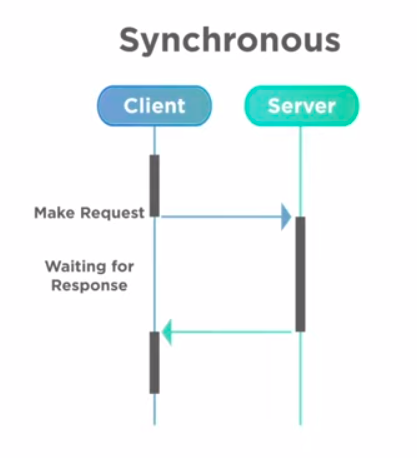
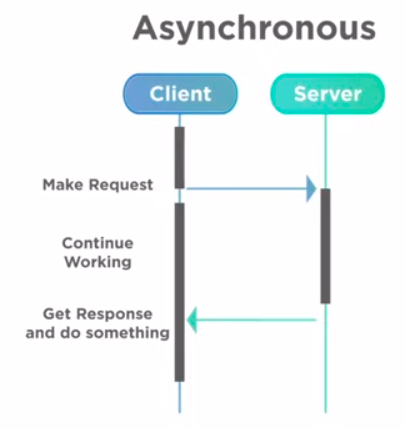
**Definição de Assíncrono/Síncrono:**

**Assíncrono** é algo que não ocorre ou se efetiva ao mesmo tempo. Exemplos: as aulas remotas nas modalidades **assíncronas**, os vídeos da DIO por exemplo, que se pode assistir quando quiser, não são aulas ao vivo, ou seja, **síncronas**, onde precisamos estar presentes num determinado momento, assim como as aulas presenciais.

**O Javascript roda de maneira** **síncrona,** ou seja, as atividades são feitas uma de cada vez, aguardando por sua resposta, logo depois da sua solicitação ter sido feita:



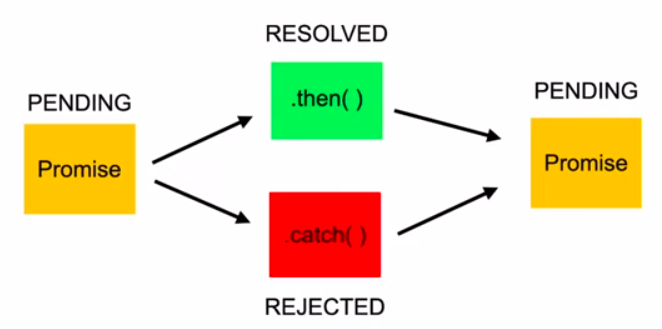
Já as atividades **assíncronas** podem ter mais de uma por vez:



**Promises (Javascript Assíncrono)**

Promises é um objeto de processamento *assíncrono*. Inicialmente, o valor de uma **promise** é desconhecido. Ela pode então ser **resolvida** ou **rejeitada**.

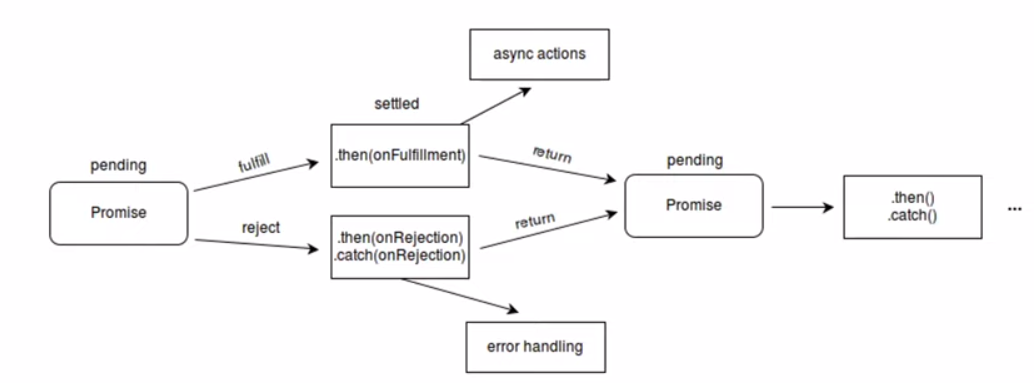
Fazendo uma analogia com o ingresso para um cinema: compramos o ingresso para um determinado dia e horário, o ingresso é a **promise**, pois só temos a promessa de que veremos o filme, mas a atividade em si (o resultado) não foi entregue realmente. Ou seja, o filme pode passar ou não, ele pode ser **resolvido** ou **rejeitado**:

****

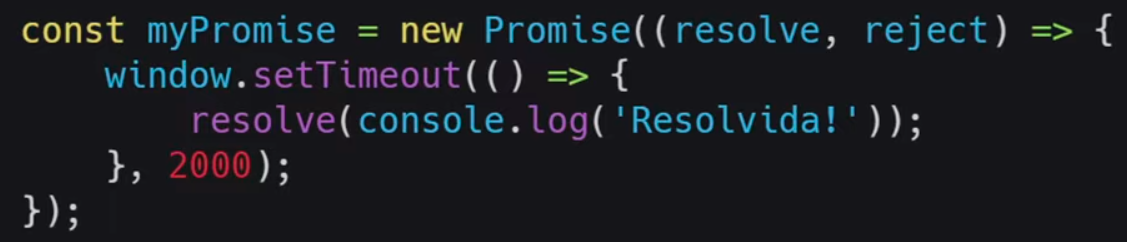
Uma promise pode ter 3 estados:

1. Pending (pendente)
2. Fulfilled (completo)
3. Rejected (rejeitado)

Quando recebemos um objeto **promise** ele sempre chega no estado pendente, depois de algum tempo ela pode ser rejeitada ou completada.

****

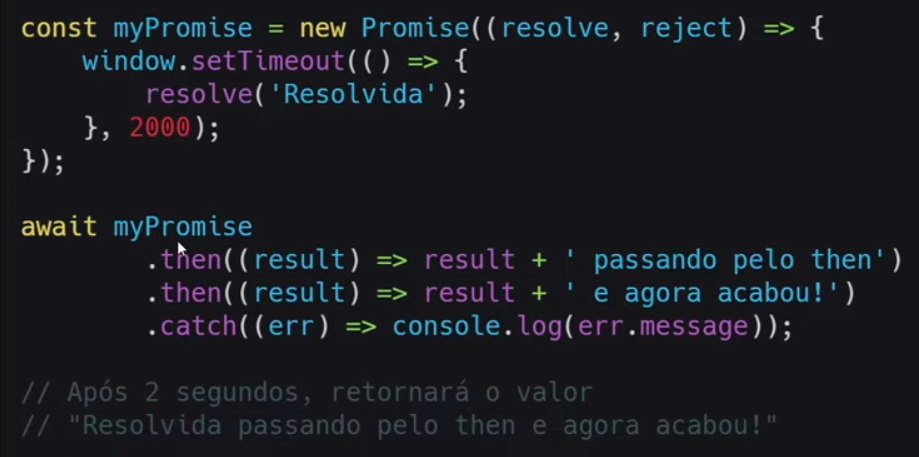
Abaixo temos uma estrutura de uma **promise:**

****

Ela tem os parâmetros **resolve** e **reject** dentro dela (que são seus call backs).

No código acima estamos dizendo que quando receber a promise depois de 2 segundos (2000) assim que resolver minha promise, vai aparecer a mensagem com o log ‘Resolvida!’.

Podemos também relacionar algumas chamadas à promise que eu criei usando **await**, veja na figura abaixo:



O **then** utilizamos para executar algo depois que nossa promise for resolvida. Posso ter mais de um **then.**

Já o **catch** usamos para pegar algum erro caso ele aconteça. No código estou pedindo para dar um console.log com a mensagem desse erro assim que ele acontecer.

**Async/Await (Javascript Assíncrono)**

Funções **assíncronas** precisam das duas palavras chave: **async** e **await**.

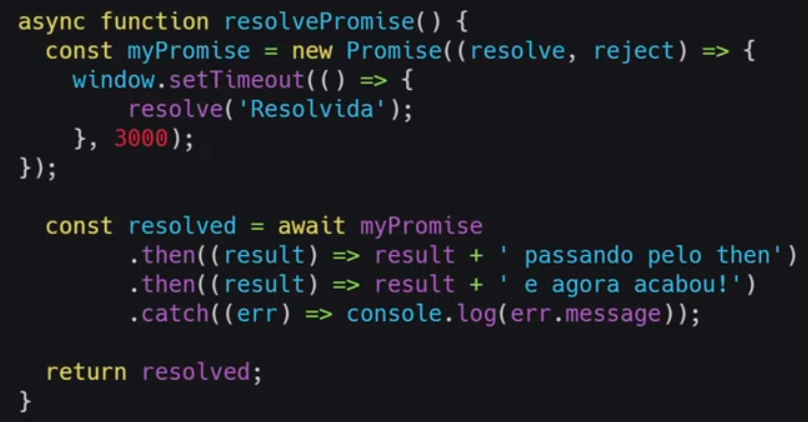
Quando vamos resolver uma **promise**, precisamos dizer que essa função será **assíncrona** para poder usar **await** depois. **Await** é uma palavra reservada feita para lidar especificamente com **promises**.

O await pede pro código fazer uma pausa para aguardar a **promise** ser resolvida. Só depois que estiver resolvida é que o código vai continuar a rodar.

Como o javascript acontece de maneira assíncrona, se não colocarmos o await ele vai perceber a promise, porém continuará rodando outras coisas do código, enquanto a promise não é resolvida. Então, sempre que eu precisar do resultado da minha promise, preciso chamar o **await**.

No código abaixo estou definindo uma função **assíncrona** que depois de 3 segundos vamos receber a palavra ‘Resolvida’.

Vamos colocar o resultado (concatenando strings da mensagem, passando pelo then e catch, etc) dentro da constante **resolved:**

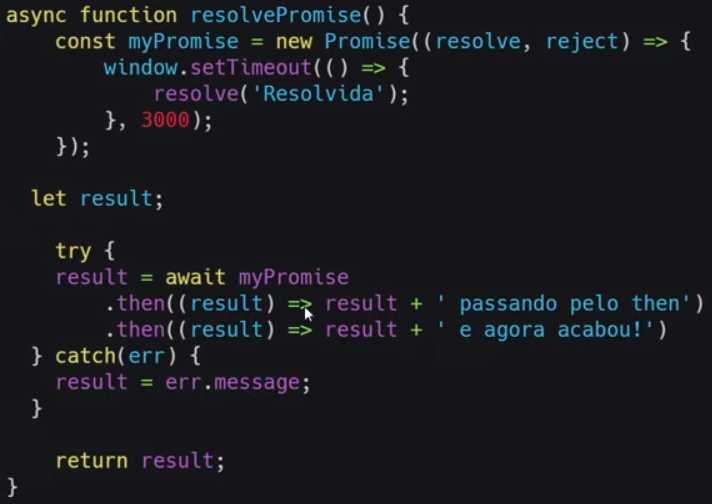
****

Se eu der um **resolvePromise()** (chamando minha função assíncrona diretamente) ele retornará como pendente, pois ele não aguardou a **promise** ser resolvida. Mas quando colocamos **await** ele retorna os resultados com a promise resolvida (ver figura à direita do console):



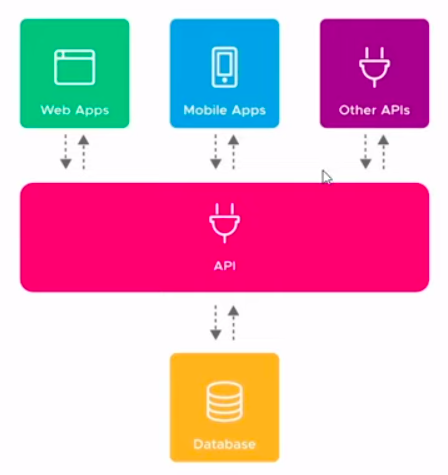
O resultado de uma promise é também uma promise!

Podemos também utilizar o **try… catch** com funções assíncronas:



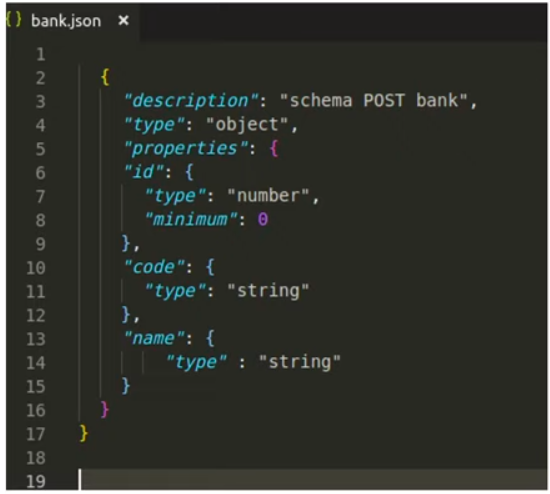
**Consumindo APIs (javascript assíncrono)**

API - Application Programming Interface é uma forma de intermediar os resultados do back-end com o que é apresentado no front-end. Elas são acessadas através de URLs.



Na figura acima temos um banco de dados que passa suas informações para aplicativos web e mobile (e também para outra API) por meio de uma API, e esta troca de informação, ou seja, o acesso à API é feito por meio de uma URL.

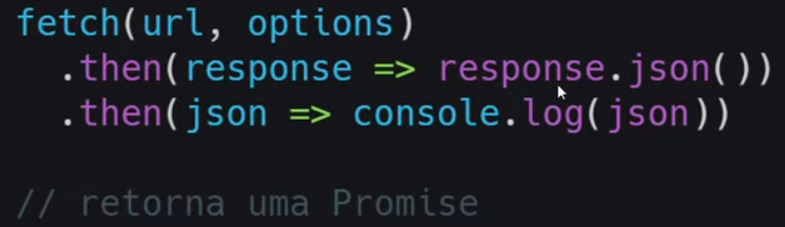
É muito comum que as APIs retornem (e também que enviem) seus dados no formato .json (JSON = JavaScript Object Notation, ou seja, Notação de Objeto em JavaScript) e portanto precisamos tratar esses dados quando recebemos eles.

****

Percebam que o conteúdo de um arquivo .json é um objeto.

**Fetch** é um método queretorna uma **promise**, então, precisamos utilizar o **await** quando usarmos ele!

Sua sintaxe é a seguinte:



Sempre precisamos passar a resposta da promise para o formato .json, só assim é possível manipular isso dentro do javascript.

Exemplo de fetch com operações do banco de dados (POST, GET, PUT, DELETE):



Na figura da esquerda temos um exemplo com GET, já na direita temos com POST que usamos para mandar dados para serem gravados no banco de dados, e pra isso, precisamos converter o conteúdo (body) de JSON para string (usando **.stringify(data)**), pois sempre precisamos tratar o valor que mandamos e recebemos de uma API.

A grosso modo, podemos dizer que fetch e API são URLs que vão possibilitar a comunicação com o banco de dados.

**Atividade Prática**

Criando catAPI:

Nesta atividade, vamos criar uma página que carrega fotos aleatórias de gatinhos sempre que clicamos em um botão.

1. Utilize a API https://thatcopy.pw/catapi/rest para fazer as chamadas com o método fetch();

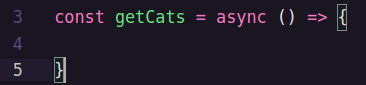
Para armazenar o endereço da API geralmente nomeamos uma constante chamada BASE\_URL:



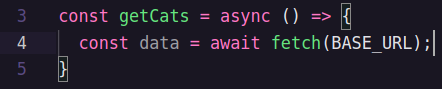
Aqui vamos utilizar somente uma API, então precisaremos somente de um endereço. Mas às vezes utilizamos várias APIs no nosso código e fica mais fácil manipular utilizando as constantes ao invés de digitar a URL sempre e também podemos identificar as APIs melhor!

Observar sempre se a URL está completa! Senão não funciona! Neste caso ela precisa da última barra depois de “*rest”* senão não irá funcionar!

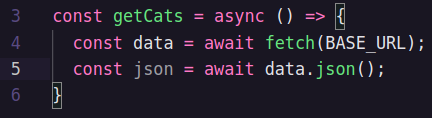
Agora vamos criar nossa função para carregar as imagens dos gatinhos, vou chama-la de getCats. Ela precisa ser uma função assíncrona:



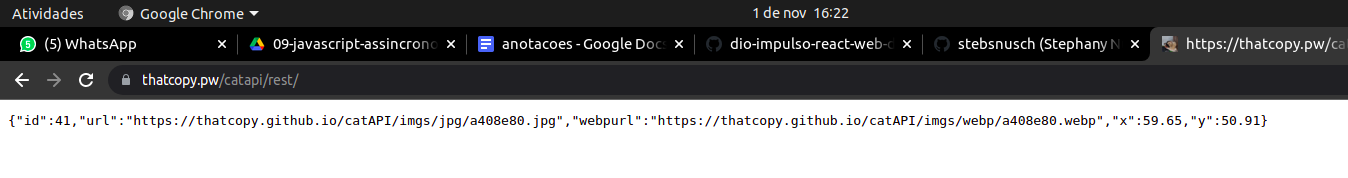
Vou criar a constante data que vai receber os dados, preciso colocar **await** nela antes de fazer o fetch da BASE\_URL:



Com isso, a nossa BASE\_URL vai retornar outra promise, então vamos precisar pegar essa informação que vamos receber, que será em formato de string, e passar pra JSON:

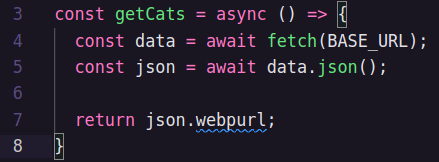


Se a gente abrir o URL da API (que colocamos como nossa BASE\_URL), o browser abre a seguinte informação:

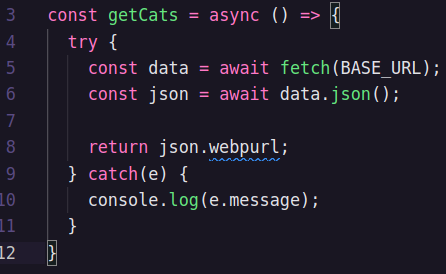


{"id":41,"url":"https://thatcopy.github.io/catAPI/imgs/jpg/a408e80.jpg","webpurl":"https://thatcopy.github.io/catAPI/imgs/webp/a408e80.webp","x":59.65,"y":50.91}

Temos o ID da foto, uma URL e a webpurl que é uma imagem comprimida, o jpeg da url é muito demorado para carregar, por isso vamos utilizar webpurl:

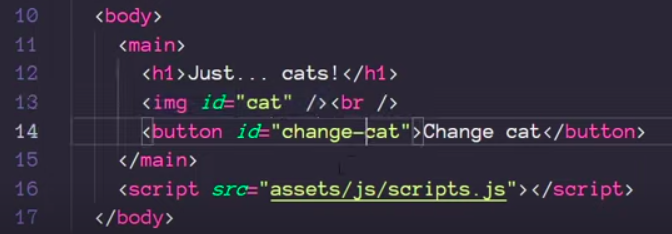


Podemos colocar tudo isso dentro de um try catch. Vamos testar tudo isso que criamos, e caso dê um erro vamos pegar ele usando o catch, retornando a mensagem desse erro:

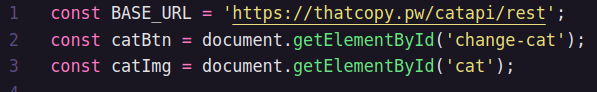


1. Utilize seus conhecimentos na manipulação do DOM para criar a imagem e ativar o evento de clique do botão!

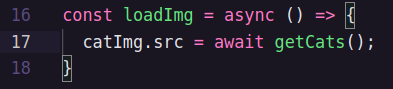
Agora precisamos ver se deu certo, vamos fazer isso interagindo com a imagem. No HTML temos algumas id de onde vamos receber a imagem e do botão onde faremos a mudança das imagens, ou seja, precisaremos selecionar esses dois elementos do DOM (**id cat** e **id** **change-cat**):



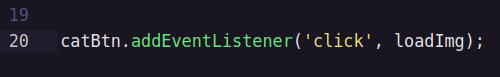
Vamos criar uma constante para manipular esses elementos DOM:



Agora vou criar uma constante para carregar a imagem e receber uma função também assíncrona e dar um await no getCats para dar tempo de carregar a URL da imagem que ele vai receber da API. Vamos fazer isso já jogando a URL que vamos receber para dentro da propriedade **src** da imagem, ou seja, preciso dar valor pro source no html, senão não vai ter nada pra apresentar:



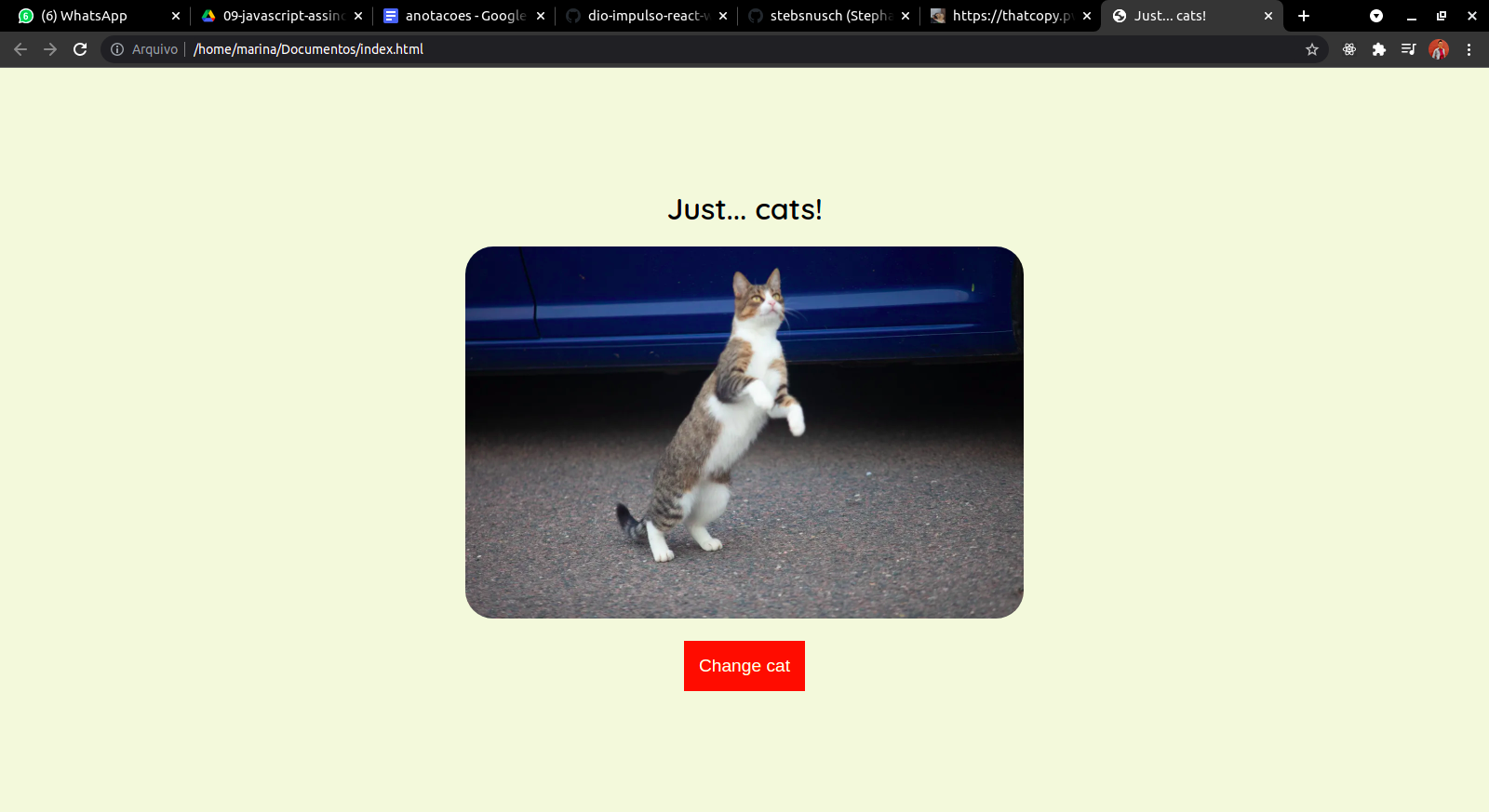
Agora precisamos colocar um listener para alterar as imagens quando clicarmos no botão, ou seja, ele vai chamar o **loadImg** para fazer o carregamento de novas URLs e mudar as imagens:



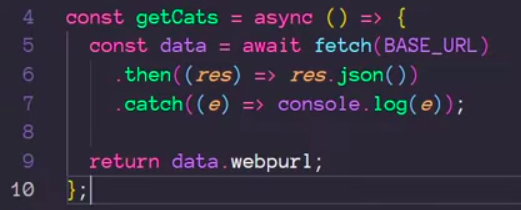
E para a página já inicializar com uma imagem carregada, já vamos chamar o **loadImg** no final do nosso código:



Pronto:



Poderíamos fazer a chamada sem o try catch também:



Enfim, são várias sintaxes para fazer isso!