[00:00:00] Olá! **Boas-vindas** ao treinamento sobre **consumo de API e tratamento de dados que retornam a partir dela**. O meu nome é **Mônica Hillman**, mas você pode me chamar de Moni Hillman. Eu serei a instrutora que vai te guiar nessa jornada de aprendizado.

[00:00:13] Vamos contextualizar a nossa história. Nós somos uma dupla de desenvolvedores da **Alura**, especificamente do produto AluraBooks, uma plataforma de venda de livros técnicos. Nós trabalhamos com a linguagem JavaScript.

[00:00:26] Durante nosso desenvolvimento, encontramos um problema nos dados dos usuários do AluraBooks. Vários usuários moravam na mesma rua e cada um deles estava escrevendo o nome dessa rua de formas diferentes. Isso impediu que os nossos cientistas de dados fizessem uma análise mais aprofundada sobre os nossos clientes.

[00:00:45] Faremos um brainstorming para construir uma solução, passando por vários termos e conceitos sobre o JavaScript. Os termos são: JavaScript assíncrono e síncrono, Event Loop, Call Stack e Task Queue.

[00:01:01] Esses quatro assuntos específicos vão nos ajudar a entender o funcionamento do JavaScript. A partir disso, aprenderemos sobre Callbacks, Fetch API e Promises e, em seguida, sobre assincronicidade, assunto relacionado ao JavaScript assíncrono e às requisições, que é o consumo da API.

[00:01:23] Esses três tópicos nos gerarão mais dúvidas, especialmente sobre: Then; JSON; tratamento de erros com catch; e Finally, que são métodos das Promises, termo que acabamos de mencionar.

[00:01:38] Também aprenderemos sobre Async Await, sobre tratamento de erros com Async e Promise All. Isso vai nos ajudar a entender funções assíncronas, assim, continuaremos o nosso aprendizado sobre assincronicidade.

[00:01:52] Por fim, vamos aprender um pouquinho sobre getElementById, sobre Value e sobre addEventListener. Usaremos esses três tópicos para manipularmos o DOM e chegar na solução esperada. Repare que passaremos por vários tópicos para chegarmos na solução.

[00:02:11] Mas, afinal, qual vai ser a solução? Começaremos pegando o formulário de cadastro de clientes da AluraBooks. Depois, vamos consumir a API do ViaCEP. Quando o nosso usuário for se cadastrar e colocar o CEP, vai puxar já o nome da rua, o nome da cidade e o estado automaticamente.

[00:02:29] A partir de agora, teremos todos os nomes de rua escritos da mesma maneira, e isso vai facilitar muito a vida dos nossos cientistas de dados. Vamos testar e verificar a solução final.

[00:02:43] Estou com a tela de cadastro do AluraBooks aberta e eu vou até o CEP. Vou colocar um CEP padrão que vai devolver um endereço de São Paulo. Em seguida, vou clicar no campo de "CEP" e apertar “zoom” para vermos melhor.

[00:02:57] Agora, adicionaremos um "01001000" e tiraremos o mouse do campo de CEP. Selecionando fora, ele completa automaticamente o endereço da Praça da Sé, cidade de São Paulo e o estado de São Paulo.

[00:03:11] O próximo passo é enviar esse formulário. Ele vai pedir para eu completar tudo e poderemos testar depois. O ponto principal, é: conseguimos resolver o problema dos nossos cientistas de dados. Agora ele completa automaticamente o endereço, a cidade e o estado.

[00:03:24] Se você quer passar por toda essa trilha de aprendizado comigo e resolver esse problema do AluraBooks, além de outros projetos pessoais seus, te convido a participar do treinamento. Te vejo nos próximos vídeos!

Olá, estudante! Desejamos boas vindas ao curso JavaScript: Consumindo e tratando dados de uma API.

No curso, usaremos o editor de código chamado VSCode. Caso queira acompanhar a instrutora com as mesmas configurações, reserve um tempinho para a instalação do mesmo.

Também desenvolveremos os scripts em um projeto já construído com html e css que é o formulário de cadastro de usuários do alurabooks, para baixá-lo você pode acessar o [repositório do github](https://github.com/alura-cursos/js-consumindo-dados-api/tree/main) ou [clicar aqui](https://github.com/alura-cursos/js-consumindo-dados-api/archive/refs/heads/main.zip).

Caso tenha dúvidas durante a instalação ou no decorrer do curso, pode contar conosco criando um tópico no fórum ou interagindo no nosso servidor do Discord. Também não deixe de ajudar outros colegas. Vamos construir juntos essa grande comunidade da Alura? :)

[00:00:00] Olá, estudante! Que bom ver você mais uma vez nos cursos da **Alura**. Dessa vez, nós seremos uma dupla de desenvolvedores da própria Alura e trabalharemos no projeto da AluraBooks, mais especificamente no cadastro de novos clientes da AluraBooks, que é uma empresa de livros.

[00:00:19] Trabalharemos preferencialmente com JavaScript. Para conseguirmos fazer um código ou uma programação em JavaScript, precisamos de um arquivo dessa linguagem.

[00:00:31] Sendo assim, você precisa baixar o arquivo disponível no "Configurando Seu Ambiente", na atividade anterior a esse vídeo. Isso também garante que construiremos o mesmo projeto. Tendo ele baixado, descompactaremos a pasta e abriremos no Visual Studio Code ela.

[00:00:50] Então, eu vou em "Arquivo" no canto superior esquerdo, "Abrir pasta" e vou procurá-la no meu computador. Eu coloquei em "Meus Documentos > dev > Cadastro AluraBooks". Após selecionarmos a pasta, vamos abri-la e criar o arquivo JavaScript.

[00:01:05] À esquerda, localizaremos a opção "EXPLORER" ou "EXPLORADOR". Depende da língua que está o seu Visual Studio Code. Nele, selecionaremos "Novo Arquivo" ou "New File". Você vai criar o arquivo script.js. Dentro dele, apertaremos console.log("Hello World") e salvaremos.

[00:01:05] O processo é diferente do CSS. Antes tínhamos feito um arquivo styles.css com todos os nossos estilos. Para criarmos um arquivo JavaScript, colocamos o final .js. Essa é diferença básica entre cada um deles. Se você fizer de outra linguagem, provavelmente esse final também vai ser diferente.

[00:01:50] Como testaremos o console.log que acabamos de escrever? Vou abrir esse script no meu computador, indicando os programas que eu desejo abrir, mas pode ser que dê um erro, porque esses arquivos JavaScript são executáveis.

[00:02:05] Então, dependendo do caso, pode ser que você tente abrir e ele rode no seu computador. Mas isso não funciona, porque ele não vai entender o que é console, nem o que é log, não vai entender nada.

[00:02:16] Para resolver esse problema, vamos importar o script no arquivo index.html. No final do index.html, antes de fechar o body, você vai abrir a tag script src="script.js" e vai salvar.

[00:02:33] Olha só que diferente: antes, quando importávamos outros tipos de arquivo, como externa, fonte ou estilos do CSS, íamos no head, no início do HTML, e colocávamos a tag link rel. Depois, definíamos a relação daquele arquivo, que era o stylesheet, e href, que é o link do arquivo, seja o externo, que é o fonts.google, seja até o styles.css.

[00:03:03] Vamos abrir no navegador esse script para ver se funcionou mesmo. Precisamos abrir o index.html, que é onde importaremos e acessaremos "Inspecionar".

[00:03:17] Você pode abrir isso tanto com "F12", quanto clicando com o botão direito do mouse, que vai ter a opção "Inspecionar". Lá dentro, ele vai abrir as "Ferramentas do Desenvolvedor". Tem várias opções, mas para a nossa, vai interessar o Console.

[00:03:30] Clicando no Console, vai parecer o "Hello World", então a nossa importação está funcionando com sucesso. Já configuramos a nossa ferramenta e nela visualizaremos tudo que construirmos com JavaScript, todo o código que fizermos. Todo o resultado vai aparecer nesta "Ferramenta do Desenvolvedor". Então, é ali que vamos testar todas as integrações e todos os nossos aprendizados.

[00:03:50] Então ainda temos muita coisa para aprendermos de JavaScript, facilitando o desenvolvimento desse formulário de cadastro da AluraBooks. No próximo vídeo, estudaremos JavaScript assíncrono e síncrono. Te vejo lá!

[00:00:00] Anteriormente, estávamos discutindo sobre os termos **síncrono** e **assíncrono**, mas onde é que isso funciona na vida real? Quando começou a pandemia, tudo se tornou home office, as aulas se tornaram online e as reuniões de trabalho também eram online.

[00:00:14] Participávamos de muitas videochamadas, seja na aula, em que a professora falava da casa dela e ouvíamos da nossa casa, seja no trabalho, onde o pessoal atualizava seus status, o que eles estavam fazendo, e você falava o seu também, ou até no happy hour com seus amigos.

[00:00:31] Chamamos essas atividades em vídeo de videochamadas, síncronas, ou seja, quando você está falando ao mesmo tempo, simultaneamente, com os seus colegas, seus amigos.

[00:00:41] A comunicação assíncrona também era bem comum, por exemplo, as mensagens de texto, quando você mandava um "Oi" para o seu colega e ele demorava horas para responder. Quem não tem aquele amigo que demora muito tempo para te responder? Essa é a comunicação assíncrona.

[00:00:57] Então, quando você manda uma mensagem e seu amigo que demora três horas para te responder, quais são as opções? Você para de fazer suas coisas, tudo que você tem para fazer, ou você se ocupa para fazer outras tarefas até chegar essa resposta dele?

[00:01:15] Isso é o que acontece em um sistema JavaScript também. Você pode ter o sistema síncrono, que é o padrão dele: responder uma tarefa após a outra. Por exemplo, uma imagem carrega, depois a outra carrega e depois a outra carrega e assim por diante, seguindo um fluxo.

[00:01:31] Também temos como fazer um sistema assíncrono, com tarefas acontecendo, sendo concluídas uma após a outra, mas também com outras em segundo plano ou afastadas para carregar depois.

[00:01:43] Vamos aplicar esse procedimento do JavaScript síncrono, o padrão, considerando a história da conversa do seu amigo que demora 10 anos para responder. Para isso, abriremos novamente o Visual Studio Code e o script.js.

[00:01:59] Agora precisamos criar outro console.log que represente essa conversa. Primeiro vamos colocar ("Mandando oi pro amigo!"). Em seguida, criaremos uma função chamada mandaMensagem.

[00:02:17] Prosseguindo, escreveremos function mandaMensagem() {}, usando os "bigodinhos" que bastante gente usa. Vamos colocar mais console.log ali dentro para simular outros elementos dessa conversa. Então, mandaremos ("Tudo bem?").

[000:02:35] Faremos outro console.log("Vou te mandar uma solicitação!"). Agora vou colocar pontos e vírgulas. Por fim, outro console.log. Estou colocando aqui uma conversa fictícia. ("Solicitação recebida!") e fechando essa função mandaMensagem.

[00:02:57] Para chamar essa função, escreveremos o mandaMensagem e adicionaremos outro console.log que vai acontecer depois da função. Vai ser ("Tchau tchau!"), só para dizer que acabou a conversa. Vamos abrir no navegador e verificar o resultado.

[00:03:14] Então, eu vou abrir a pasta que está o projeto, "Documentos > dev > Cadastro AluraBooks", o index.html e a "Ferramenta do Desenvolvedor" novamente. Usaremos "F12" ou botão direito e "Inspecionar". Por fim, acessaremos "Console".

[00:03:31] Tudo que escrevi aparece. Vou colocar o código do lado do Console para comparamos com o primeiro Console, o "Mandando oi pro amigo!".

[00:03:44] Ele começou, depois foi para outra linha, que é o mandaMensagem. Não é a função. A função só acontece quando ela é chamada. Então, na linha nove, chamamos um mandaMensagem que rodou os três console.log que estão na função e depois foi para o "Tchau tchau!". Ou seja, está acontecendo uma coisa por vez.

[00:04:01] Imagina se essa conversa fosse muito mais longa, com console.log maiores, quanto tempo perderíamos nela. Não seria muito melhor fazer ela acontecer em segundo plano, para realizarmos nossas tarefas enquanto ela acontece?

[00:04:17] Essa é a base do assíncrono que aprenderemos durante o curso. Entenderemos como funciona essa leitura - tanto de códigos síncronos, quanto assíncronos - no próximo vídeo. Até lá!

[00:00:00] Anteriormente, discutimos como funciona a comunicação síncrona, a comunicação assíncrona e ainda colocamos em código uma simulação de um chat entre colegas e amigos.

[00:00:10] Quando fizemos isso, eu comentei que, por padrão, o JavaScript é um sistema síncrono. Mas como funciona essa leitura de ação por ação, uma após a outra, por trás dos panos no JavaScript? Vamos entender isso agora.

[00:00:25] Basicamente, podemos separar em três partes o que acontece no nosso código: o Event Loop, o Call Stack e o Task Queue. Vamos analisar o código que fizemos antes.

[00:00:40] Quando chamamos o primeiro console.log, ele vai ser enviado lá para o Call Stack, “Essa ação foi enviada para o Call Stack”. Em seguida, aparecerá uma chamada da função. Conforme havia comentado, a criação da função fica ali, mas ela só é chamada quando chamamos no código.

[00:00:58] Então, o mandaMensagem() é a chamada e ele fica no Call Stack até terminar o que precisa fazer: imprimir três console.log.

[00:01:10] Então, cada um deles vai ser chamado, terminar de acontecer e sair. O próximo console.log será chamado, impresso na tela e depois vai sair. Por fim, o terceiro console.log vai aparecer e sair.

[00:01:25] Assim, a chamada da função termina sua atuação ali, o que tinha para fazer, e vai sair do Call Stack. Só após terminar tudo que tinha para fazer, vai sair da Call Stack. Então, será chamado o último console.log, o "Tchau tchau!". Imprimiu na tela, sumiu. Basicamente é isso que acontece no sistema síncrono.

[00:01:48] Agora sabemos mais a fundo o que é cada parte. O **Event Loop** é como um segurança na festa e vai guiando para qual porta a pessoa tem que entrar, qual é o acesso dela e quando ela deve acontecer. Então ele fica o tempo inteiro no código olhando o que vai ser chamado para a Call Stack.

[00:02:06] Na **Call Stack** vão todas as coisas que devem acontecer na tela. Então ele chama uma coisa por vez do código, essa é a função dele. O **Event Loop** manda para lá o nosso código, as partes VIPs que têm que acontecer primeiro. Elas entram primeiro ali na fila.

[00:02:24] Por padrão, tudo vem vazio. Mas também temos a **Task Queue**, que vai dar conta do nosso sistema síncrono. É a outra fila que o guarda vai fazer a segurança. No próximo vídeo, estudaremos melhor esse assunto e entenderemos também visualmente como funciona essa parte. Até lá!

**Leitura síncrona**

function mandarMensagem() {

console.log(“Estou aprendendo a programar.”);

}

console.log(“O javascript é legal.”);

mandarMensagem();

console.log(“Eu gosto de HTML e CSS.”);COPIAR CÓDIGO

Com o event loop reproduzindo esse código através do empilhamento de ações na call stack, qual será a ordem que poderemos visualizar essas frases no console?

O javascript é legal.

Estou aprendendo a programar.

Eu gosto de HTML e CSS.

É isso aí! O código será lido linha por linha e o console.log de dentro da função será impresso quando a função for chamada, ou seja, entre a frase “O javascript é legal” e a “Eu gosto de HTML e CSS”.

[00:00:00] No último vídeo, estávamos testando o código que simulamos em JavaScript, da conversa entre amigos e colegas, só que eles eram síncronos e a leitura funcionou através do *Event Loop* e do *Call Stack*.

[00:00:13] No sistema do JavaScript, havia uma parte de *Task Queue* que não vimos funcionar e agora a colocaremos em prática. Naquele código que fizemos antes, a função mandaMensagem() agora vai começar com o setTimeOut, depois abriremos parênteses, faremo a chamada da função com mandaMensagem, 5000) e fecharemos parênteses.

[00:00:37] Agora testaremos o funcionamento no nosso sistema do *Event Loop*. Ele vai chamar a primeira linha, o primeiro console.log e colocar na *Call Stack*. Então, imprimiu na tela, chegou a hora dele sair de lá.

[00:00:52] Agora o *Event Loop* vai chamar o setTimeOut, que funciona diferente. Ele é um setTimeOut de cinco segundos, portanto, não vai acontecer agora, será enviado para o *Task Queue*. O *Event Loop* vai colocá-lo em outra fila e continuará com o código funcionando da mesma maneira.

[00:01:10] Agora, em vez de chamar a função, ele chamará o último console.log. Então, o console.log do "Tchau tchau!" é o que acontece agora. Quando ele terminar, vai esperar passar cinco segundos que dissemos para ele esperar e depois chamará a função mandaMensagem para o *Call Stack*.

[00:01:30] Agora ele vai fazer o comportamento normal: chamar o primeiro console.log, que será empilhado em cima da função; imprimir na tela; e sair. Vamos colocar o segundo console.log que está dentro da função. Aconteceu, foi embora. Colocou o terceiro, também aconteceu e foi embora.

[00:01:48] Após terminar de rodar a função inteira, ela vai sair do *Call Stack*. Quando a função terminar, ela também vai sair do *Call Stack*. Rodou o nosso código inteiro e já é possível notar a diferença.

[00:01:59]. Acrescentamos a função setTimeOut do JavaScript que faz acontecer algo depois de um certo tempo que definirmos. Então, ele calcula em milissegundos. Eu coloquei 5000 para ser 5 segundos.

[00:02:16] Podemos deixar algo no canto fazendo a sua função para depois ser chamado. Isso também acontece no cotidiano.

[00:02:27] Imagina que você está lavando a louça e faz a divisão de quais itens serão lavados antes. Eu faço assim: lavo primeiro os meus copos. Se a panela está muito suja, eu coloco uma água e deixo por algum tempo de molho para ajudar na lavagem.

[00:02:43] Enquanto isso, eu não paro toda a minha louça para esperar a panela, vou lavar os pratos, os talheres e tudo o que estiver na pia. Depois eu pego a panela e lavar de novo.

[00:02:56] Basicamente, funciona isso. Você deixa ali no canto para ele se preparar e depois puxa de volta para rodar na sua tela. Essa função do mandaMensagem está sendo enviada como um parâmetro do setTimeOut. Isso é chamado de *Call Back*. No próximo vídeo, estudaremos melhor esse assunto. Até lá!

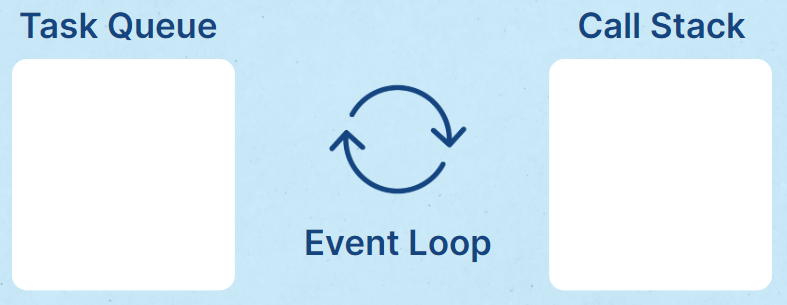
**Para saber mais: como o JavaScript funciona?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/javascript-consumindo-tratando-dados-api/task/109363/next)

No cotidiano da pessoa desenvolvedora front-end suas tarefas envolvem desenvolver vários tipos de funções: as que acontecem a partir de alguma ação do usuário, as que carregam dados externos (geralmente recebidos através de uma API), entre outras.

**Por padrão o JavaScript funciona de modo síncrono**, executando as tarefas linha a linha. Funções como as descritas anteriormente podem atrasar essa execução por terem um tempo de espera relativo a fatores externos (como o usuário ou a API). Para isso não afetar o nosso projeto, temos o que é chamado de **programação assíncrona**.

A assincronicidade em programação é o ato de executar uma tarefa em “segundo plano”, para a execução das outras tarefas menores acontecerem enquanto a maior está sendo carregada, sem interromper o código. O javascript tem o comportamento de executar uma coisa por vez, mesmo se transformarmos trechos de código em assíncrono. Mas como funciona isso?



Podemos destrinchar o fluxo de execução de tarefas em JavaScript em três partes: Event Loop, Call Stack e Task Queue. O **Event Loop** é um ciclo que monitora e executa as ações que mandamos para o JavaScript. O processo de leitura do código só é finalizado quando não existem mais ações a serem executadas. A **call stack** é um mecanismo que organiza como irá funcionar o script quando existem muitas funções: qual função está sendo executada, quais estão sendo chamadas dentro de alguma função, etc. Por fim, a **task queue** é a fila de tarefas assíncronas. Se algo precisa ocorrer em segundo plano ou mais tarde, é nessa fila que ele será adicionado e executado mais tarde.

Você pode descobrir mais sobre a linguagem de programação JavaScript no Alura+ [“O que é JavaScript”](https://cursos.alura.com.br/extra/alura-mais/o-que-e-javascript--c1311) onde o instrutor Júlio César nos mostra a história da linguagem até os dias de hoje e para que ela serve. O artigo [“Hoisting no JavaScript”](https://www.alura.com.br/artigos/hoisting-no-javascript) do instrutor Luan Alves também ajudará no esclarecimento de algumas particularidades na execução do seu código.

## Nessa aula, você aprendeu como:

* Importar um arquivo javascript dentro de um arquivo com formato html;
* Diferenciar um código javascript assíncrono;
* Compreender a leitura do código javascript;
* Diferenciar os termos event loop, call stack e task queue.

|  |  |
| --- | --- |
|  | /\* |
|  |  |
|  | --- Código da conversa versão síncrono --- |
|  |  |
|  | console.log("Mandando oi pro amigo!"); |
|  |  |
|  | function mandaMensagem() { |
|  | console.log("Tudo bem?"); |
|  | console.log("Vou te mandar uma solicitação!"); |
|  | console.log("Solicitação recebida!"); |
|  | } |
|  |  |
|  | mandaMensagem(); |
|  |  |
|  | console.log("Tchau tchau!"); |
|  |  |
|  | \*/ |
|  |  |
|  | console.log("Mandando oi pro amigo!"); |
|  |  |
|  | function mandaMensagem() { |
|  | console.log("Tudo bem?"); |
|  | console.log("Vou te mandar uma solicitação!"); |
|  | console.log("Solicitação recebida!"); |
|  | } |
|  |  |
|  | setTimeout(mandaMensagem, 5000); |
|  |  |
|  | console.log("Tchau tchau!"); |

[00:00:00] Hoje eu estava com muita, mas muita fome, e eu peguei lá no meu congelador uma lasanha de micro-ondas. Seguindo as instruções da embalagem, coloquei por cinco minutos no micro-ondas e depois comi.

[00:00:11] Você pode estar estranhando, pensando que isso não tem nada a ver com o que está aprendendo, mas eu te prometo que existe uma motivação. Existem ações que geram outras ações, por exemplo: eu coloquei no micro-ondas para gerar o meu almoço. Elas são chamadas de *Callbacks*, e aparecem no JavaScript direto.

[00:00:31] Recordando o nosso código, há algum *Callback* naquela simulação da conversa entre amigos em código? Sim, quando enviamos a função mandaMensagem dentro da setTimeout. Ou seja, *Callbacks* são, basicamente, funções enviadas como parâmetro para outras funções.

[00:00:52] Essas funções acontecem, geralmente, depois de um tempo que o setTimeout faz o envio ou de acordo com alguma interação do usuário com a tela, como um clique ou uma seleção de mouse. Existem várias situações que podem ser *Callbacks* e as mais usadas são as requisições.

[00:01:08] Uma maneira de exemplificar e justificar o uso de *Callbacks* no assíncrono é literalmente o setTimeout que fizemos agora. Ele transformou a execução da função mandaMensagem em cinco segundos após o início do código.

[00:01:24] Então, aconteceu uma tarefa em segundo plano, que não interrompeu o andamento das funções posteriores, dos outros console.log. Isso facilita e agiliza muito o nosso trabalho, por exemplo, com as requisições.

[00:01:41] Se eu estou tentando mostrar imagens da tela que vêm através de outros sites e elas demoram demais, eu não preciso interromper o meu site. Ela vai ficar lá no canto, carregando. Falando nas requisições, no próximo vídeo vamos aprender a fazer a nossa primeira. Até lá!



[00:00:00] No último vídeo, eu citei para você que o caso mais comum de uso de *Call Backs* e de operações assíncronas são as requisições. E nós, da equipe de desenvolvedores da Alura, nos deparamos com um problema nos dados dos usuários que se cadastraram no AluraBooks.

[0:00:19] Esse formulário que estamos construindo já existia, só que ele era um pouco diferente. Verificamos o banco de dados para sabermos como os usuários estavam cadastrados e encontramos vários problemas.

[00:00:32] Por exemplo, um cliente que morava na Rua Getúlio Vargas escreveu o nome inteiro, completo, e o número 1520. No mesmo campo da rua, ele colocou até o número. O Cliente 2 colocou "Rua" abreviada, "Getúlio" abreviado e "Vargas", e o número 1520. O outro colocou "R. Getúlio Vargas", e o outro, só "G. Vargas".

[00:00:55] Quatro clientes moram no mesmo endereço, talvez até no mesmo prédio, e cada um deles escreveu o mesmo endereço de jeitos diferentes. Para normalizar isso, nós pensamos e desenvolvemos um *brainstorming*.

[00:01:08] Faremos a solução consumindo a *API* do ViaCEP. Eu vou entrar no site para mostrar a vocês como funciona. Então, ao entrar aqui, há um tutorial de como acessar o *webservice* de CEP, a validação do CEP, uma explicação dos possíveis erros e os formatos de retorno.

[00:01:27] Usaremos a URL do viacep.com.br/ws/01001000/json para consumir essa *API*. Basicamente, aquele número com 01001000, vários zeros, é um CEP que eles usaram como padrão. É basicamente ali que vamos colocar o CEP do cliente para trazer o endereço de volta. Foi essa a solução do *brainstorming*.

[00:01:55] Mas, antes de tudo, o que é uma *API*? O que é isso que vamos consumir agora? O significado de **API** é Interface de Programação de Aplicações. Ela permite que dois componentes de software se comuniquem. Chamamos esses dois lados de cliente e servidor.

[00:02:12] Então, a *API* fica no meio fazendo a conexão. O cliente faz uma solicitação para essa *API*, ela faz os trâmites e pede ao servidor para retornar a resposta.

[00:02:25] Assim está bem técnico e difícil de entender, então vamos imaginar que você queira pagar um boleto no aplicativo do banco. Quando você acessa o aplicativo no seu celular, você preenche os campos de usuário, conta, dados do banco e a senha.

[00:02:41] No momento que você, cliente, apertou o botão "Entrar", ele está enviando uma solicitação para o servidor. No meio deste processo está a *API* formalizando a sua requisição para o servidor entender.

[00:02:53] E o servidor vai retornar, através da *API*, deixando mais organizado, legível, seu saldo, número da conta e nome. Ele vai permitir várias interações: pagar boleto, ver o saldo, transferir, fazer um PIX. Várias coisas.

[00:03:14] Enfim, no meio de todas essas interações, existe a *API*, aquele link viacep.com.br/ws/ que vamos pegar para conseguir acessar o servidor deles. Mas como fazer essa interação? Através do *Fetch API*.

[00:03:26] Outra vez no ViaCEP, copiaremos o link viacep.com.br/ws/01001000/json e voltaremos ao código. Você pode tirar totalmente aquela simulação de conversa de chat que tínhamos feito, porque não usaremos mais.

[00:03:40] Primeiro, vamos criar uma variável com o nome de consultaCEP. Vamos adicionar consultaCEP = fetch ('viacep.com.br/ws/01001000/json/').

[00:04:03] Já estamos gerando uma requisição. Vou fazer console.log(consultaCEP). Agora ele vai imprimir o resultado desse *Fetch* no navegador. Vou salvar e abrir.

[00:04:17] Para isso, vamos à pasta do nosso projeto, "Documentos > dev > Cadastro AluraBooks". Em seguida, abriremos o Index. Com ele já aberto, vamos à Ferramenta do Desenvolvedor, que é o botão direito, "Inspecionar" ou "F12". No *Console*, verificaremos que ele retornou uma *Promise*.

[00:04:37] O método *Fetch* é assíncrono e tem como parâmetro obrigatório a URL da *API*. Ou seja, usamos o ViaCEP, usando outra opção, obteríamos outro link, e é esse link que é obrigatório, o único parâmetro obrigatório desse método.

[00:04:54] Colocamos esse valor dentro de uma variável para conseguirmos consultar o retorno - consultamos o valor através do console.log. E o que ele retornou para nós? Uma *Promise*, isto é, uma promessa.

[00:05:07] E o que é isso? Por que não retornou alguma frase ou algum conteúdo, retornou uma promessa? É o que descobriremos no próximo vídeo. Fica aí a curiosidade! Te vejo lá!

[00:00:00] Fizemos a requisição com o *Fetch API* para a *API* do viaCEP e tivemos uma *Promise* como retorno. Mas o que é uma *Promise*? A **Promise** é uma promessa de que algo vai acontecer. Como retorno, ela pode ser resolvida ou rejeitada.

[00:00:19] Isso permite que métodos assíncronos se tornem síncronos. Ou seja, ao invés de retornar um valor específico, o valor final, como ainda não chegou lá, ele retorna uma promessa que esse valor uma hora vai chegar. Podemos comparar essa situação, essas promessas, com uma encomenda ou com algo que gostamos muito de fazer: comprar na internet.

[00:00:43] Quando o nosso pedido sai para a entrega, ele gera uma *Promise*, isto é, uma promessa de que o pedido será entregue ou pode acontecer algum problema e o entregador não conseguir chegar até a sua casa para entregar esse produto.

[00:01:00] Repare como uma promessa pode ser comparada com situações do nosso cotidiano. Falando sobre a anatomia da *Promise* para entendermos corretamente seu funcionamento, ela é composta de uma função com os parâmetros *Resolve* e *Reject*.

[00:01:18] Se essa promessa foi resolvida, ela vai ser chamada de função *Resolve* e enviará uma mensagem ou fará alguma ação que você definiu que fosse acontecer.

[00:01:30] Na *Promise* de entrega, quando acontece a entrega da encomenda, mandamos uma mensagem avisando que a pessoa recebeu a encomenda. Se ela não recebeu, chamamos a função de *Reject* e enviamos uma mensagem indicando que não foi possível essa entrega.

[00:01:48] Na maioria dos casos, não construímos uma *Promise* do zero. Ela é gerada a partir de algo síncrono que, no caso, é o nosso *Fetch API*. Ele está fazendo uma *Promise* por trás dos panos que foi gerada através da nossa requisição.

[00:02:02] Pode acontecer de a requisição demorar para carregar. Então, ao invés de dar um valor final de erro, ele gera uma promessa, e no futuro teremos o resultado da requisição.

[00:02:15] Uma última curiosidade dentro dessa anatomia da *Promise*: perceba que estamos enviando uma função como parâmetro para ela e aparecem *Callbacks*. E é isso que o *Resolve* e *Reject* são, dois *Callbacks* da função da *Promise*.

[00:02:30] Vamos voltar ao código. O *Console* está mostrando essa *Promise* que recebemos do *Fetch API*. Eu abri a *Promise*. Existem três tipos de categoria.

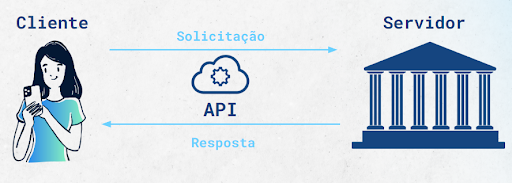
[00:02:44] O **Protótipo** diz que é uma *Promise*, está afirmando que o que tem ali é uma promessa. O estado dessa *Promise* é rejeitada. Em seguida, temos o **Promise Result**, que é o resultado da *Promise*. Ele deu erro, ou seja, caiu no **Reject** da promessa. Esse erro não deveria acontecer. Como fazemos para contornar essa situação? Não é isso que eu quero!

[00:03:03] Então, vamos lá no Visual Studio Code. No *Fetch API*, antes do link viacep.com.br, vamos colocar https://. Salvei, vou voltar no formulário e clicar "F5" só para ele recarregar o *script*. Agora ele mudou, não está mais dando erro nenhum e o estado da *Promise* virou o *Fulfilled*, significa que está completa. Se analisarmos, existem três tipos de estados que podem aparecer.

[00:03:32] O **Fulfilled**, que quer dizer que está completa. O **Reject**, que é o que deu antes e o **Pending**, que quer dizer que ainda não concluiu. Além disso, o PromiseResult também mudou, ele retornou o objeto do tipo Response, com vários elementos dentro: o body, o bodyUsed e headers. Eles não nos dizem respeito agora. Sobre o objeto Response, para o acessarmos, precisamos usar os métodos das *Promises*, que vão retornar outras *Promises*.

[00:04:00] Esses métodos são: o *Then*, o *Catch* e o *Finally*. Eles nos permitirão mostrar na tela todo esse valor do que estamos recebendo, mas, vamos entendê-los melhor na aula que vem. Te vejo lá!

API é uma sigla que significa **Interface de Programação de Aplicações**. Uma API é um mecanismo que permite que duas partes de um software se comuniquem usando um conjunto de definições e protocolos. Sua arquitetura geralmente é explicada em termos de cliente e servidor. A aplicação que envia a solicitação é chamada de cliente e a aplicação que envia a resposta é chamada de servidor.



Quer saber mais sobre esse assunto? Olha só esse Alura+ do instrutor Vinicius Dias onde ele nos responde [“O que é uma API?”](https://cursos.alura.com.br/extra/alura-mais/o-que-e-uma-api--c697).

## Nessa aula, você aprendeu como:

* Criar funções que recebem como parâmetros outras funções (callbacks);
* Identificar uma API;
* Consumir dados de uma API através do método fetch;
* Compreender o funcionamento das promises.
* [00:00:00] Lembra que comentei sobre uma entrega, que estávamos esperando receber, e esse status de entrega era como uma promessa? E quando ela era entregue, essa promessa era resolvida, e quando acontecia alguma coisa com o entregador, a promessa era rejeitada?
* [00:00:17] Então, em qualquer uma dessas situações, a resposta que vai chegar em nós, é um objeto do tipo *Response*. E como fazemos para acessar esse objeto? Vamos no trecho de código, que criamos aquela variável consultaCEP. Então, ao final do *fetch*, você vai colocar: .then ();.
* [00:00:37] Em um panorama mais geral, uma promessa sempre vai retornar um objeto do tipo *response*, seja ela rejeitada ou resolvida. E esse then() funciona assim, como a sua tradução. Ele é basicamente um “então”. Que é: faça o *fetch*, ele vai lá e faz a requisição.
* [00:00:56] E então, com aquela resposta, ele vai fazer alguma operação que vamos colocar dentro do then(). E a resposta, sendo do objeto do *Response*, não vem da maneira que podemos acessar. Vamos precisar converter. Que é da mesma maneira que, como se aquela encomenda que eu estava comentando, ela tivesse vindo com a voltagem 110V. E as tomadas da sua casa são 220V, e você precisa de um adaptador.
* [00:01:24] Ele vai fazer essa conversão de voltagem para você poder usar o seu aparelho. Vamos fazer a mesma coisa que essa resposta. Então, vamos colocar: resposta =>;, porque estamos formando uma *arrow function*. E a resposta, vamos colocar novamente json(). Salvei, e vamos entender, em termos técnicos, porque aconteceu essa conversão.
* [00:01:49] O objeto do tipo *Response* nos trouxe um corpo de resposta que não conseguíamos acessar. Ele trouxe um amontoado de *bytes*. Usamos o JSON para ele converter essa resposta em json, que é um formato muito usado no desenvolvimento em JavaScript, porque ele parece um objeto JavaScript.
* [00:02:07] Então, vamos conseguir acessá-lo. Mas ainda assim, salvando essa alteração, vamos ver o resultado. Salvei e agora eu vou testar o que retorna para nós. Cliquei na tecla “F5”, ainda não exibiu nada. Porque eu só fiz a conversão, eu não pedi para exibir na tela.
* [00:02:22] E seguindo a lógica do *then()*, sendo “então”, vamos colocar outro *then()*, quando fecha o anterior. Então, no final vamos incluir .then(). Eu vou pegar novamente uma resposta ou eu vou colocar “r”, (r =>). É novamente uma *arrow function*: console.log. E vai imprimir esse “r”: console.log(r). Vou salvar e ir no navegador, clicar na tecla “F5”, novamente.
* [00:02:48] Ele primeiro fez a promessa, nos disse que ela estava executando, e depois imprimiu o valor dela. Ainda na linha da variável que pedimos o console.log, e imprimiu todos os dados que precisávamos: o bairro, o CEP e o complemento.
* [00:03:04] Então, eu vou ajeitar o código. Estou com o script.js aberto. Vou clicar na tecla “Enter” depois do .then, o primeiro. E depois clicar na tecla “Enter” no segundo .then. Agora ele ficou apresentável, um embaixo do outro, fica mais fácil visualizarmos. Tudo se tratando da primeira promessa.
* [00:03:21] Com isso, temos uma cadeia de *then()*, porque fizemos o *fetch*, fizemos a requisição. Então, com a requisição, nós convertemos. Depois, esse outro “então”, é referente à promessa de antes, da promessa da conversão.
* [00:03:36] Então vamos sempre fazer algumas coisas que resultem em outras. Eu tenho uma promessa, “então”, ela vai realizar algum comando quando ela for resolvida ou rejeitada.
* [00:03:51] Tem várias coisas que você pode associar, que falando em português e em voz alta, fica fácil de entender. Porque você faz uma associação na sua cabeça com a própria língua portuguesa.
* [00:04:04] Mas, e se retornasse o *Reject*? Ele ainda só retornaria o que nos foi exibido? Como eu faço a impressão desse erro? É legal avisar para o usuário que está com algum problema na requisição que ele solicitou. No próximo vídeo, vamos conhecer o método *catch*, que vai ajudar a fazer essa impressão de erros.

[00:00:00] No vídeo passado, aprendemos sobre o método then(). O método then(), em português, significa "então". O que ele faz? Ele pega a requisição, e aquele valor, ele faz alguma operação com ele. Faz alguma coisa com a resposta retornada. Mas ele só faz se aquela promessa foi resolvida.

[00:00:20] O que acontece se der algum erro e aquela promessa for rejeitada? Por enquanto, não acontece nada. Vamos precisar aprender outro método para imprimir isso na tela. Você vai no Visual Studio Code, e após o último then que colocamos, está na linha três, vamos colocar .catch().

[00:00:35] Vou selecionar a tecla "Enter", antes do ponto, para ele ficar para baixo e ficar mais fácil de visualizarmos. Então, vamos colocar catch(erro => );, que é outra arrow function, console.log(erro): catch(erro => console.log(erro));

[00:00:47] Se eu testar agora, não vai funcionar, não vai acontecer nada. Quer dizer, vai acontecer. Vai dar certo, então não vai cair no catch. Porque o nosso CEP está digitado corretamente.

[00:01:09] Eu vou tirar um zero do link do fetch, porque eu quero que dê problema mesmo. Vou salvar e testar no navegador. Então, eu fui no nosso formulário de cadastro, em que estamos testando nossos scripts e vou clicar na tecla "F5", para recarregar a página. Está demorando um pouco, está pensando. E deu erro.

[00:01:32] Aqueles erros em vermelho são erros do navegador. Então, se você estiver usando o Chrome, o Opera, ou outra coisa, pode ser que apareça o mesmo erro com outra mensagem, porque não é o erro que estamos tratando.

[00:01:44] Aquele último erro escrito em branco, TypeError: failed to fetch, é aquilo que mandamos imprimir no console.log, através do catch. E toda promise retorna esses dois métodos. Ela retorna o then, caso ela for resolvida. E retorna o catch, caso ela seja recusada.

[00:02:07] Você pode interpretar assim: catch em português significa "pegue". Assim, caso aconteça algum erro, ele pega o erro e imprime na tela. Basicamente podemos entender dessa forma para ficar mais fácil de entender como ele está funcionando.

[00:02:22] E essa mensagem é difícil de entender. TypeError: failed to fetch. Imagine sobre a encomenda que eu estava falando até agora. Se a minha encomenda saiu para entrega e não é entregue na minha casa, eu gostaria de uma mensagem que me avisasse o que aconteceu.

[00:02:35] Se não, eu ia pensar: "Meu Deus, entregou para quem? Entregou para um vizinho? Entregou para alguém errado?". Então, para ajudar esse cliente, para dar influência para outros sites fazerem uma mensagem que possamos entender o que está acontecendo, vamos customizar isso.

[00:02:49] Para fazer isso, vamos no Visual Studio Code, na linha três, em que temos o segundo .then. E depois da flecha, do r =>, vamos abrir chaves, e fechar após o console.log(r).

[00:03:06] Vou clicar na tecla "Enter" duas vezes para facilitar a visualização do que eu estou fazendo. Vou colocar if, uma condicional que significa "se". Se (r.erro) for verdade, eu quero que ele faça alguma operação. Coloquei duas chaves de novo, o que estiver ali dentro é um código. E vou colocar throw Error ('').

[00:03:37] Entre os parênteses: ('Esse cep não existe!') E se não for verdade, então else, ele vai dar o console.log(r). O que aconteceu? Eu vou salvar, só que esse erro também não vai dar certo, porque eu quero um CEP que não existe.

[00:04:02] Então eu vou colocar no fetch, em que tem o CEP, eu vou colocar 01001250. Não deve existir, vou salvar e vamos analisar no navegador o que aconteceu. Vou recarregar.

[00:04:18] Agora ele deu esse erro de que o CEP não existe. Por que eu fiz esse tratamento, do (r.erro)? Vamos no viaCEP comigo. Quando ele fala da validação do CEP, no site do viaCEP, ele nos explica como lidar com erros.

[00:04:33] Então, quando consultado um CEP de formato inválido, por exemplo, nove dígitos, alfa numérico, espaço, o código de retorno da consulta é 400. Ele puxa o erro direto. Ele vai dar a promessa rejeitada e vai cair naquele console.log do catch que fizemos.

[00:04:51] Mas se você reparar, nenhum desses casos, é: caso o CEP não exista. No segundo parágrafo, ele explica como funciona, como ele faz esse tratamento de erro.

[00:05:02] Ele explica que, quando consultado um CEP de formato válido, porém inexistente, o retorno encontrará um valor de erro igual a true. Por isso que tratamos desse jeito. Observe o nosso código. Estou com o Visual Studio Code aberto.

[00:05:17] Então, para CEPs que não existem, ele não vai voltar reject, ele não vai no catch, mas no then. E para pegarmos esse erro, temos que fazer essa condicional.

[00:05:31] Se o retorno foi um erro igual a true, não precisamos informar que é true. Eu só coloquei (r.erro), ele vai imprimir um erro chamado de "Esse cep não existe!". E o catch vai pegar aquele erro que eu coloquei. E caso não esteja incorreto, ele vai continuar, vai dar o else console.log(r) e imprimir sem mandar erro nenhum.

[00:05:55] Quando conseguimos outras maneiras de exibir erros, podemos colocar no throw Error. Desta maneira, faz ainda mais sentido usar o catch, porque ele pega esse erro"

[00:06:05] E com os outros condicionais, podemos dar um jeito de imprimir mensagens customizadas direto no catch, sem ser esse caso do "CEP não existe", que ele passa como Resolve. Então, quem sabe você não faz, como desafio, algumas mensagens customizadas a partir do throw Error, ou só mudando no console.log e fazendo uma condicional lidando com o que está vindo errado.

[00:06:25] A partir disso, aprendemos a fazer mensagens customizadas para erros. E quem sabe, vendo a maneira como se faz, com as condicionais, com o throw Error, você não consegue fazer, como desafio, outros erros que podem cair com essas ações de forçar o erro.

[00:06:41] Coloca outro CEP e veja qual o possível erro que pode vir, e tenta tratar para fazer outras mensagens. Aposto que vai ser um desafio para você treinar e vai ser legal!

[00:06:51] Por enquanto é isso, a maneira de tratar é com o catch. Mas ainda tem um método de promises que não vimos, o finally. E vamos ver no próximo vídeo!

[00:00:00] Acabamos de conhecer tanto o método then() quanto o método catch. Cada um deles tem a sua função. O then é quando a nossa promessa é resolvida, e o catch é quando a nossa promessa foi rejeitada.

[00:00:12] Mas também tem mais um método que não conhecemos até agora. É isso que vamos fazer na tela para vermos como funciona, e entendê-lo melhor.

[00:00:21] Para usá-lo, vamos depois do catch, quando fechamos os parênteses, vamos selecionar a tecla "Enter", e colocar finally(). Abre e fecha parênteses, e coloca mensagem => console.log('Processamento concluído!'), entre aspas, porque é uma frase que queremos colocar na tela, precisa desses parênteses.

[00:00:52] Vou salvar e vamos testar o que acontece. Eu vou no navegador, no nosso cadastro, e clicar na tecla "F5". Agora ele vai fazer, vai carregar, esta lento porque minha internet está lenta hoje.

[00:01:09] Ele deu erro porque tínhamos feito aquela alteração no CEP para forçar o erro, e depois ele deu "Processamento concluído!". Vamos tentar com o CEP certo, para ver o que acontece. Eram três zeros, assim, no fetch colocando aquele que eu sei que existe, o CEP "01001000".

[00:01:27] Salvei, vou novamente no navegador, clico na tecla "F5", e agora ele carregou, mostrou a resposta, e informou que o processamento foi concluído. Ou seja, ele apareceu tanto quando deu errado, quanto quando deu certo.

[00:01:40] O finally é literalmente a sua tradução também, sendo "Finalmente". Independente da resposta dessa promessa, ele vai imprimir o que colocarmos.

[00:01:51] E quando realmente não nos importamos em qual vai ser esse resultado, temos uma frase padrão que vai retornar na tela. Um exemplo seria se, lembra da encomenda?

[00:02:01] A encomenda de novo, vamos analisar esse caso. Imagine que a transportadora liga ou manda mensagem, um questionário, independente se a encomenda foi entregue ou se ela não foi entregue. Ela pede esse feedback por procedimento padrão dela.

[00:02:14] Seria o mesmo caso, independente da resposta, aquele finally vai acontecer.

## Arrow Functions

Dê uma olhada na estrutura do nosso código dentro do primeiro método **.then** dessa requisição: .then(resposta => resposta.json()). O conteúdo que está ali presente é uma função, mas construída de uma maneira diferente se torna uma **arrow function**. A versão ES6 do ECMAScript trouxe uma nova forma mais sucinta de trabalhar com funções chamada de Arrow Functions, por causa da sintaxe que lembra uma flecha: **() =>**.

Em uma função tradicional, caso você crie uma variável dentro dela, seu contexto é referente a função onde ela está. Para entender melhor: se você usar a palavra chave “.this”, você está se referindo a essa função em si.

Já em uma arrow function temos um contexto externo. Por exemplo, se essa arrow function for criada dentro de outra função seu contexto será aquela função que ela está dentro. Caso a função for aplicada fora de outra função, seu contexto será global, o código inteiro.

Quer aprender mais sobre esse termo? O nosso parceiro Marco Bruno te ensina em seu vídeo:

## Retornos de requisições

Quando estamos realizando uma requisição para a API, estamos trocando protocolos HTTP’s. [HTTP](https://www.alura.com.br/artigos/desmistificando-o-protocolo-http-parte-1) é um protocolo, uma forma de conversa entre duas máquinas, que permite transferir hiper-texto de um lado a outro. Daí o nome Hyper Text Transport Protocol.

Uma requisição é composta de uma request (solicitação) e uma response (resposta). Request e Response são dois tipos de mensagem diferentes quando falamos de HTTP. A especificação HTTP diz exatamente o que podemos colocar dentro de cada um destes tipos de mensagem para que todos que "falem" o idioma HTTP consigam trocar informações corretamente.

Em uma response é retornado um response code (código de resposta) e um motivo, que dá significado ao código. A estrutura padrão desse código tem três dígitos, sendo o primeiro referente a classificação dele. Elas são:

* 1XX: Informativo – a solicitação foi aceita ou está em andamento;
* 2XX: Confirmação – a solicitação foi concluída ou entendida;
* 3XX: Redirecionamento – faltou alguma coisa na solicitação;
* 4XX: Erro do cliente – houve um erro na solicitação;
* 5XX: Erro no servidor – houve uma falha no servidor durante a solicitação.

Durante essa aula nós conhecemos um deles: quando consultado um CEP de formato inválido na API do ViaCEP ela nos retornará o código 400 (Bad Request).

Caso você queira saber mais sobre os tipos de código de resposta do protocolo HTTP recomendo a aplicação [HTTP Cat](https://http.cat/) que demonstra de forma descontraída as diferentes categorias que podemos encontrar. Para entender melhor sobre o funcionamento das requisições, temos o curso [“HTTP: Entendendo a web por baixo dos panos”](https://cursos.alura.com.br/course/http-fundamentos) do instrutor Fábio Pimentel.

Além de todos esses conhecimentos novos, fizemos com sucesso uma requisição com os métodos das promises .then e .catch. Pra reforçar ainda mais o seu aprendizado, recomendo o artigo [“Começando com fetch no JavaScript”](https://www.alura.com.br/artigos/comecando-com-fetch-no-javascript) e o Alura+ da instrutora Juliana Negreiros onde ela explica brevemente sobre [“JavaScript assíncrono e fetch”](https://cursos.alura.com.br/extra/alura-mais/javascript-assincrono-e-fetch-c93).

## Nessa aula, você aprendeu como:

* Tratar promises a partir dos seus métodos;
* Utilizar o método then para acessar o valor retornado do fetch API;
* Converter dados em JSON através do método json();
* Lidar com possíveis erros com o método catch;
* Enviar respostas independente do retorno com o método finally.

[00:00:00] Não há só uma maneira de fazer um código assíncrono. Do mesmo jeito que não existem só esses casos na realidade de situações assíncronas que eu comentei até agora.

[00:00:10] Vamos imaginar que você queira marcar uma consulta com o dentista na terça-feira. Você vai ligar para a secretária do consultório, ela vai atender, vai dizer que ela vai procurar um horário naquele dia, gerando uma promessa com você. Essa promessa, essa busca por horários, pode ser resolvida achando esse horário disponível e ela pode ser rejeitada, não tendo horário nenhum.

[00:00:32] Mas vamos supor que tem vários horários para terça-feira. Você só declarou querer na terça, não especificou o horário. A secretária ia ter que procurar cada horário e ver se dá para você comparecer. Por exemplo. "Eu tenho horários às 2h. Você pode vir?" Aí você vai responder sim ou não. "Eu tenho horário às 5h. Você pode vir?"

[00:00:51] Imagine que vai ter várias then. E dentro de cada then do nosso código, temos uma Arrow Function. O que tínhamos aprendido no início desse curso sobre funções enviadas como parâmetros para outras funções? Que são os callbacks.

[00:01:08] Quando temos vários "então", temos vários callbacks. E esse termo e essa situação é bem conhecida com o termo de Callback Hell - o Inferno de callbacks. Quando temos vários then() com várias funções. Não é uma condição de fácil leitura.

[00:01:27] Temos outra maneira de fazer essa programação assíncrona, e é isso que vamos transformar agora no código que estávamos fazendo, para ficar mais fácil a leitura.

[00:01:37] Eu quero você abra o Visual Studio Code, e vamos apagar todos aquele then, finally e catch que tínhamos feito. Vamos voltar à estaca zero. Também vou arrumar o CEP, porque eu tinha colocado um CEP errado. Então coloquei "01001000". Salvei.

[00:01:53] Vamos aprender outra maneira, do início, de como fazer um código assíncrono. Quero que você selecione a tecla "Enter" nessa declaração da variável consultaCEP, para criarmos uma função.

[00:02:04] Para criarmos uma função, vamos usar async function buscaEndereco, abre e fecha parênteses, abre e fecha chaves. Você vai fechar as chaves após o var consultaCEP. E você vai usar a palavra await antes do fetch.

[00:02:21] Agora, ao invés de darmos um console.log(consultaCEP), porque agora não queremos chamar a variável. O que queremos é chamar a função. Eu vou chamar a função com buscaEndereco(), abre e fecha chaves, porque ele chama ela e a coloca na Call Stack.

[00:02:39] E eu vou fazer o console.log dentro dessa função. console.log(consultaCEP). Vou salvar, e vamos ver o que acontece. Eu vou ao navegador e selecionar a tecla "F5" na tela de cadastro. Agora eu fiz a solicitação. Note, ele já me mandou até o objeto response. Bem mais rápido do que foi o outro.

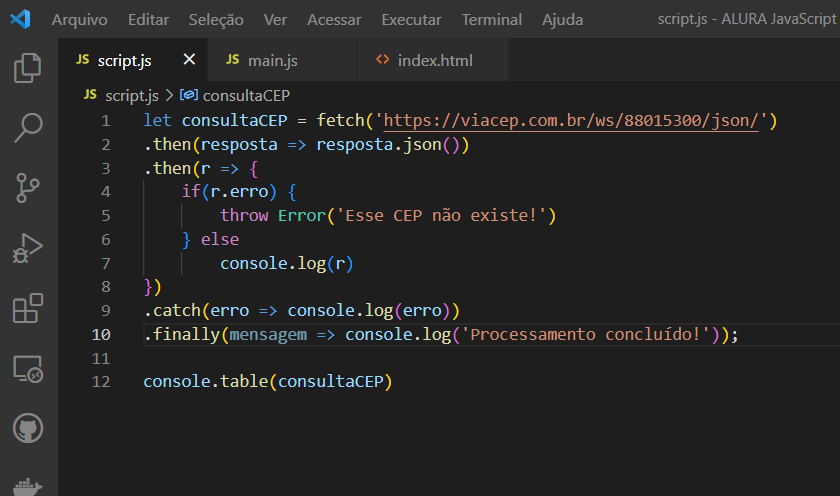
[00:03:03] Essa questão do Async/Await foi declarada pela ES em 2017 para facilitar a leitura dos códigos assíncronos. Porque, apesar de ser assíncrono, ele é construído como um código síncrono. Ou seja, parece que é feito linha por linha, mesmo que, no fundo, ele esteja esperando uma coisa acontecer antes de fazer a outra.

[00:03:21] Apenas definindo a função como async, podemos usar essa palavra await. Ou seja, se você tentar colocar esse await em qualquer lugar que não seja uma função assíncrona, vai dar problema. Ele vai te cobrar que ela só é aceita dentro de uma função assíncrona.

[00:03:35] Agora estamos com o mesmo problema que deu no início. Lembra que eu mencionei sobre o conversor? Precisamos converter esse retorno do fetch em JSON. Eu vou inserir na linha 3, var consultaCEPConvertida = await consultaCEP.json. E vou imprimir isso no lugar, consultaCEPConvertida no console.log.

[00:04:00] Vou salvar, e vamos analisar no navegador o resultado. Vou clicar na tecla "F5" e pronto! Agora ele está retornando o endereço que queríamos com sucesso, em três linhas dentro dessa função. Antes eu tinha feito dois then e usados outras funções. Dessa maneira que executamos agora, parece que é bem mais fácil, não é? Uma coisa, um passo por vez.

[00:04:21] E como eu trato o erro? Porque no then tinha o catch. E como eu faço com Async/Await para resolver isso? É isso que veremos no próximo vídeo. Até lá!



[00:00:00] Lembra da consulta que você estava tentando marcar? Quando você ligou para o consultório e falou com a secretária, ela gerou uma promessa para você. Ela tentou pegar um horário e não tinha, deu uma promessa rejeitada, não tinha. Ela precisou te avisar que não tinha, ela pegou essa informação e repassou para você.

[00:00:21] Tem dois termos que vamos usar bastante nesse vídeo, o try, "tentar", e o catch, sendo o "pegar". Vamos aplicar essa forma de acontecer um reject no código.

[00:00:33] Vamos abrir o Visual Studio Code, e no início, antes da declaração da variável consultaCEP, você vai colocar try, sendo o “tentar”, abre e fecha chaves. Vamos pegar o fechamento e colocar abaixo do console.log. E depois colocar catch (erro), abre e fecha chaves.

[00:00:55] Dentro dessas chaves, vamos colocar console.log(erro). Vamos salvar, e analisar o resultado no navegador. Na tela de cadastro, em que estávamos com o formulário do AluraBooks.

[00:01:11] Vou clicar na tecla "F5", está dando tudo certo. Agora vamos testar o que eu fiz do try e catch. Vamos minimizar o Visual Studio Code, no navegador, na tela de cadastro e clicar na tecla "F5". Continua aparecendo o endereço, mas isso porque nosso CEP está certo.

[00:01:27] Eu vou tirar um zero do Fetch API só para forçar esse erro. Salvei e vou clicar na tecla "F5", está demorando um pouco. Ele retornou aqueles dois erros, em vermelho, que eu até comentei - erro do navegador, não e não do código. E ele imprimiu o failed to fetch, o erro que gerou do código.

[00:01:48] Estamos captando o erro, e a partir daquele ciclo que aconteceu na consulta, é bem fácil de entender fazendo essa tradução. Porque ele vai tentar fazer tudo aquilo para captar o endereço, aquela lista que diz endereço, cidade, estado, etc. Senão, ele vai pegar o erro e mostrar na tela.

[00:02:07] É até mais fácil de compreender como funciona, desse jeito. E lembra que quando estávamos vendo sobre o tratamento de erros, com o then e o catch, vimos que a viaCEP, ela manda um erro diferente.

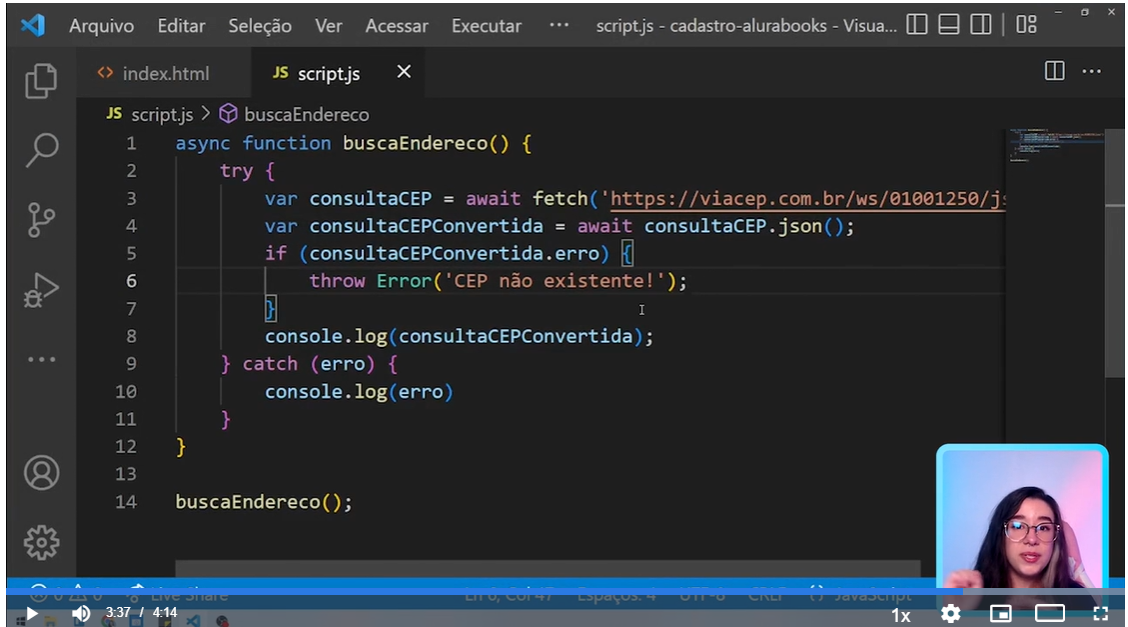
[00:02:20] Caso sejam CEPs com os dígitos que são necessários, mas ele não exista, eles mandam um erro igual a true. Eles não mandam um erro 400, mas sim dessa maneira, então precisaremos fazer uma condicional novamente.

[00:02:34] Dentro do try, antes do console.log, vamos colocar if (consultaCEPConvertida.erro), abrir e fechar chaves. Dentro, vamos colocar throw Error('CEP não existente!'). Fecha as chaves embaixo, e o console.log embaixo, porque se não tiver erro, ele vai continuar.

[00:03:10] Vamos salvar e colocar um CEP que sabemos que vai dar esse erro: 01001250, na linha três. Salvei e vamos forçar esse erro. Vamos no navegador, clicar na tecla "F5", e retornou "CEP não existente!". Com isso, já conseguimos fazer uma mensagem de erro customizada.

[00:03:31] Essa parte em específico é bem parecida. Forçamos um erro e, se acontecer essa situação para exibir esse erro, ele vai pegar com o catch e imprimir na tela. Essa é basicamente a estrutura, não é muito diferente do que vimos antes, mas não conhecíamos o try, e nesse caso, é ele que temos que usar, em conjunto ao catch.

[00:03:52] Porém, e se quiséssemos fazer várias requisições em simultâneo? Por exemplo, fazer um acompanhamento mensal no dentista. Como faríamos várias requisições ao mesmo tempo, sem utilizar "Ctrl + C" e "Ctrl + V" na mesma função e aplicar arranjos? Veremos no próximo vídeo uma maneira de fazermos isso! Até lá!



[00:00:00] Vamos supor que você queira fazer um acompanhamento mensal com esse dentista que estávamos marcando consulta. Você teria que fazer várias requisições, solicitando dias e horários diferentes. Isso não é diferente do que acontece nessas nossas consultas e contatos com a API.

[00:00:18] Por exemplo, no do viaCEP que estamos consumindo. Pode ser que você queira ver vários CEPs ao mesmo tempo. Como você faria isso? Vamos descobrir agora, em conjunto com o promise.all.

[00:00:32] Porém, primeiro vamos transformar essa função em algo que está esperando receber um parâmetro, um valor de parâmetro. Na primeira linha, vamos ter async function buscaEndereco(). Dentro dos parênteses, você vai colocar a palavra CEP, ou qualquer uma que você quiser, isso não vai importar muito.

[00:00:51] No fetch, vamos colocar no lugar em que tinha as aspas simples ou aspas duplas, crase. Coloquei no início e no final. E onde tinha o valor do CEP, agora vamos colocar um cifrão (“$”), abre e fecha chaves. Dentro delas, vamos colocar aquele valor que esperamos receber como parâmetro.

[00:01:13] Como eu tinha colocado o CEP, também vou colocar ${cep}. Depois, vamos colocar um retorno, que vai retornar o (consultaCEPConvertida). Vamos incluir após o console.log, na linha oito. Clicamos na tecla "Enter", então vai ser na linha nove.

[00:01:31] Vou colocar return consultaCEPConvertida. Ele vai retornar para quem estiver chamando essa função, esse valor. No final, antes da chamada da função, vamos colocar let ceps =, vamos incluir um array de CEPs aleatórios. Vou colocar aquele que estávamos usando: 01001000.

[00:01:56] Vírgula, abre chaves de novo, e vamos colocar a mesma coisa, mas terminando em um: 01001001. Eu tenho o meu array de CEPs, e vamos fazer um array de conjunto de CEPs. Ele vai fazer várias buscas. Para isso, vamos incluir ceps.map(). E dentro do map faremos uma arrow function igual as que fizemos nos outros lugares.

[00:02:22] Vamos colocar valores =>. Ele vai pegar o endereço, buscaEndereco(), e dentro dos parênteses ele vai colocar esses valores que ele estava pegando. Então, aqui ele vai fazer um novo array com o que retornar daquela função buscaEndereco, para cada um dos valores de dentro do CEP. Esses valores vão ser promessas, e precisamos resolver essas promessas.

[00:02:51] E para isso, usaremos o promise.all. O promise.all(conjuntoCeps), ele vai resolver o array de promessas, e vamos pedir para imprimir com o then. then(respostas => console.log(respostas)). Porque eu quero imprimir o que ele vai ter resolvido.

[00:03:20] Salvei e vamos testar no navegador. Cliquei na tecla "F5", ele fez cada uma das situações separadamente. Cada vez que ele chamou a função, ele passou naquele console.log de dentro da função. Depois imprimimos o promise.all, o resultado dele.

[00:03:39] Ele retornou um array com as duas promessas resolvidas. Para entender melhor o que aconteceu, podemos tentar imprimir aquele conjunto CEPs, aquele que chamou duas vezes a função buscaEndereco, para verificar o que acontece antes de botar o promise.all.

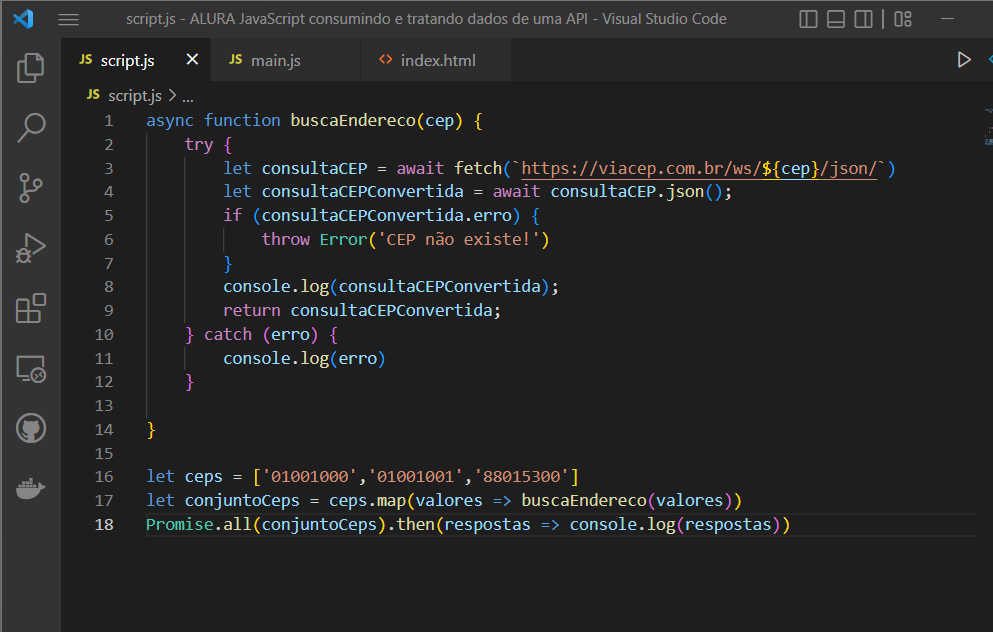
[00:03:54] Então, console.log(conjuntoCeps). Salvei, vou no navegador e vou atualizar. Lembra que antes tinha aparecido os dois resultados do endereço, e depois um array com as duas promises revolvidas.

[00:04:11] Quando colocamos o console.log, para ele agir antes do promise.all, ele retornou duas Promises. Elas não estavam resolvidas e com todo o corpo possível para visualizarmos na tela.

[00:04:21] É isso que o promise.all fez, ele nos ajudou a fazer várias requisições ao mesmo tempo. Entendendo para que funciona o promise.all, já podemos tirar essa chamada gigante, que não vamos fazer várias requisições para resolver esse problema que estamos tentando solucionar para a API do viaCEP, sendo aquela normalização de dados.

[00:04:40] Eu vou apagar essas linhas que eu fiz fora da função. E o que precisamos fazer agora? Nós já temos uma busca dinâmica, que ele pega um parâmetro e altera na URL. Mas só mostramos no console, na Ferramenta do Desenvolvedor.

[00:04:57] Como faremos para colocar isso lá no formulário, para facilitar o cadastro do usuário quando ele estiver preenchendo? Vamos manipular o DOM para chegar em um resultado muito legal, que quando o usuário coloca o CEP dele, os outros campos são completados automaticamente. Vamos aprender isso na próxima aula. Até lá!



# Para saber mais: Then ou Async Await?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/javascript-consumindo-tratando-dados-api/task/109379/next)

Quando produzimos um código assíncrono com o uso do .then nós fazemos uso de callback dentro deles. O maior problema com callbacks é que eles não são bem dimensionados mesmo para códigos assíncronos moderadamente complexos, onde temos vários .then em seguida do outro. O código resultante geralmente se torna difícil de ler, fácil de quebrar e difícil de depurar. Isso é o que chamamos de **callback hell**.

Para resolver isso, foi desenvolvido outra forma de construir um código assíncrono: o **async await**, que funciona de forma semelhante ao **then** mas o código fica mais “bonito”. Esse “embelezamento” em códigos é o que chamamos de **syntax sugar**.

Em ciência da computação, syntax sugar ou açúcar sintático (em tradução literal), é a sintaxe dentro de uma linguagem de programação que foi concebido para tornar as coisas mais fáceis de ler ou expressar. Isso torna a linguagem "mais doce" para uso humano: as coisas podem ser expressas de forma mais clara, de forma mais concisa, ou em um estilo alternativo que alguns podem preferir.

O async/await apesar de ser uma opção mais "legível" ao .then() é importante frisar que não são logicamente equivalentes: o async/await faz o processamento de forma sequencial, Promises com .then() são processadas em paralelo, o que faz com que este método seja mais rápido. O async/await simplifica a escrita e a interpretação do código, mas não é tão flexível e só funciona com uma Promise por vez.

O artigo [“Async/await no JavaScript: o que é e quando usar a programação assíncrona?”](https://www.alura.com.br/artigos/async-await-no-javascript-o-que-e-e-quando-usar) pode te ajudar a entender ambos os casos e suas diferenças.

## Nessa aula, você aprendeu como:

* Construir funções assíncronas com async;
* Criar requisições com await;
* Tratar erros com try catch;
* Implementar várias requisições simultaneamente com Promise All.

[00:00:00] Até aqui, aprendemos muitos conceitos JavaScript assíncronos. Nós consumimos uma API, aprendemos a pegar os valores dela, aprendemos a converter o retorno, a tratar erros. Isso é um aprendizado que vai servir para muita coisa, não só para esse problema que estamos tentando resolver.

[00:00:18] Não se esqueça que somos uma dupla de desenvolvedores da Alura e estamos enfrentando um problema do AluraBooks, que é a normalização de dados de cadastro dos usuários.

[00:00:28] Os clientes estavam cadastrando o nome de uma mesma rua de formas diferentes, mesmo sendo vizinhos, queremos evitar que isso aconteça. Então, vamos consumir aquela API e fazer o autocomplete. O cliente vai preencher o CEP e terá como retorno o bairro, a cidade, etc., tudo automaticamente.

[00:00:45] O primeiro passo é selecionarmos o campo de CEP, faremos tudo em volta dele. Vamos ao index.html e depois procuraremos o campo de CEP. Não é isso? Eu tinha passado, ele está próximo à linha 103. E o ID dele é CEP, precisamos só dessa informação.

[00:01:04] Então, vou procurar e, no final do script.js, eu vou criar uma variável chamada CEP, var cep. Nela, vou colocar document.getElementById('cep'). Depois eu vou adicionar cep.addEventListener("focusout"). Abriu uma janela gigante.

[00:01:20] Então, ("focusout", () => buscaEndereco(cep.value)). O que aconteceu? Eu criei uma variável CEP, atribuí um caminho grande, o document, quer dizer que ele é um documento HTML inteiro.

[00:01:25] Para isso, usei o método getElementById, o que ele faz? Assim como a sua tradução, ele pega um elemento pelo seu ID. O ID que tínhamos olhado no HTML é o CEP. Então, ele puxou todo aquele campo, não somente o valor que o usuário inseriu.

[00:02:13] Procurou o elemento em si, o elemento HTML. Depois, eu pedi para pegar esse elemento HTML e colocar um ouvinte de eventos para ficar cuidando da vida dele e saber quando acontece determinado evento, o defini como focusout, um evento que ocorre quando a pessoa clica na parte de fora.

[00:02:34] Ao escrever o CEP, ela clicou naquele campo. Ele está selecionado, está com o foco ativo. Após escrever, qualquer lugar em que ela clicar vai tirar o foco. Isso é o focusout.

[00:02:46] Quando ele acontece, chama o buscaEndereço, e manda o valor do CEP. Lembra que tínhamos feito essa parte mais dinâmica no promise.all? Ele recebe um parâmetro e altera - conforme o parâmetro que recebe - a URL do viaCEP?

[00:03:02] Isso é chamado de Template Strings, que é basicamente composto dessas crases e de valores dinâmicos colocados através do cifrão e das chaves ${}. Com isso, nós já passeamos pelo DOM do nosso site.

[00:03:19] Conseguimos encontrar o valor do campo do input, e enviar para fazer essa consulta dinâmica. Mas ainda não colocamos nada, podemos até testar! Vamos colocar, salvar, acessar no navegador e clicar "F5". Vou colocar o CEP padrão que estávamos usando na função: 01001000 e depois tirar o foco do campo de CEP.

[00:03:47] Coloquei e tirei, cada vez que eu coloco algo e tiro, ele faz uma requisição nova. Já está uma consulta muito dinâmica. Agora precisamos fazer com que essas informações que foram retornadas, como logradouro, estado e cidade, sejam completadas automaticamente. E é isso que, no próximo vídeo, vamos trazer!

[00:00:00] Já conseguimos captar o valor do campo de CEP, fazer uma consulta dinâmica e agora, para facilitar a vida, tanto do cliente quanto a nossa, na nossa normalização de dados, vamos fazer o preenchimento automático.

[00:00:15] O primeiro passo é selecionar aqueles campos de cidade, de rua e de estado. Vamos verificar no HTML quais são os IDs desses campos. Primeiro, endereço, id="endereco", o ID de campo é id="cidade", e o ID de estado, é id="estado". Estão bem condizentes.

[00:00:33] Dentro do try, vamos colocar, após a condicional do erro. Selecionamos var cidade = document.getElementById('cidade'). Depois, var logradouro =, que é a rua. document.getElementById('endereco').

[00:00:33] No endereço não tem cedilha, está certo. Vou voltar no script, var estado = document.getElementById('estado'). Já selecionei os três campos do formulário para cada coisa.

[00:01:37] Era basicamente o que tínhamos feito para o elemento id="cep". Então, definimos uma variável com nome condizente, para facilitar. Depois, colocamos o documento. Ele pega o documento HTML inteiro. Então, pegamos o getElementById, porque ele procura o elemento pelo ID, e colocamos nessas variáveis.

[00:02:00] Agora conseguimos acessá-las, mudar o valor ou qualquer coisa que quisermos. Mas o que queremos? Colocar o valor de CEP. Para isso, escreveremos, por exemplo, cidade.value, e assim conseguimos acessar o valor. = consultaCEPConvertida., qual era o nome deste valor? .localidade;, que é o nome do valor que retorna a API com valor de cidade.

[00:02:17] Depois vamos colocar logradouro.value = e consultaCEPConvertida.logradouro;, que é o nome da nossa rua. Para o estado, vamos colocar estado.value = consultaCEPConvertida.uf; Salvei e agora vamos testar.

[00:02:18] Salvei e vou no Consulta CEP de novo, copiaremos com "Ctrl + C" esse CEP para não perdermos. É aquele CEP padrão para a API. Vou clicar "F5" e colocar o valor dentro dele, depois vou clicar fora desse campo. Agora, ele completou automaticamente o endereço da "Praça da Sé", a cidade e o UF. Está correto, não tem mais erro.

[00:03:26] Se você quiser verificar qual é o valor, basta ir ao Console, abrir o retorno e você vai ter as informações corretas. Se eu quiser autocompletar o bairro, eu posso pegar pelo bairro. Se eu quiser completar o "ddd", ou o "ibge", tenho várias informações no Console. Nós pegamos a localidade, logradouro e UF.

[00:03:47] Vamos entender como funcionou. Antes eu tinha feito a variável com esses elementos. Tendo o elemento, eu consigo acessar o valor. Eu peguei o nome da variável .value e acessei o campo do valor dele. Depois atribuí, com o igual, o valor retornado da API que eu mostrei agora no console.log.

[00:04:10] Ele teve consultaCEPConvertida, que é tudo o que apareceu, o ponto, a categoria que existe dentro, enfim, fica bem específico, como consultaCEPConvertida.localidade;. Vou olhar no navegador o que é isso. A localidade é São Paulo. Depois, eu coloquei consultaCEPConvertida.logradouro;, é "Praça da Sé".

[00:04:31] Então, é assim que acessamos os valores que estão dentro desse retorno, desse objeto, desse json. Mas, lembrando outra vez, se a pessoa colocar um CEP errado, não temos como prever o que o usuário vai fazer no nosso sistema, mas podemos nos prevenir sobre o que pode acontecer, podemos imprimir um erro na tela. E como fazemos isso? Vou te contar no próximo vídeo! Até logo!

[00:00:00] Um usuário pode preencher errado o campo de CEP e é o nosso dever avisar. Antes estávamos só captando e mostrando no console.log esses erros. Como faremos para mostrar na tela?

[00:00:14] Novamente, vamos mexer no DOM, para isso, abriremos o index.html desse projeto no Visual Studio Code. Embaixo do campo de CEP, vamos criar uma div com id="erro", só para ele ocupar o espaço que vai ficar a nossa mensagem.

[00:00:35] Vamos no script.js e antes do try, colocaremos var mensagemErro. Atribuiremos para ele o document.getElementById, esse que acabamos de criar, o erro. E vamos iniciar com nenhuma mensagem. Então, mensagemErro.innerHTML = "". Não vai ter nada dentro dele.

[00:01:04] O que estamos fazendo aqui? Estamos selecionando aquela div que acabamos de fazer pelo ID e colocando dentro do HTML, com o innerHTML, uma mensagem vazia. Deve iniciar com vazio, porque se não tem erro, não é para ter nada lá.

[00:01:20] E como faremos para inserir essa mensagem de erro? Usaremos o innerHTML dentro do Catch. Então, no Catch, colocaremos mensagemErro.innerHTML = e o elemento HTML de parágrafo. Você vai usar <p>, e vamos colocar a mensagem CEP inválido. Tente novamente.

[00:01:46] Vamos fechar com </p>. Agora temos uma mensagem que vai aparecer quando pegar algum erro. Vamos testar? Salvei e vou ao navegador.

[00:01:51] Apertarei "F5" para recarregar e vou colocar aquele CEP que estávamos testando por padrão. Vou tirar um zero para forçar o erro e clicar fora. Ele está pensando. Deu erro e apareceu embaixo: CEP inválido. Tente novamente. Também apareceu no console.log porque eu deixei.

[00:02:24] Já conseguimos informar para o usuário quando acontece algo errada, oferecendo-lhe autonomia para arrumar esse erro e tentar novamente, tentar de outra maneira. O que você acha? Essa mensagem podia ser mais específica, podia explicar mais coisas? Você pode também, com a sua criatividade, colocar uma mensagem diferente.

[00:02:46] Com isso, conseguimos normalizar boa parte do nosso problema dos dados. Agora, quando o usuário digitar o CEP, todos que possuírem o mesmo CEP, a rua, o nome da cidade, o estado, até o bairro, que eu não coloquei e você pode colocar como desafio, tudo será normalizado, igual entre todos os vizinhos e não teremos esse problema que estávamos enfrentando.

[00:03:08] Então, muito obrigada por me acompanhar no desenvolvimento dessa solução para a nossa equipa da Alura. Eu acredito que você vai conseguir usar esse aprendizado em vários outros projetos seus. Eu te espero nas redes sociais, no compartilhamento desse aprendizado que você teve aqui. Até mais!

Dentro da área de experiência do usuário, os designers estudam as [**heurísticas de Nielsen**](https://www.alura.com.br/artigos/10-heuristicas-de-nielsen-uma-formula-pra-evitar-erros-basicos-de-usabilidade), que ajudam a projetar uma boa interface e por consequência uma ótima experiência de uso, durante o desenvolvimento desse projeto foi pensado em duas delas:

* Prevenção de erros

Não é uma boa ideia deixar seu usuário errar sem explicar previamente o motivo do erro. Melhor do que isso, tente criar uma interface que permita ao usuário não errar. Para isso, aplicamos o mecanismo de auto completar o endereço de acordo com o CEP do usuário, evitando que os dados sejam enviados errados e ele não consiga receber seu pedido.

* Ajude os usuários a reconhecerem, diagnosticarem e recuperarem-se de erros.

As mensagens de erros tem que ser claras e próximas do conteúdo ou ação que causou o erro. Por isso, aplicamos uma mensagem que aparece abaixo do campo de CEP caso o usuário digite ele incorretamente. Isso ajudará a detectar e resolver possíveis problemas.

Para ambas heurísticas, a solução foi aplicada com a manipulação do DOM. O Document Object Model (DOM) é uma interface de programação para os documentos HTML e XML. Representa a página de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo. O DOM representa o documento com nós e objetos, dessa forma, as linguagens de programação podem se conectar à página. Para entender mais sobre isso, aprenda outras maneiras de transformar seu **HTML** através do **JavaScript** com o instrutor Pedro Marins curso [“JavaScript: manipulando o DOM”](https://cursos.alura.com.br/course/javascript-manipulando-dom).

Parabéns pela conclusão do projeto, estudante! Se você chegou até aqui, espero que tenha seu projeto pronto (e se precisar de alguma ajuda, não deixe de abrir um tópico no fórum ou mandar mensagem no discord da Alura) e tenha adquirido muitos conhecimentos novos.

Agora para hospedar o projeto desenvolvido, compartilho o artigo [“Como colocar seu projeto no ar com o Github Pages?”](https://www.alura.com.br/artigos/como-colocar-projeto-no-ar-com-github-pages) do instrutor Neilton Seguins.

Que tal compartilhar seu certificado, seu aprendizado até aqui ou até seu projeto comigo? Para isso, você pode:

* Marcar a Alura nas redes sociais. Você pode encontrar os nossos perfis por [aqui](https://beacons.ai/aluraonline/);
* Me marcar através das minhas redes sociais, que podem ser visualizadas por [aqui](https://linktr.ee/monicahillman);
* Enviar mensagem no [Discord](https://discord.gg/QeBdgAjXnn" \t "_blank) de alunos da Alura.
* [00:00:00] Olá, estudante! **Parabéns!!** Se você chegou até aqui, quer dizer que você concluiu este treinamento sobre **consumo e tratamento de dados de uma \*API**. Então, vamos recapitular o que você aprendeu. Primeiro, como identificar o fluxo de eventos do JavaScript e como ele funciona, através do Event Loop, do Task Queue e do Call Stack.
* [00:00:21] Depois, como consumir uma API, e compreendeu como isso funciona por trás dos panos, através do Fetch API, das Promises e dos métodos dessas Promises. Além disso, você aprendeu a tratar os dados retornados de API e aplicou tanto quando construiu as funções assíncronas, e quanto nas Promises, com os métodos nativos dela.
* [00:00:43] Por último, você aprendeu a manipular elementos através de interação do usuário. Ou seja, manipulou o DOM para captar o valor que aquele usuário colocou na tela de cadastro e depois, para autopreencher aqueles valores retornados da API, nos outros campos desse site.
* [00:01:00] Então, vamos olhar o projeto em si. Estou com o formulário de cadastro, na tela de cadastro da AluraBooks e vamos testar.
* [00:01:11] Vou lá embaixo, no campo de endereço, no CEP e preencherei o CEP padrão do viaCEP: 01001000, depois clicarei fora desse campo. Ele já detectou o que aconteceu e autopreencheu o endereço com "Praça da Sé", com cidade "São Paulo" e com o Estado "São Paulo".
* [00:01:29] E se eu colocar um endereço errado, o que vai acontecer? Vou clicar "F5" para testarmos novamente. Eu vou colocar um CEP que não existe: 01001250 e ele deu problema.
* [00:01:42] Agora vou dar um clique com o mouse fora do campo do CEP, e ele alertou: "CEP inválido. Tente novamente!". Já tratamos, lidamos com os erros e informamos na tela para o usuário. Esse conhecimento você vai poder aplicar em outros projetos também.
* [00:01:54] Não só aqui na tela de cadastro do AluraBooks, você pode colocar em algum projeto seu do Challenge, algum treinamento que você esteja fazendo, que tenha um e-commerce, um formulário de cadastro aleatório. Por último, eu quero deixar os meus parabéns por você ter concluído esse treinamento.
* [00:02:11] Eu quero ver você estudando ainda muito mais, não vamos parar por aqui! Então, compartilhe nas redes sociais, você pode me marcar, pode marcar a Alura. Vamos contar para o mundo qual foi o conhecimento que você adquiriu aqui, o resultado do seu projeto.
* [00:02:25] Quem sabe você faz outros exercícios, outras aplicações com que foi aprendido aqui. Me marca por lá que eu quero ver! E também não esqueça de deixar a sua avaliação sobre esse curso. Ao final desse vídeo, provavelmente você receberá um formulário para atribuir uma nota de 0 a 10 e deixar a sua opinião.
* [00:02:42] Essa opinião é muito importante para melhorarmos e entendermos quais são as suas necessidades, para fazer você um desenvolvedor Front End cada vez melhor! Até logo! Te vejo em outros cursos!