Async Javascript

IT TALENTS
Training Camp

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is lightweight text-data interchange format
- JSON is language independent *
- JSON is "self-describing" and easy to understand

- JSON uses JavaScript syntax for describing data objects, but JSON is still language and platform independent. JSON parsers and JSON libraries exists for many different programming languages.

Working with JSON

```
const text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

- Using the built-in function *JSON.parse()* we convert string into a JavaScript object:

```
const obj = JSON.parse(text);
```

- Using the built-in function *JSON.stringify()* we convert object into a string:

```
const str = JSON.stringify(obj);
```

What is AJAX?

- AJAX = Asynchronous JavaScript and XML.
- AJAX is a technique for creating fast and dynamic web pages.
- AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.
- Classic web pages, (which do not use AJAX) must reload the entire page if the content should change.
- Examples of applications using AJAX: Google Maps, Gmail, Youtube, and Facebook tabs.

How AJAX works?

- Ajax dos the following things:
  - Sends data to the sever
  - Receives data from the server
  - Both

- In general AJAX means sending data from the browser to the server and/or getting its response and delivering it to the browser without reloading the page.

How AJAX works?

1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

What is XMLHttpRequest?

- XMLHttpRequest (XHR) objects are used to interact with servers.
- You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.
- XMLHttpRequest is used heavily in AJAX programming.
- Despite its name, XMLHttpRequest can be used to retrieve any type of data, not just XML.

How to create one?

```
let xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 &&  xhr.status == 200) {
       document.getElementById("demo").innerHTML =  xhr.responseText;
    }
};

xhr.open("GET", “url");
xhr.send();
```

XMLHttpRequest synchronous requests

- You can create a synchronous XMLHttpRequest by passing a 3rd argument to the open function:

```
xhr.open("GET", “url“, false);
```

- Synchronous XMLHttpRequest (async = false) is not recommended because the JavaScript will stop executing until the server response is ready. If the server is busy or slow, the application will hang or stop.
- Synchronous XMLHttpRequest is in the process of being removed from the web standard, but this process can take many years.

Modify Request headers

- Using the *setRequestHeader()* method you can add additional request headers.
- *Each time you call setRequestHeader() after the first time you call it, the specified text is appended to the end of the existing header's content.*
- *Syntax:*

  xhr.setRequestHeader(header, value)

Using XHR for POST/PUT/PATCH requests

- The *send(body)* method accepts an optional parameter which lets you specify the request's body
- This is primarily used for requests such as POST or PUT. If the request method is GET or HEAD, the body parameter is ignored and the request body is set to null.
- The body needs to be serialized first using the *JSON.stringify(body)* method

Callback hell

- Callback Hell is achieved when unwieldy number of nested “if” statements or functions are being used.
- Callback Hell, also known as Pyramid of Doom, is an anti-pattern seen in code of asynchronous programming.

```
// First, we must get user by id
CallEndpoint("api/getidbyusername/hotcakes", function(result) {
    CallEndpoint("api/getfollowersbyid/" + result.userID, function(result) {
        CallEndpoint("api/someothercall/" + result.followers, function(result) {
            CallEndpoint("api/someothercall/" + result, function(result) {
                CallEndpoint("api/someothercall/" + result, function(result) {
                    // Ahhhhhhh... but you didn't believe you'd end up here
                });
            });
        });
    });
}); // Always use semicolons; at work and at home.
```

Promises

- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

- A JavaScript Promise object can be:
  - Pending
  - Fulfilled
  - Rejected

- The Promise object supports two properties: state and result.
- While a Promise object is "pending" (working), the result is undefined.
- When a Promise object is "fulfilled", the result is a value.
- When a Promise object is "rejected", the result is an error object.

Creating Promise

- Promises are created via the new Promise() constructor.

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        if (Math.random() > 0.5) {
            resolve('success');
        } else {
            reject('error');
        }
    }, 300);
});
```

Handling Promises

- In order to receive the value from a promise we need to use its predefined methods:
  - .then()
  - .catch()
  - .finally()

- A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called.

The Promise.then() Method

- The .then() method accepts 2 optional parameters – callback for success and callback for error.

```
myPromise.then(
    function (value) { /* code if successful */ },
    function (error) { /* code if some error */ }
);
```

- The then() function returns a new promise, different from the original. This means that the promises can be chained:

```
doSomething()
    .then(result => doSomethingElse(result))
    .then(newResult => doThirdThing(newResult));
```

The Promise.catch() Method

- The .catch() method accepts a callback for handling errors.
- It also returns new promise object.

```
myPromise
    .then(function (value) { /* code if successful */ })
    .catch(function (error) { /* code if some error */ });
```

The Promise.finally() Method

- When the promise is settled, i.e either fulfilled or rejected, the specified callback function is executed.
- This provides a way for code to be run whether the promise was fulfilled successfully or rejected once the Promise has been dealt with.
- It also returns new promise object.

```
myPromise
    .then(function (value) { /* code if successful */ })
    .catch(function (error) { /* code if some error */ })
    .finally(function () { /* code that will always be executed */
});
```

Useful Promise Methods

- Promise.all([promise1, promise2]) -> method returns a promise that *resolves* as soon as all promises are resolved OR will reject as soon as the FIRST promise reject.

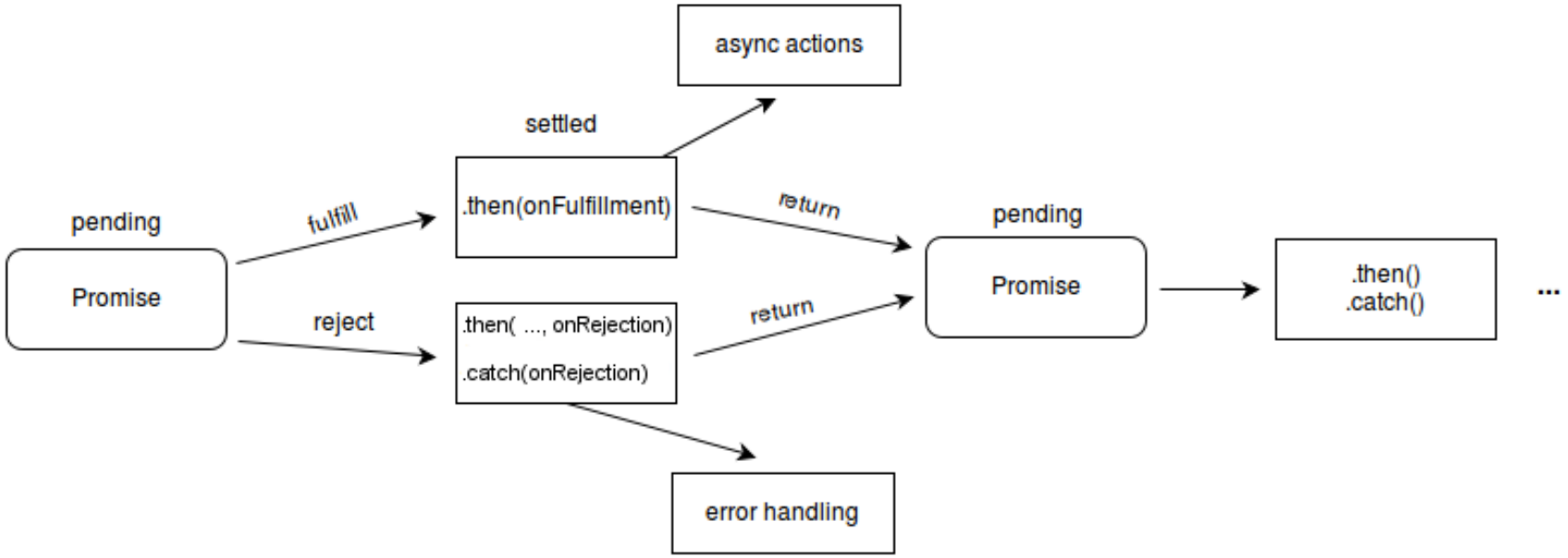
- Promise.race([promise1, promise2]) -> method returns a promise that fulfills or rejects as soon as one of the given promises that succeed or reject.

Guarantees

- Unlike old-fashioned passed-in callbacks, a promise comes with some guarantees:
- Callbacks added with then() will never be invoked before the completion of the current run of the JavaScript event loop.
- These callbacks will be invoked even if they were added after the success or failure of the asynchronous operation that the promise represents.
- Multiple callbacks may be added by calling then() several times. They will be invoked one after another, in the order in which they were inserted.

- One of the great things about using promises is chaining.

Promises



- Read this article: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- Exercises: https://www.codingame.com/playgrounds/347/javascript-promises-mastering-the-asynchronous/the-promise-class

IT TALENTS Training Camp

Fetch API

- The Fetch API interface allows web browser to make HTTP requests to web servers.
- The same purpose as XMLHttpRequest Object implemented in much simpler way.
- It works with Promises.

- Automatically returns a resolved or fulfilled Promise object.

Basic Usage

- A basic fetch request is really simple to set up. Have a look at the following code:

```
fetch('http://example.com/movies.json')
    .then(response => response.json())
    .then(data => console.log(data));
```

- Here we are fetching a JSON file across the network and printing it to the console. The simplest use of fetch() takes one argument — the path to the resource you want to fetch — and returns a promise containing the response

The Response Object – the ”hacky” part

- The response object is resolved as soon as all headers arrive.
- The response.json() method takes a Response stream and reads it to completion.
- It returns a promise that resolves to a JavaScript object. This object could be anything that can be represented by JSON — an object, an array, a string, a number...

```
fetch('http://example.com/movies.json')
    .then(response => response.json())
    .then(data => console.log(data));
```

Advanced usage

- The fetch() method can optionally accept a second parameter, an init object that allows you to control a number of different settings:
  - method: “POST” // “PUT”, “PATCH”, “DELETE”, etc
  - body: JSON.stringify({}) // The request body
  - mode: “cors” // The mode for the request, e.g., cors, no-cors, or same-origin.
  - redirect: “error” // How to handle a redirect response:
    - follow: Automatically follow redirects. Unless otherwise stated the redirect mode is set to follow
    - error: Abort with an error if a redirect occurs.
    - manual: Caller intends to process the response in another context.

Summary

- Working with JSONs
- AJAX
- XMLHTTPRequest
- Synchronous vs asynchronous requests
- Promises
  - Handle promises
  - Promise chaining
  - Promise methods -> all(), race()
- Async/Await