacotes



- Possibilidade de importar na nossa aplicação códigos de terceiro, disponibilizados publicamente, em forma de pacotes.
- Exemplo: **moment.js** e **date-fns** (pacotes de datas), **axios** (pacote para realizar requests HTTP), **react** (pacote para criação de interfaces de usuário), etc...
- Os pacotes vão facilitar o desenvolvimento, adicionando novas ferramentas e novas possibilidades para a aplicação.

# Pacotes



- Quando instalamos o primeiro pacote é criado uma **pasta** chamada **node\_modules** onde ficarão os códigos de terceiro que nosso projeto utiliza.
- Além disso um **arquivo lock** será criado para identificar a versão e a integridade dos pacotes instalados que estão dentro da **pasta node\_modules**.
- Dentro do arquivo package.json ficará a versão do pacote que estamos usando no projeto. Essa informação pode ou não permitir que o pacote seja atualizado automaticamente.



## Pontos de atenção



- Quando subimos nosso projeto para um repositório remoto **não** precisamos subir a pasta **node\_modules**.
- Quando clonamos ou baixamos um projeto que possui um arquivo package.json utilizamos o comando **yarn** ou **npm install** para que os pacotes e dependências sejam instalados, criando assim a pasta **node\_modules** e o arquivo **lock**.
- Tanto utilizando o Yarn ou NPM temos a pasta **node\_modules**.
- O arquivo lock muda dependendo do gerenciador de pacotes que estamos utilizando, mas ele tem sempre o mesmo propósito:
- Com o Yarn o arquivo lock se chama **yarn.lock**.
- Com o NPM o arquivo lock se chama **package-lock.json**.
- Podemos instalar uma versão específica de um pacote.



# EXERCÍCIOS

# Exercícios



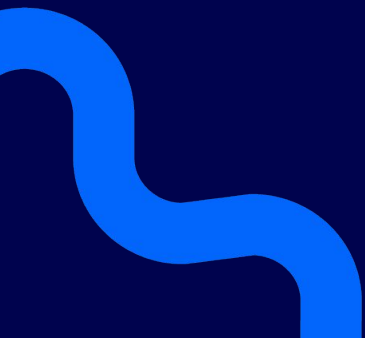
1. Utilizando o **Yarn** ou **NPM**, crie o arquivo package.json e configure seu projeto.
2. Adicione ao seu projeto o pacote **date-fns**.
3. Crie um arquivo **formatters.js**:
  - a. Nesse arquivo crie uma função chamada **formatDate** que recebe uma data como parâmetro e, utilizando a **date-fns**, retorna essa data no formato "DD/MM/YYYY".
  - b. Esse arquivo deve exportar a função formatDate.

**Dica:** para criar uma data com javascript utilize: new Date()



# Contexto Web

Módulo WEB Developer



# Comunicação com API's



# O que é uma API?

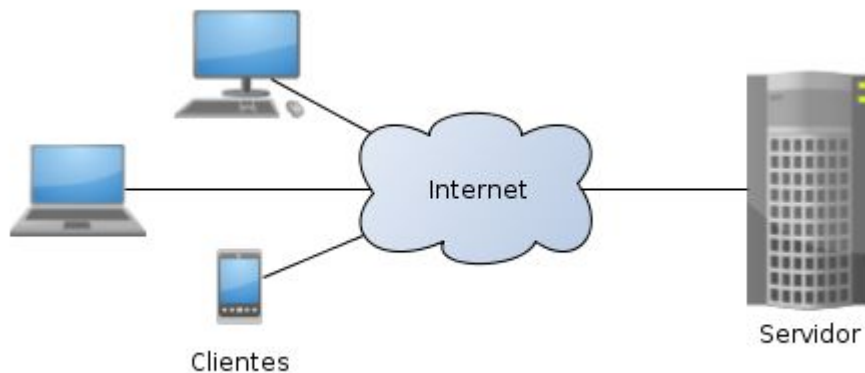


API é um conjunto de definições e protocolos usado no desenvolvimento e na integração de software de aplicações. API é um acrônimo em inglês que significa interface de programação de aplicações.

Fonte: <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>

# Modelo Cliente Servidor

É o modelo onde vários clientes (computadores, celulares, etc) se comunicam com um servidor que fornece informações necessárias para esses clientes.



Fonte: [https://pt.wikipedia.org/wiki/Modelo\\_cliente%E2%80%93servidor](https://pt.wikipedia.org/wiki/Modelo_cliente%E2%80%93servidor)

# Requisições





# Promises



Utilizamos Promises no javascript para fazer requisições. Uma promise (de “promessa”) representa um valor que pode estar disponível agora, no futuro ou nunca.

Uma promise pode está sempre em um destes estados:

- pending (pendente): Estado inicial, que não foi realizada nem rejeitada.
- fulfilled (realizada): sucesso na operação.
- rejected (rejeitado): falha na operação.

Fonte: [Promise - JavaScript | MDN](#)

# Sintaxe de Promise



```
const promise = new Promise((resolve, reject) => {
  const string1 = "otterwise"
  const string2 = "otterwise"
  if (string1 === string2) {
    resolve()
  } else {
    reject()
  }
})

promise
  .then(function () {
    console.log("Promise foi resolvida com sucesso");
  })
  .catch(function () {
    console.log("Promise foi rejeitada");
  });
```

# Sintaxe async/await

O try/catch é utilizado geralmente em funções assíncronas para capturar os erros de uma promise.

```
const helperPromise = function () {
  const promise = new Promise((resolve, reject) => {
    const string1 = "otterwise"
    const string2 = "otterwise"
    if (string1 === string2) {
      resolve()
    } else {
      reject()
    }
  })
  return promise
}

async function demoPromise() {
  try {
    let message = await helperPromise();
    console.log("Promise resolvida com sucesso");
  } catch (error) {
    console.log("Erro: " + error);
  }
}

demoPromise();
```

# Métodos de Promise

**Promise.all:** permite que executemos diversas promises em paralelo. Aguarda todas as promises finalizarem para dar a operação como finalizada.

```
const helperPromise = function () {  
  const promise = new Promise((resolve, reject) => {  
    const string1 = "otterwise"  
    const string2 = "otterwise"  
    if (string1 === string2) {  
      resolve()  
    } else {  
      reject()  
    }  
  })  
  return promise  
}  
  
async function demoPromise() {  
  try {  
    let message = await Promise.all([helperPromise(),  
    helperPromise()]);  
    console.log("Promise resolvida com sucesso");  
  } catch (error) {  
    console.log("Erro: " + error);  
  }  
}  
  
demoPromise();
```

# Métodos de Promise

**Promise.race:** quando há redundância de serviço podemos realizar uma “corrida” entre as promises. Assim que a primeira promise resolver, a operação é dada como finalizada.

```
const helperPromise = function () {
  const promise = new Promise((resolve, reject) => {
    const string1 = "otterwise"
    const string2 = "otterwise"
    if (string1 === string2) {
      resolve()
    } else {
      reject()
    }
  })
  return promise
}

async function demoPromise() {
  try {
    let message = await Promise.race([helperPromise(),
    helperPromise()]);
    console.log("Promise resolvida com sucesso");
  } catch (error) {
    console.log("Erro: " + error);
  }
}

demoPromise();
```

# Métodos HTTP



# Métodos HTTP



São métodos utilizados para interagirmos com API's e cada método tem um significado. Esses métodos utilizam o padrão de comunicação HTTP, que é nada mais do que um padrão de comunicação entre serviços na web.

Tipos de método HTTP e seus significados:

- OPTIONS
  - Retorna os verbos http de um recurso e outras opções.
- GET
  - Busca por um recurso.
- POST
  - Cria um recurso
- PUT
  - Atualiza um recurso
- PATCH
  - Atualiza parcialmente um recurso
- DELETE
  - Remove um recurso

# Padrão REST





# Padrões de comunicação



- Representational State Transfer (REST)
  - Define um conjunto de restrições a serem utilizadas para a criação de serviços web.
  - Na arquitetura REST, os clientes enviam solicitações para recuperar ou modificar recursos e os servidores enviam respostas para essas solicitações.
  - O servidor não mantém estado. Cada solicitação do cliente deve conter informações necessárias para o server entender a solicitação. O estado da sessão é mantido inteiramente no cliente.
  - Identificado por recursos. Por exemplo, se quisermos uma lista de usuarios, chamados da seguinte forma:
    - GET <http://api.project.com/users>
    - Dessa forma estamos falando para o servidor que queremos uma lista de usuários



# Padrões de comunicação

- Mais alguns exemplos (utilizar o node-fetch para fazer as requisições):
  - Precisamos do usuário com id 1
    - GET para o endpoint <http://api.project.com/users/1>
  - Cadastrar um novo usuário
    - POST para o endpoint <http://api.project.com/users> enviando o objeto com as informações desse usuário
  - Editar todas as informações de um usuário com id 1
    - PUT para o endpoint <http://api.project.com/users/1> enviando o objeto com as novas informações desse usuário
  - Excluir o usuário com id 1
    - DELETE para o endpoint <http://api.project.com/users/1>
  - Editar apenas uma informação do usuário com id 1
    - PATCH para o endpoint <http://api.project.com/users/1> enviando o objeto apenas com as informações que queremos alterar

# Padrões de comunicação



- Mais alguns exemplos:
  - Todos os posts que o usuário 1 tem cadastrado
    - GET para o endpoint <http://api.project.com/users/1/posts>

Outros padrões de comunicação que são utilizados no mercado:

- GraphQL: <https://www.redhat.com/pt-br/topics/api/what-is-graphql>
- SOAP: <https://pt.wikipedia.org/wiki/SOAP>

# Códigos de Status



- 100 - 199: Respostas de informação
  - Ex.:
    - **100 Continue:** Essa resposta provisória indica que tudo ocorreu bem até agora e que o cliente deve continuar com a requisição ou ignorar se já concluiu o que gostaria.
- 200 - 299: Respostas de sucesso
  - Ex.:
    - **200 OK:** Requisição bem sucedida.
- 300 - 399: Respostas de direcionamento
  - Ex.:
    - **301 Moved Permanently:** Esse código significa que a URI do recurso mudou
- 400 - 499: Erros do cliente
  - Ex.:
    - **400 Bad Request:** Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida.
- 500 - 599: Erros do servidor
  - Ex.:
    - **500 Internal Server Error:** O servidor encontrou uma situação com a qual não sabe lidar.

Fonte: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>



# EXERCÍCIOS

# Exercícios



1. Crie um projeto novo e instale a biblioteca axios.
  - a. Consuma a lista de posts através do endpoint <https://jsonplaceholder.typicode.com/posts>.
  - b. Agora que você tem a lista de posts, consulte o primeiro post da lista.
  - c. Cadastre um novo post enviando um objeto com os atributos title e body para o endpoint <https://jsonplaceholder.typicode.com/posts>.
  - d. Faça a edição de um post enviando novos atributos title e body para o post com id 1.
  - e. Faça a exclusão do primeiro post vindo da listagem de posts.