2/7/2020

# Asynchronous JavaScript

Personal notes by Latif Essam

Ref: Codecademy

Latif Essam
CODECADEMY, MDN

# Contents

# Http Requests: basics

Understand the basics of how your web browser communicates with the internet.

Background:

This page is generated by a web of HTML, CSS, and JavaScript, sent to you by Codecademy via the internet. The internet is made up of a bunch of resources hosted on different servers. The term “resource” corresponds to any entity on the web, including HTML files, stylesheets, images, videos, and scripts. To access content on the internet, your browser must ask these servers for the resources it wants, and then display these resources to you. This protocol of requests and responses enables you view *this* page in your browser.

## HTTP Requests

Web developers use the event loop to create a smoother browsing experience by deciding when to call functions and how to handle asynchronous events.
MDN Documentation: Event Loop

### XHR GET Requests I

MDN Documentation: Extensible Markup Language (XML).

Example:

```
const xhr =new XMLHttpRequest();
const url = 'https://api-to-call.com/endpoint';
xhr.responseType = 'json';
xhr.onreadystatechange = ()=>{
  if(xhr.readyState === XMLHttpRequest.DONE) {
    return xhr.response;
  }
}
xhr.open('GET',url);
xhr.send();
```

```
// XMLHttpRequest GET
                creates new object
const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';

xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE) {
      // Code to execute with response
    }
};

xhr.open('GET', url);
xhr.send();
```

handles response

opens request and sends object

A query string is separated from the URL using a `?` character. After `?`, you can then create a parameter which is a key value pair joined by a `=`. Examine the example below:

```
'https://api.datamuse.com/words?key=value'
```

If you want to add an additional parameter you will have to use the `&` character to separate your parameters. Like so:

```
'https://api.datamuse.com/words?key=value&anotherKey=anotherValue'
```

# XHR POST Requests

The major difference between a GET request and POST request is that a POST request requires additional information to be sent through the request. This additional information is sent in the *body* of the post request.

Example:

```
const xhr= new XMLHttpRequest();
const url = 'https://api-to-call.com/endpoint';
const data = JSON.stringify({id: '200'});
xhr.responseType = 'json';
xhr.onreadystatechange = ()=>{
  if(xhr.readyState === XMLHttpRequest.DONE){
    return xhr.response;
  }
}
xhr.open('POST',url);xhr.send(data);
```

```
// XMLHttpRequest POST
                creates new object
const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';
const data = JSON.stringify({id: '200'});   converts data to a string

xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE) {   handles response
    // Code to execute with response
  }
};
xhr.open('POST', url);
xhr.send(data);                                   opens  request and sends object
```

Example of REbrandly app: which take a link and give new short new link with the same destination.

main.js

```
// Information to reach API
const apiKey = '66b53ab1341c41f187f1e19451232a77';
const url = 'https://api.rebrandly.com/v1/links';
// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');
// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value;
//API expects to see an object with a key destination that has a value of a URL.
  const data = JSON.stringify({destination: urlToShorten});
  const xhr = new XMLHttpRequest();
  xhr.responseType = 'json';
  xhr.onreadystatechange=()=>{
    if(xhr.readyState === XMLHttpRequest.DONE){
      renderResponse(xhr.response);
    }
  }
xhr.open('POST',url);
xhr.setRequestHeader('Content-type', 'application/json');
xhr.setRequestHeader('apikey', apiKey);
xhr.send(data);

}
// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  shortenUrl();
}
shortenButton.addEventListener('click', displayShortUrl);
```

# *REQUESTS II*

## *Introduction to Requests with ES6:* Monster Level

Many of our web page interactions rely on asynchronous events, so managing these events is essential to good web development.

To make asynchronous event handling easier, *promises* were introduced in JavaScript in ES6:

- Mozilla Development Network: Promises

A promise is an object that handles asynchronous data. A promise has three states:

- *pending*: when a promise is created or waiting for data.
- *fulfilled*: the asynchronous operation was handled successfully.
- *rejected*: the asynchronous operation was unsuccessful.

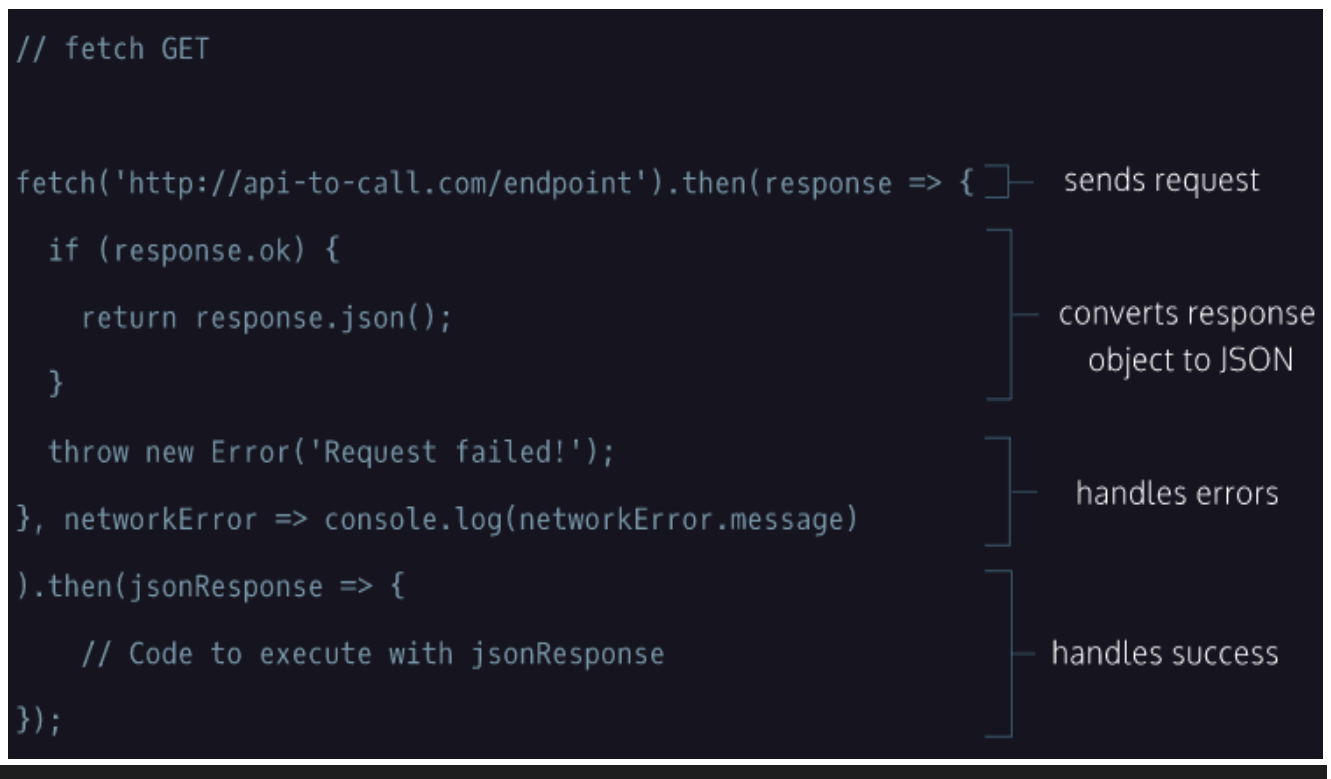
The great thing about promises is that once a promise is fulfilled or rejected, you can chain an additional method to the original promise. we will use `fetch()`, which uses promises to handle requests. Then, we will simplify requests using `async` and `await`.

## fetch () GET Requests

The first type of requests we're going to tackle are GET requests using `fetch (),` MDN: Fetch API.

The `fetch ()` function:

- Creates a request object that contains relevant information that an API needs.
- Sends that request object to the API endpoint provided.
- Returns a promise that ultimately resolves to a response object, which contains the status of the promise with information the API sent back.




Example:

```
fetch('https://api-to-
call.com/endpoint')
.then(
  response => {
   if (response.ok) return response.json();
    throw new Error('Request failed!');
    },
   networkError=> console.log(networkError.message);
)
.then(jsonResponse=> return jsonResponse);
```

REQUESTS II

# fetch() POST Requests

```
// fetch POST


fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'})
}).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
}, networkError => console.log(networkError.message)
).then(jsonResponse => {
    // Code to execute with jsonResponse
});
```

sends request

converts response object to JSON

handles errors

handles success

Example:

```
fetch('https://api-to-
call.com/endpoint',
{method:'POST', body: JSON.stringify({id:'200'})}
).then(response =>{
  if(response.ok){
    return response.json();
  }
  throw new Error('Request failed!');
},
networkError=>{console.log(networkError.message)}
).then(jsonResponse=>{
  return jsonResponse;
})
```

Using fetch to make POST Request:

```
// Information to reach API
const apiKey = '66b53ab1341c41f187f1e19451232a77';
const url = 'https://api.rebrandly.com/v1/links';
// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');
// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value;
  const data = JSON.stringify({destination:urlToShorten});
  fetch(url,{method:'POST',headers:{'Content-type':'application/json','apikey':apiKey},body:data})
  .then(response=>{
    if(response.ok) return response.json();
    throw new Error('Request failed!');
  },
  networkError => {console.log(networkError.message)}
  ).then(jsonResponse =>{renderResponse(jsonResponse)});
}
// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild)
  }
  shortenUrl();
}
shortenButton.addEventListener('click', displayShortUrl);
```

# async GET Requests ES8: **Dragon Level**

Keep in mind:

- Using an `async` function that will return a promise.
- `await` can only be used in an `async` function. `await` allows a program to run while waiting for a promise to resolve.
- In a `try...catch` statement, code in the `try` block will be run and in the event of an exception/error, the code in the `catch` statement will run.

```
// async await GET


async function getData() {
  try {
   const response = await fetch('https://api-to-call.com/endpoint');   <- sends request
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) {
    console.log(error);
  }
}
```

handles response if successful

handles response if unsuccessful

Example:

```
const getData = async ()=>{
  try {
    const response = await fetch('https://api-to-call.com/endpoint');
    if(response.ok){
      const jsonResponse = await response.json();
      return jsonResponse;
    }
    throw new Error('Request failed!')
  }
  catch(error){
    console.log(error);
  }
}
```

Build get request using async await method

```
// Information to reach API
const url = 'https://api.datamuse.com/words?';
const queryParams = 'rel_jja=';

// Selecting page elements
const inputField = document.querySelector('#input');
const submit = document.querySelector('#submit');
const responseField = document.querySelector('#responseField');
// AJAX function && Code goes here
const getSuggestions = async ()=>{
  const wordQuery = inputField.value;
  const endpoint= url + queryParams + wordQuery;
  try{
    const response = await fetch(endpoint,{cache:'no-cache'});
    if(response.ok){
      const jsonResponse = await response.json();
      renderResponse(jsonResponse);
    }
  }catch(error){
    console.log(error);
  }
}
// Clear previous results and display results to webpage
const displaySuggestions = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild)
  }
  getSuggestions();
}
submit.addEventListener('click', displaySuggestions);
```

## async POST Requests ES8: **Dragon Level**

an `async` POST request requires more information.

Like in the `fetch()` call, we now have to include an additional argument that contains more information like `method` and `body`.

```
// async await POST


async function getData() {
  try {
    const response = await fetch('https://api-to-call.com/endpoint', {
      method: 'POST',
      body: JSON.stringify({id: '200'})
    });
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) {
    console.log(error);
  }
}
```

sends request

handles response if successful

handles response if unsuccessful

Syntax:

```
const getData = async ()=>{
  try{
    const response = await fetch('https://api-to-call.com/endpoint',{
      method:'POST',
      body:JSON.stringify({id:200})
    });
    if(response.ok){
      const jsonResponse = await response.json();
      return jsonResponse;
    }
    throw new Error('Request failed!');
  }
  catch(error){
    console.log(error);
  }
}
```

Build POST request using async await.

```
// information to reach API
const apiKey = '66b53ab1341c41f187f1e19451232a77';
const url = 'https://api.rebrandly.com/v1/links';
// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');
// AJAX functions
// Code goes here
const shortenUrl = async ()=>{
  const urlToShorten = inputField.value;
  const data = JSON.stringify({destination: urlToShorten});

  try{
    const response =await fetch (url,{
      method:'POST',
```

1.

```
      body:data,
      headers:{
        'Content-type': 'application/json',
        'apikey': apiKey
        }
    });
    if(response.ok){
      const jsonResponse = await response.json();
      renderResponse(jsonResponse);
    }
  }
  catch(error){
    console.log(error);
  }
}
// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  shortenUrl();
}
shortenButton.addEventListener('click', displayShortUrl);
```

# Review Requests II

Let's recap on the concepts covered in the previous exercises:

1. GET and POST requests can be created a variety of ways.
2. Use AJAX to asynchronously request data from APIs. `fetch()` and `async`/`await` are new functionalities developed in ES6 (promises) and ES8 respectively.
3. Promises are a new type of JavaScript object that represent data that will eventually be returned from a request.
4. `fetch()` is a web API that can be used to create requests. `fetch()` will return promises.
5. We can chain `.then()` methods to handle promises returned by `fetch()`.
6. The `.json()` method converts a returned promise to a JSON object.
7. `async` is a keyword that is used to create functions that will return promises.
8. `await` is a keyword that is used to tell a program to continue moving through the message queue while a promise resolves.
9. `await` can only be used within functions declared with `async`.

# *ASYNC AWAIT*

The `async` keyword is used to write functions that handle asynchronous actions.

```
async function myFunc() {
  // Function body here
};

myFunc();
```

or like this

```
const myFunc = async () => {
  // Function body here
};

myFunc();
```

`async` functions always return a promise. This means we can use traditional promise syntax, like `.then()` and `.catch` with our `async` functions. An `async` function will return in one of three ways:

- If there's nothing returned from the function, it will return a promise with a resolved value of `undefined`.
- If there's a non-promise value returned from the function, it will return a promise resolved to that value.
- If a promise is returned from the function, it will simply return that promise

```
async function fivePromise() {
  return 5;
}

fivePromise()
.then(resolvedValue => {
    console.log(resolvedValue);
  })  //
```

## The await Operator

The `await` keyword can only be used inside an `async` function. `await` is an operator: it returns the resolved value of a promise. `await` halts, or pauses, the execution of our `async` function until a given promise is resolved.

## Handling Dependent Promises

```
function nativePromiseVersion() {
    returnsFirstPromise()
    .then((firstValue) => {
        console.log(firstValue);
        return returnsSecondPromise(firstValue);
    })
   .then((secondValue) => {
        console.log(secondValue);
    });
}
```

Let's break down what's happening in the `nativePromiseVersion()` function:

- Within our function we use two functions which return promises: `returnsFirstPromise()` and `returnsSecondPromise()`.
- We invoke `returnsFirstPromise()` and ensure that the first promise resolved by using `.then()`.
- In the callback of our first `.then()`, we log the resolved value of the first promise, `firstValue`, and then return `returnsSecondPromise(firstValue)`.
- We use another `.then()` to print the second promise's resolved value to the console.

Here's how we'd write an `async` function to accomplish the same thing:

```
async function asyncAwaitVersion() {
 let firstValue = await returnsFirstPromise();
 console.log(firstValue);
 let secondValue = await returnsSecondPromise(firstValue);
 console.log(secondValue);
}
```

Let's break down what's happening in our `asyncAwaitVersion()` function:

- We mark our function as `async`.
- Inside our function, we create a variable `firstValue` assigned `await returnsFirstPromise()`. This means `firstValue` is assigned the resolved value of the awaited promise.
- Next, we log `firstValue` to the console.
- Then, we create a variable `secondValue` assigned to `await returnsSecondPromise(firstValue)`. Therefore, `secondValue` is assigned this promise's resolved value.
- Finally, we log `secondValue` to the console.

## Handling Errors

With `async...await`, we use `try...catch` statements for error handling.  By using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also catch both synchronous and asynchronous errors. This makes for easier debugging!

```
async function usingTryCatch() {
 try {
   let resolveValue = await asyncFunction('thing that will fail');
   let secondValue = await secondAsyncFunction(resolveValue);
 } catch (err) {
   // Catches any errors in the try block
   console.log(err);
 }
}

usingTryCatch();
```

Remember, since `async` functions return promises we can still use native promise's `.catch()` with an `async` function

```
async function usingPromiseCatch() {
   let resolveValue = await asyncFunction('thing that will fail');
}

let rejectedPromise = usingPromiseCatch();
rejectedPromise.catch((rejectValue) => {
console.log(rejectValue);
})
```

This is sometimes used in the global scope to catch final errors in complex code.

## Handling Independent Promises

```
async function serveDinner(){
  const vegetablePromise = steamBroccoli();
  const starchPromise = cookRice();
  const proteinPromise = bakeChicken();
  const sidePromise = cookBeans();
  console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${a
wait proteinPromise}, and ${await sidePromise}.`)
}

serveDinner();
```

# Await Promise.all()

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously is to `await` a `Promise.all()`.

Ex:

```
async function asyncPromAll() {
  const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);
  for (let i = 0; i<resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

`Promise.all()` allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. `Promise.all()` also has the benefit of *failing fast*, meaning it won't wait for the rest of the asynchronous actions to complete once any one has rejected.

the promise returned from `Promise.all()` will reject with that reason. As it was when working with native promises, `Promise.all()` is a good choice if multiple asynchronous tasks are all required, but none must wait for any other before executing.

**ASYNC AWAIT**

# Review

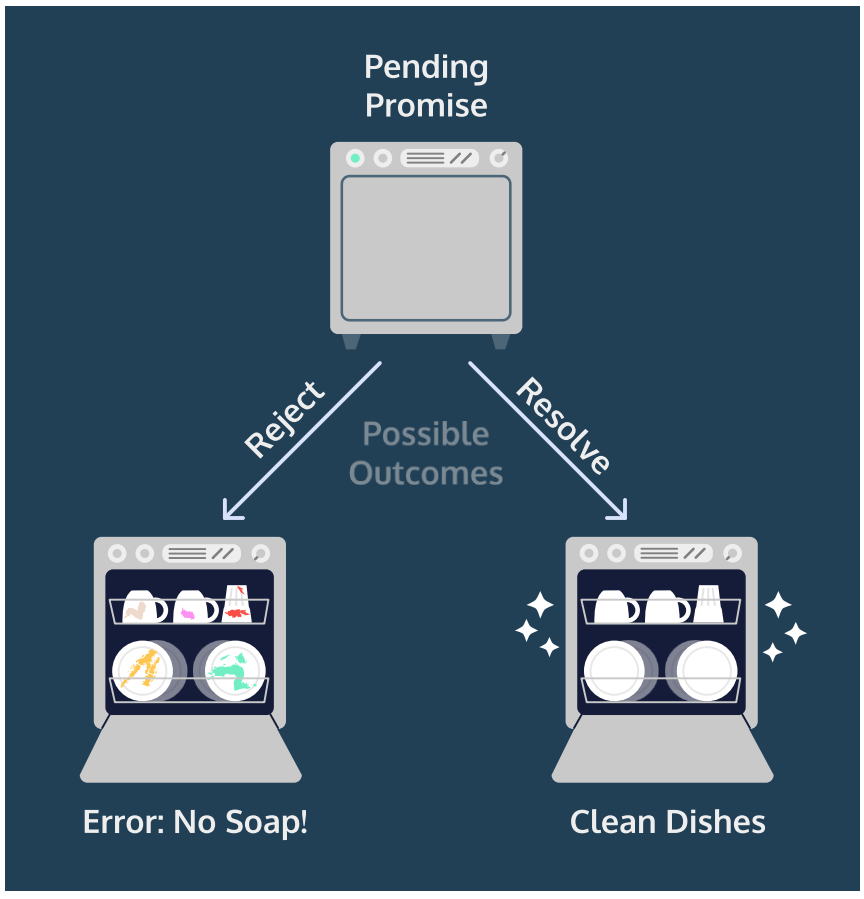
Awesome work getting the hang of the `async...await` syntax! Let's review what you've learned:

- `async...await` is syntactic sugar built on native JavaScript promises and generators.
- We declare an async function with the keyword `async`.
- Inside an `async` function we use the `await` operator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending.
- `await` returns the resolved value of the awaited promise.
- We can write multiple `await` statements to produce code that reads like synchronous code.
- We use `try...catch` statements within our `async` functions for error handling.
- We should still take advantage of concurrency by writing `async` functions that allow asynchronous actions to happen in concurrently whenever possible.

# What is a Promise?

Promises are objects that represent the eventual outcome of an asynchronous operation.

A Promise is an object representing the eventual completion or failure of an asynchronous operation.




A `Promise` object can be in one of three states:

- **Pending**: The initial state— the operation has not completed yet.

- **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request's promise might resolve with a JSON object as its value.
- **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an `Error` of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected.

## Constructing a Promise Object

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);
```

The `Promise` constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the `resolve()` and `reject()` functions.
The `resolve()` and `reject()` functions aren't defined by the programmer. When the `Promise` constructor runs, JavaScript will pass **its own** `resolve()` and `reject()` functions into the executor function.

- `resolve` is a function with one argument. Under the hood, if invoked, `resolve()` will change the promise's status from `pending` to `fulfilled`, and the promise's resolved value will be set to the argument passed into `resolve()`.
- `reject` is a function that takes a reason or error as an argument. Under the hood, if invoked, `reject()` will change the promise's status from `pending` to `rejected`, and the promise's rejection reason will be set to the argument passed into `reject()`.

A Promise's constructor takes as its argument a function, called the "executor function", as a single parameter.

## Consuming Promises

Promise objects come with an aptly named `.then()` method. It allows us to say, "I have a promise, when it settles, **then** here's what I want to happen..."

`.then()` is that it always returns a promise.

`.then()` is a higher-order function— it takes two callback functions as arguments (these callbacks called *handlers)*.

- The first handler, sometimes called `onFulfilled`, is a *success handler*, and it should contain the logic for the promise resolving.
- The second handler, sometimes called `onRejected`, is a *failure handler*, and it should contain the logic for the promise rejecting.

We can invoke `.then()` with one, both, or neither handler!

This is good but we should provide the 2 arguments and use whatever we want.

i.e.: This allows for flexibility, but it can also make for tricky debugging. If the appropriate handler is not provided, instead of throwing an error, `.then()` will just return a promise with the same settled value as the promise it was called on.

Example:

```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve('Yay!');
  } else {
    reject('Ohhh noooo!');
  }
});
const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};
const handleFailure = (rejectionReason) => {
  console.log(rejectionReason);
};
prom.then(handleSuccess, handleFailure);
```

- `prom` is a promise which will randomly either resolve with `'Yay!'` or reject with `'Ohhh noooo!'`.
- We pass two handler functions to `.then()`. The first will be invoked with `'Yay!'` if the promise resolves, and the second will be invoked with `'Ohhh noooo!'` if the promise rejects.

## Using catch() with Promises

One way to write cleaner code is to follow a principle called *separation of concerns*. Separation of concerns means organizing code into distinct sections each handling a specific task. It enables us to quickly navigate our code and know where to look if something isn't working.

So Instead of passing both handlers into one `.then()`, we can chain a second `.then()` with a failure handler to a first `.then()` with a success handler and both cases will be handled.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .then(null, (rejectionReason) => {
    console.log(rejectionReason);
  });
```

Since JavaScript doesn't mind whitespace, we can chain more and more of Promise's Functions like catch (),
The `.catch()` function takes only one argument, `onRejected`.

So using `.catch()`:

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

- `prom` is a promise which randomly either resolves with `'Yay!'` or rejects with `'Ohhh noooo!'`.
- We pass a success handler to `.then()` and a failure handler to `.catch()`.
- If the promise resolves, `.then()`'s success handler will be invoked with `'Yay!'`.
- If the promise rejects, `.then()` will return a promise with the same rejection reason as the original promise and `.catch()`'s failure handler will be invoked with that rejection reason.

## Chaining Multiple Promises

To check multiple Promises for resolved value which is called Promise composition.
Promises are designed with composition in mind! Here's a simple promise chain in code:

```
firstPromiseFunction()
.then((firstResolveVal) => {
  return secondPromiseFunction(firstResolveVal);//check the resolved value with another promise
secondPromiseFunction.
})
```

```
.then((secondResolveVal) => {
  console.log(secondResolveVal);
});
```

Let's break down what's happening in the example:

- We invoke a function `firstPromiseFunction()` which returns a promise.
- We invoke `.then()` with an anonymous function as the success handler.
- Inside the success handler we **return** a new promise— the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- We invoke a second `.then()` to handle the logic for the second promise settling.
- Inside that `.then()`, we have a success handler which will log the second promise's resolved value to the console.

In order for our chain to work properly, we had to `return` the promise `secondPromiseFunction(firstResolveVal)`. This ensured that the return value of the first `.then()` was our second promise rather than the default return of a new promise with the same settled value as the initial.

# Avoiding Common Mistakes

Promise composition allows for much more readable code than the nested callback syntax that preceded it.

**Mistake 1:** Nesting promises instead of chaining them.

```
returnsFirstPromise()
.then((firstResolveVal) => {
  return returnsSecondValue(firstResolveVal)
    .then((secondResolveVal) => {
      console.log(secondResolveVal);
    })
})
```

Let's break down what's happening in the above code:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we invoke `returnsSecondValue()` with `firstResolveVal` which will return a new promise.
- We invoke a second `.then()` to handle the logic for the second promise settling all **inside** the first `then()`!
- Inside that second `.then()`, we have a success handler which will log the second promise's resolved value to the console.

Instead of having a clean chain of promises, we've nested the logic for one inside the logic of the other. Imagine if we were handling five or ten promises!

**Mistake 2:** Forgetting to `return` a promise.

```
returnsFirstPromise()
.then((firstResolveVal) => {
  returnsSecondValue(firstResolveVal)
})
.then((someVal) => {
  console.log(someVal);
})
```

Let's break down what's happening in the example:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we create our second promise, but we forget to `return` it!
- We invoke a second `.then()`. It's supposed to handle the logic for the second promise, but since we didn't return, this `.then()` is invoked on a promise with the same settled value as the original promise!

# Using Promise.all()

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we're dealing with multiple promises, but we don't care about the order? Let's think in terms of cleaning again.

To maximize efficiency we should use *concurrency*, multiple asynchronous operations happening together. With promises, we can do this with the function `Promise.all()`.

`Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

- If every promise in the argument array resolves, the single promise returned from `Promise.all()` will resolve with an array containing the resolve value from each promise in the argument array.
- If any promise from the argument array rejects, the single promise returned from `Promise.all()` will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Let's look at a code example:

```
let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(), returnsPromThree()]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let's break down what's happening:

- We declare `myPromises` assigned to invoking `Promise.all()`.
- We invoke `Promise.all()` with an array of three promises— the returned values from functions.
- We invoke `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
- We invoke `.catch()` with a failure handler which will print the first rejection message if any promise rejects.

# Review

- Promises are JavaScript objects that represent the eventual result of an asynchronous operation.
- Promises can be in one of three states: pending, resolved, or rejected.
- A promise is settled if it is either resolved or rejected.
- We construct a promise by using the `new` keyword and passing an executor function to the `Promise` constructor method.
- `setTimeout()` is a Node function which delays the execution of a callback function using the event-loop.
- We use `.then()` with a success handler callback containing the logic for what should happen if a promise resolves.
- We use `.catch()` with a failure handler callback containing the logic for what should happen if a promise rejects.
- Promise composition enables us to write complex, asynchronous code that's still readable. We do this by chaining multiple `.then()`'s and `.catch()`'s.
- To use promise composition correctly, we have to remember to `return` promises constructed within a `.then()`.
- We should chain multiple promises rather than nesting them.
- To take advantage of concurrency, we can use `Promise.all()`.