



# Promises

## **R**esumo

Um objeto Promise representa uma operação que produziu ou vai produzir um valor. As promessas fornecem uma maneira robusta de encapsular o resultado (possivelmente pendente) do trabalho assíncrono, mitigando o problema de retornos de chamadas aninhadas. Nesta aula, vamos tratar sobre como trabalhar usando promise e as melhores práticas utilizadas.

## Estados e fluxo de controle


Promise pode estar em um dos três estados:

**Pendente** - A operação ainda não foi concluída e a promise está pendente de execução.

**Executada** - A operação foi concluída e a promise é retornada com um valor. Isso é análogo ao retorno de um valor de uma função síncrona.

**Rejeitado** - Ocorreu um erro durante a operação e a promise é rejeitada com um motivo. Isso é análogo a lançar um erro em uma função síncrona.

Diz-se que uma promise é resolvida quando é cumprida (executada) ou rejeitada. Uma vez que uma promise estabelecida, ela se torna imutável. Os métodos then e catch a

uma promise podem ser usados para anexar retornos de chamada que são executados quando ela é finalizada. , se os retornos de chamada são invocados com o valor de retorno e o motivo de rejeição, respectivamente. Abaixo, veja a estrutura básica de como podemos usar promise:

```
const promise = new Promise((resolve, reject) => {  
  // Executando algum trabalho  
  // ...  
  if (/* Se o trabalho for executado com sucesso é retornado um valor */) {  
    resolve(value);  
  } else {  
    // Se alguma coisa der errado temos que entender o erro.  
    // Por default é retornado um Objeto Error  
  
    let reason = new Error(message);  
    reject(reason);  
    // Vamos lançar um erro e rejeitar a promise.  
  
    throw reason;  
  }  
});
```

Fonte: Autoral

Os métodos then e catch podem ser usados para anexar retornos de chamada que foram atendidas e rejeição:

```
promise.then(value => {  
  // Trabalho executado completamente com sucesso  
  // Temos um valor de retorno  
  
}).catch(reason => {  
  // Alguma coisa deu errado  
  // promise será rejeitada com uma razão  
});
```

Fonte: Autoral



Vamos trazer um exemplo mais prático do que podemos fazer:

```
function exibir(valor) {  
  console.log("Valor da promise: " + valor)  
}
```

Definimos uma função que será usada no retorno da promise se ela der certo ou não.

```
let minhaPromise = new Promise(function(resolvida, rejeitada) {  
  let x = 0;  
  
  // Fazendo a validação  
  if (x == 0) {  
    resolvida("Deu tudo certo!");  
  } else {  
    rejeitada("Aconteceu um erro!");  
  }  
});  
  
minhaPromise.then(  
  function(value) {  
    exibir(value);  
  },  
  function(error) {  
    exibir(error);  
  }  
);
```

Fonte: Autoral

Quando colocarmos o código para executar teremos o resultado que ocorreu tudo correto. Sendo assim, nosso retorno será:



Valor da promise: Deu tudo certo!

Fonte: Autoral

Outra forma de fazer usando arrow function

```
let minhaPromise = new Promise(function(resolvida, rejeitada) {
  let x = 0;

  // Fazendo a validação
  if (x == 0) {
    resolvida("Deu tudo certo!");
  } else {
    rejeitada("Aconteceu um erro!");
  }
});

minhaPromise.then(
  (result) =>{
    console.log("Valor da promise: " + result)
  },(error) =>{
    console.log("Aconteceu algum erro: " + error)
  }
);
```

Fonte: Autoral

Perceba que não usamos uma função externa para passar o resultado da promise. Tratamos o resultado com a arrow function.

Vou trazer aqui alguns exemplos bem comuns de uso do promise.

Geralmente, usamos o promise para tratar o retorno de uma requisição de uma API Rest. Normalmente utilizamos o fetch para

fazer isso por nós.



```
fetch(url)
  .then(response => {
    // comportamento do response
  })
  .catch(error => {
    // comportamento se der erro
  });
```


Fonte: Autoral

O objeto response vai receber o retorno dos dados da requisição. O catch será chamado, no caso de ocorrer algum problema na requisição.

## Conteúdo Bônus


API RESTful é uma interface onde dois ou mais sistemas fazem uso para trocar informação. O termo API se refere a interface de programação de aplicação e é usado para definir regras de como as aplicações deverão se comunicar. RESTful é um termo usado para os serviços da Web que implementam a arquitetura REST.

Representational state transfer (REST) ou transferência de estado representacional é uma arquitetura de software que determina como uma API deve funcionar. REST é uma arquitetura de alta performance, confiável e escalável. As principais características dessa arquitetura são:

- **Interface uniforme:** os dados que trafegam seguem um padrão uniforme. 
- **Ausência de estado:** toda requisição deve ser completada, independente de todas as solicitações anteriores. O cliente pode solicitar recursos em qualquer ordem e o resultado de uma solicitação não interfere na outra.
- **Sistema em camadas:** um serviço pode ser usado para se conectar a outros serviços, ou até mesmo como um serviço intermediário, gerando assim um sistema com muitas camadas, onde um serviço A faz uso do serviço B que faz uso do serviço C e assim por diante.
- **Armazenamento:** essa arquitetura possibilita o armazenamento de determinadas informações em cache, melhorando o tempo de resposta do servidor.
- **Código sob demanda:** o servidor pode modificar ou personalizar a interface do usuário, como por exemplo um usuário administrador ter acesso a uma tela que os demais não têm. Essa informação, se o usuário é administrador ou não, vem do servidor.

Podemos criar um API usando qualquer arquitetura, mas quando o fazemos usando REST, chamamos de API REST. Os serviços da Web que implementam a arquitetura REST são chamados de Web RESTful e o termo API RESTful é utilizado para fazer referência a APIs da Web RESTful.

Mas afinal de contas, como as APIs RESTful funcionam?

Eles funcionam basicamente como um navegador da internet, onde o cliente busca por uma URL que existe e, ao tentar ssar aquela URL, o cliente na verdade está fazendo uma requisição ao servidor, que por sua vez envia uma resposta para o cliente no formato HTML. Ou seja, estamos falando de requisição e resposta.

As APIs RESTful funcionam seguindo os 4 passos a seguir:

- O cliente envia uma solicitação ao servidor, seguindo a documentação da API.
- O servidor verifica que se o cliente tem as permissões necessárias para acessar tal recurso.
- O servidor processa a solicitação internamente feita pelo cliente.
- O servidor retorna a resposta para o cliente. A resposta contém as informações solicitadas pelo cliente e indica se a solicitação foi bem-sucedida ou não.

Toda solicitação a API RESTful deve conter pelo menos dois componentes principais:

- **Identificador de recurso exclusivo:** este componente nada mais é que a url do recurso. É mais conhecido como endpoint, ou seja, o caminho que devemos fazer a requisição para acessar determinado recurso.
- **Método:** API RESTful geralmente são implementadas usando métodos HTTP (Protocolo de Transferência de Hipertexto).

## Principais métodos HTTP



- **GET:** O método Get é usado para buscar recursos no servidor localizados na url especificada.
- **POST:** Os clientes usam Post para enviar dados ao servidor. Devemos, neste caso, passar a representação dos dados que queremos enviar para o servidor junto com a requisição.
- **PUT:** Método usado para atualizar recursos no servidor.
- **DELETE:** Usado para remover recursos no servidor.

## Referência Bibliográfica

FLANAGAN, David. **JavaScript: O Guia Definitivo**. 6ª Ed. Porto Alegre: Bookman, 2013.

FREEMAN, Eric. **Use a cabeça!: programação JavaScript**. 1ª Ed. São Paulo: Alta Books, 2016.

## ATIVIDADE PRÁTICA

**Título da Prática:** O que é uma Promise?

**Objetivos:** Implementar uso de promise

**Materiais, Métodos e Ferramentas:** Para realizar esta prática vamos utilizar o Visual Studio Code.



## Prática



Essa prática consiste em criar um código em JavaScript que utilize Promises, arrays, funções e o recurso 'setTimeout'. O objetivo é processar um array de números de forma assíncrona, verificando se cada número é par ou ímpar e exibindo uma mensagem correspondente para cada um.

Para isso, é necessário implementar a função 'processarItem', que recebe um número como argumento. Essa função retorna uma Promise que será resolvida após um atraso de 1 segundo, simulado pelo 'setTimeout'. Caso o número seja par, a Promise é resolvida com uma mensagem informando que o número é par. Caso contrário, a Promise é rejeitada com uma mensagem indicando que o número é ímpar.

Além disso, deve ser criada a função 'processarArray', que recebe um array de números como parâmetro. Essa função mapeia cada número do array para uma chamada de 'processarItem' correspondente, gerando um array de Promises. Em seguida, utiliza o 'Promise.all' para aguardar a resolução de todas as Promises.

No código principal, um array de números é definido e passado como argumento para a função 'processarArray'. Em seguida, são encadeados os métodos 'then' para tratar os resultados das Promises resolvidas e 'catch' para lidar com possíveis erros.

O enunciado desafia o desenvolvedor a utilizar os conceitos de Promises, arrays, funções e a função 'setTimeout' para criar uma solução que processe um array de números de forma assíncrona e exiba as mensagens correspondentes para cada número. A solução deve demonstrar um bom entendimento desses conceitos e fornecer uma implementação funcional e eficiente.

Boa sorte!



## Resolução

```
function processarItem(item) {  
  
  return new Promise((resolve, reject) => {  
  
    setTimeout(() => {  
  
      if (item % 2 === 0) {  
  
        resolve('O número ${item} é par.');  
      } else {  
  
        reject('O número ${item} é ímpar.');  
      }  
  
    }, 1000);  
  
  });  
  
}  
  
function processarArray(array) {  
  
  const promises = array.map(item => processarItem(item));  
  
  return Promise.all(promises);  
  
}  
  
const arrayNumeros = [1, 2, 3, 4, 5];  
  
processarArray(arrayNumeros)
```

```
.then(resultados => {
```

```
  resultados.forEach(resultado => console.log(resultado));
```

```
})
```

```
.catch(erro => {
```

```
  console.error(erro);
```

```
});
```



**[Ir para exercício](#)**