**Asynchronous Data Transfer and the Fetch API**

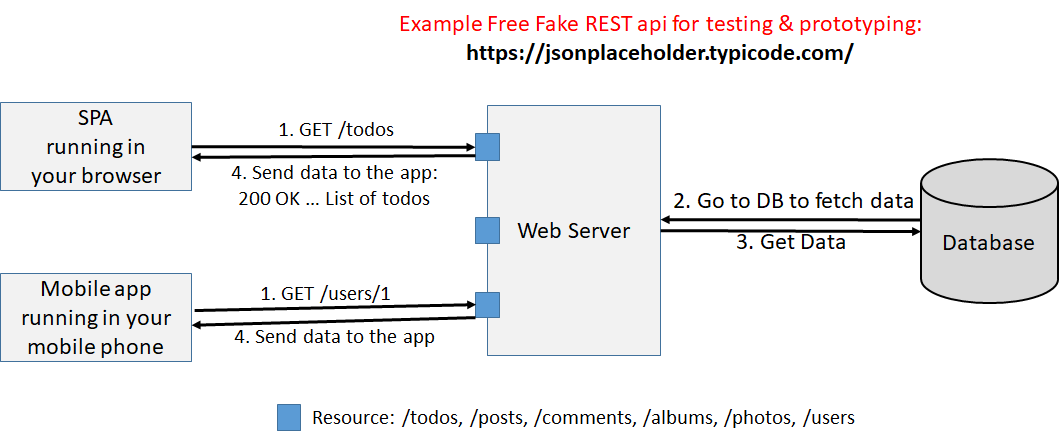
(Note: The discussion here benefits from:

1. Net Ninja: Asynchronous JS tutorial: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9jx2TTZk3IGWKSbtugYdrlu>,
2. Traversy Media: Async JS Crash Course - Callbacks, Promises, Async Await:

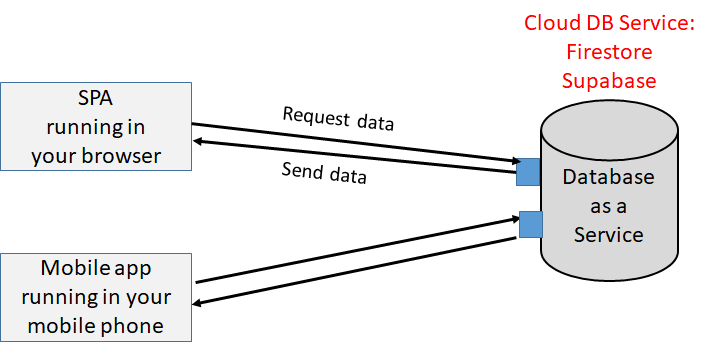
<https://www.youtube.com/watch?v=PoRJizFvM7s>)

In our single page applications (SPA) that we designed with vanilla JS, the data rendered on the page resided solely inside our application. We stored the application data in some global state (some JS object) and have all Components use it. This all works without any problems, but when the application terminates, i.e., when you close the page or the browser, the data gets lost! You can store your data into the browser’s local storage and retrieve it from there the next time your application starts, but this is NOT a preferred method as the data is stored in the local storage of the browser, which means that it can gets deleted when accidently purged from the local storage and worse yet, it is only available on that machine. If you want to access that data from any other machine, it is NOT possible. What you need to do is to store this data into a remote storage on the Internet and access it from within your application through real-time data transfers. There are two models here:

1. RESTful services (<https://restfulapi.net/>): REST is an acronym for REpresentational State Transfer, and is an architectural style for distributed hypermedia systems. The idea behind RESTful services is the following: You create a backend service that runs in a Web server and exports API endpoints called resources identified by URIs (Universal Resource Identifiers) that serves data to the clients. The actual data is stored in a backend database (e.g., MySQL, MongoDB, etc.). The client applications, e.g., SPAs running in Web browsers, Mobile apps running in mobile phones, and even desktop apps, send requests to the exported endpoints. The Web server receives these requests, talks to the database to fetch the requested data, and sends them back to the client.



1. Serverless Architectures (<https://martinfowler.com/articles/serverless.html>): In this design there is no Web server that sits between your client application and the database. The database is on the cloud and your app directly accesses the database through the API calls provided by the database server. The database can either be a SQL database (MySQL, PostgreSQL, Oracle, Microsoft SQL server, …) or a no-SQL database such as MongoDB, CouchDB, etc. The examples of such services are Firebase by Google (<https://firebase.google.com/>) or Supabase (<https://supabase.com/>), which is an open-source Firebase alternative. Here is the serverless architectural model:



Here is Net Ninja’s tutorial on Firebase:

<https://www.youtube.com/playlist?list=PL4cUxeGkcC9itfjle0ji1xOZ2cjRGY_WB>

<https://www.youtube.com/playlist?list=PL4cUxeGkcC9jERUGvbudErNCeSZHWUVlb>

Here is Net Ninja’s tutorial on Supabase:

<https://www.youtube.com/playlist?list=PL4cUxeGkcC9hUb6sHthUEwG7r9VDPBMKO>

**Rest APIs and CRUD applications**

In this chapter we will be looking at accessing data through RESTful services as illustrated in the first model. The serverless architecture is beyond the scope of this course. But you are encouraged to go through the tutorial listed in the previous section.

As illustrated in the figure above, in this model we have a Web Server exporting several API endpoints (resources) to serve data to the clients. The clients can be SPAs running in Web browsers, mobile applications running in mobile devices, or even desktop applications. The actual data is stored persistently in a backend database. The database can be either an SQL database or a no-SQL database. Here is the interaction between a client and a RESTful API:

1. The client sends an HTTP request to an endpoint asking for a resource, e.g., /todos, /users/1, etc.
2. The Web server receives the request, and then talks to the backend database to fetch/store the requested data
3. The database executes the query, receives the data and sends it to the Web server
4. The Web server packs the data into a HTTP response message and sends it back to the client.

There are many free RESTful APIs on the Web that you can make use of. <https://jsonplaceholder.typicode.com/> is onesuch freeFake REST API for testing & prototyping that you can use in your apps. Another is <https://dummyjson.com/>. <https://rapidapi.com/> gives a list of many free public APIs that you can use in your apps.

While we only talked about “Reading” data from an API, it is usually the case we will Create new Records, Update and Delete existing records in our applications. Such applications, therefore, are called CRUD (Create, Read, Update, Delete) apps (<https://www.codecademy.com/article/what-is-crud>). Each operation corresponds to a different HTTP request as we will see later in this chapter (Create: POST, Read: GET, Update: PUT/PATCH, Delete: DELETE).

**Sending/Receiving Formatted Data (XML vs JSON)**

When a client and a server exchange data, the first question that must be answered is the format in which the data exchange occurs. Obviously, the data exchanged in this transmission must be independent of the machine architecture and the database being used. That is, we must agree on a universally recognized data format so that this exchange works seamlessly.

There are in general two ways to format the data: XML (e**X**tensible **M**arkup **L**anguage) or JSON (Java Script Object Notation). XML is kind of the old way of formatting data, and JSON is the new way to format the data and has largely become the de-facto method as it can easily be converted to JS objects and manipulated in JS. But you can still encounter XML formatted data especially in legacy systems, so it is important that you know about it.

XML was designed to be both human- and machine-readable data format to store and transport data. As an example, assume that you have a collection of CDs with certain attributes that you want to store and transfer. Assume that a CD has the following attributes:

|  |
| --- |
| CD(title: string, artist: string, country: string, company: string, price: number, year: number) |

Here is a specific CD:

|  |
| --- |
| CD1 = (“Empire Burlesque”, “Bob Dylan”, “USA”, “Columbia”, 10.90, 1985); |

Here is how we represent this CD using the XML and JSON notation. As you can see, XML specifies each attribute of the CD between an opening and closing tags with the tag name specifying the name of the attribute. JSON, on the other hand, uses a notation that is very similar to JS object notation, which is a more efficient way of representing the same data. To learn more about XML, please refer to [<https://www.w3schools.com/xml/default.asp>]. To learn more about JSON, please refer to [<https://www.w3schools.com/js/js_json_intro.asp>].

|  |  |
| --- | --- |
| **XML** | **JSON** |
| <CD>  <TITLE>Empire Burlesque</TITLE>  <ARTIST>Bob Dylan</ARTIST>  <COUNTRY>USA</COUNTRY>  <COMPANY>Columbia</COMPANY>  <PRICE>10.90</PRICE>  <YEAR>1985</YEAR>  </CD> | {  "title": "Empire Burlesque",  "artist": "Bob Dylan",  "country": "USA",  "company": "Columbia",  "price": 10.90,  "year": 1985  } |

Look at 07-Ajax/01-XML/CD.xml, 07-Ajax/01-XML/CDs.xml

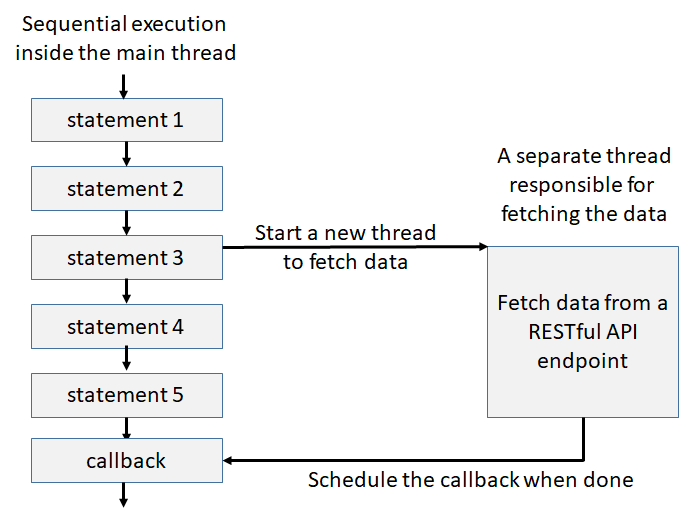
Look at 07-Ajax/02-JSON/CD.json, 07-Ajax/02-JSON/CDs.json

Look at <https://jsonplaceholder.typicode.com/todos> for other JSON formatted data. Another good place to look at is <https://dummyjson.com/>

Asynchronously doing data transfer between a Web client and a Web server used to be called AJAX, which is an acronym for **A**synchronous **J**avascript **A**nd **X**ML. In the early days of Web programming and before the invention of JSON, XML was the primary choice for formatted data. That’s why this operation was called AJAX (<https://www.w3schools.com/js/js_ajax_intro.asp>). The term was coined by Jesse James Garret in 2005 in his paper titled “Ajax: A New Approach to Web Applications”. For more information, please refer to the MDN official documentation (<https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX>). Below, we will cover the technologies surrounding this technique.

**Handling Asynchronous Events in JS (Callback function processing)**

Before we go into the details of making a HTTP request from within JS, let’s first talk about how JS handles asynchronous events. JS is a single-threaded, asynchronous PL. It is single threaded in the sense that it has a single main thread (called the event loop) that executes what is to be executed sequentially one after the other. This main thread executes everything changing the DOM and rendering the page on the browser window. JS is asynchronous in the sense that you can schedule a task in a separate thread that gets executed asynchronously and invokes a call-back function when it finishes executing. This is how the fetch API works and is illustrated in the following figure:



Assume that you want to fetch some data from a remote Web server (a RESTful API endpoint). There are two alternatives: You can do this synchronously inside the main thread, which would block the browser until the fetch operation is done. If this operation takes several seconds, the browser window will be unresponsive during this time, which is definitely not what you want. Alternatively you can schedule the fetch operation in a separate thread and let the fetch run asynchronously in parallel as shown above. The main thread then continues executing the next statement. When the parallel thread executing the fetch operation is done, it notifies the main thread by scheduling a callback function. The main thread then executes this callback function. Notice that in this model the main thread never gets blocked waiting for the fetch operation to complete. Both threads run in parallel asynchronously.

We have already seen a version of this asynchronous operation when we used the setTimeout function to schedule a callback after some delay as given in 07-Ajax/03-SetTimeout/index1.html

|  |
| --- |
| <script>  console.log(1);  console.log(2);  // This will schedule the callback after 1 second  setTimeout(()=>console.log("callback function fired"), 1000);  console.log(3);  console.log(4);  </script> |

Here, the main thread starts with the first statement (console.log(1)) and continues executing the statements sequentially. When it executes the setTimeout function, this creates a timeout event to fire in 1 seconds. The main thread then continues executing the last two statements and enters the event loop. After 1 second, the timer expires, which schedules the callback in the main thread’s macro task queue. Eventually, the main thread takes notice of the callback function and executes it printing “callback function fired” on the console. As you can see, we have two separate threads executing in parallel. The main thread executes the event loop and the timer thread takes care of the timeout events in parallel.

**Fetching Data from a Web Server with JS**

At the heart of asynchronous data fetch with JS lies the XMLHttpRequest object. All modern browsers (Chrome, Firefox, IE7+, Edge, Safari, Opera) have a built-in XMLHttpRequest object. When you want to send a request to a Web server and get a response asynchronously, you first create an XMLHttpRequest, send the request to the server over this object, and then get the response from the server. Once the response is received, the XMLHttpRequest notifies the main thread by a callback function, which executes the callback and usually updates the page with the new data. Notice that the main thread and the data fetch thread run asynchronously in parallel, which means that during the fetch operation your Web page continues to run without interruption because *the communication occurs asynchronously in the background*, which does not block the current Web page. In the following example we implement a GET function that takes the object to fetch and logs it the console when the operation is successfully completed:

|  |
| --- |
| function GET(resource){  let xhr = new XMLHttpRequest();  // Install the handler that will be called when the response has arrived from the HTTP server  xhr.onload = function (){  if (xhr.status == 200) {  console.log(xhr.responseText)    } else if (this.status >= 400){  console.log(`Get failed. ${xhr.status}`);  } //end-if  } //end-onload  // This error handler gets called when a network error occurs  xhr.onerror = (err) => {  console.log("Error occurred: ", err);  } //end-onerror  // Initialize the xhr object.  // method: the type of request: GET, POST, PUT, DELETE, PATCH  // url: resource  // async: true (asynchronous) or false (synchronous)  xhr.open("GET", resource, true);    // Send the HTTP request. When the response arrives, onload will be called  xhr.send();  } //end-GET  // Get this resource  console.log(1)  console.log(2)  // Get this resource asynchronously  GET('https://jsonplaceholder.typicode.com/todos/1')  // GET('http://127.0.0.1:5500/file.txt') // Get the data from a local Web server  console.log(3)  console.log(4)  </script> |

When you look at the above code you see that we first create an XMLHttpRequest object and set up the “onload” method, which gets called when the request completes. Inside the “onload” function, we check the HTTP response code (this.status). When the request completes successfully, the response code is 200. So, when we get a 200 response, we simply log the “xhr.responseText” to the console.

We still need to send the request to the server. To do this, we first initialize the “xhr” objected using the open method, which takes the request to be sent (GET/POST/PUT/PATCH/DELETE etc.), the URI of the resource to fetch, and whether the request should execute synchronously or asynchronously. Unless absolutely necessary, you should make the last parameter true so that the request gets executed asynchronously. Finally, we call “xhr.send()” method, which sends the request to the Web server, gets the response, and then calls the “onload” method after the request is complete.

Look at 07-Ajax/04-XMLHttpRequest/index1.html

**XMLHttpRequest States**

After you create the “xhr” object, it goes through several states before the request is complete.

|  |  |
| --- | --- |
| xhr.readyState | 0: request not initialized  1: server connection established  2: request received  3: processing request  4: request finished and response is ready |

The readyState is 0 when the “xhr” object is created. When the connection to the server is established, the readyState becomes 1. It then goes through states 2, 3 and finally becomes 4 when the request is totally finished and the response is ready. If you want to see all these states and handle the response that way, there is another callback function inside the “xhr” object that you can install and use. It is the “onreadystatechange” method. To check if the request has really completed, you must now check if xhr.readyState is 4 and that xhr.status==200:

|  |
| --- |
| function ButtonClickHandler(e){  let xhr = new XMLHttpRequest();  // Install the handler that will be called when the response has arrived from the HTTP server  xhr.onreadystatechange = function (){  if (this.readyState == 4 && this.status == 200) {  document.getElementById("para1").innerHTML = this.responseText;  } //end-if  } //end-onload  // Initialize the xhr object.  // method: the type of request: GET or POST  // url: the server (file) location  // async: true (asynchronous) or false (synchronous)  xhr.open("GET", "file.html", true); // Load the page from the local Web server    // Send the HTTP request. When the response arrives, onreadystatechange will be called  xhr.send();  } //end-ButtonClickHandler |

Look at 07-Ajax/04-XMLHttpRequest/index2.html

**Accessing HTTP Headers**

It is also possible to set a HTTP request’s header using setRequestHeader(header, value) method. Here is an example where we send a POST request to an ASP page that contains the form element values. Recall that if this was a GET request, the form element values would have been put directly inside the URL as a list of key=value pairs. Since this is a POST request though, we have to put the key=value pairs inside the body of the message, which is more secure. But now we have to specify the content type in the body of the message, which is specified by setting the HTTP “Content-type” header. Since we are sending form elements as key=value pairs, we set the Content-type to "application/x-www-form-urlencoded".

|  |
| --- |
| xhr.open("POST", "ajax\_test.asp", true);  xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  xhr.send("fname=Henry&lname=Ford"); |

We can also access headers in the response using the “xhr” object’s getResponseHeader method. Here is an example where we access the “Last-Modified” header of the HTTP response. It is also possible to get ALL HTTP headers from the response using xhr.getAllResponseHeaders().

|  |
| --- |
| xhr.getResponseHeader("Last-Modified"); |

**Sending/Receiving Formatted Data (XML vs JSON)**

At the beginning of this chapter we said that the two most common data formats to send/receive formatted data between a Web client and a Web server are XML and JSON, the latter being the most modern format as it can easily be converted to/from a JSON object.

In the following example, we will get a CD in XML format from our local Web server and display it on the Web page. Notice that when you receive the XML data, there is a special “xhr” property that you can use to access it: xhr.responseXML. While you can access the entire received data as a string using xhr.responseText, xhr.responseXML gives you the XML formatted data as a XML DOM for easy access.

Look at 07-Ajax/01-XML/index.html

In the following example, we will get a CD in JSON format from our local Web server and display it on the Web page.

Look at 07-Ajax/02-JSON/index1.html

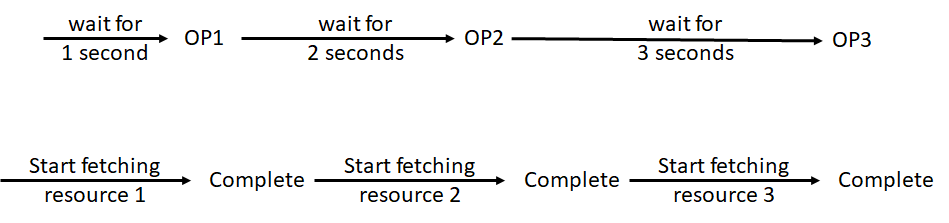
In the above example we use the JSON.parse method to convert from a JSON object to a JS object. If we want to do the reverse conversion, i.e., convert a JS object to a JSON object, so that we can transmit it to the Web server using a POST request, then we need to use JSON.stringify() method. This is illustrated in the following example, where we make a GET and a POST request to a fake REST API server, i.e., <https://jsonplaceholder.typicode.com/>.

Look at 07-Ajax/02-JSON/index2.html

Discussion on CORS: <https://zinoui.com/blog/cross-domain-ajax-request>

**CallBack Hell**

Consider a hypothetical application, where we want to perform an operation, say OP1, after 1 second. When that operation completes, we then want to wait for another 2 seconds and perform another operation OP2. Finally, when that operation completes, we want to wait for another 3 seconds and perform a final operation, say OP3. Although this is a very hypothetical example, this comes up in practice with HTTP requests: Say you want to fetch some data from a Web server. When that request completes, you then want to fetch another data. Finally, when this request completes, you want to fetch some other data. So we have 3 data to be fetched, one after the other. These are illustrated in the following figure:



We can easily implement the above logic using setTimeout and call-back functions as follows:

Look at: 07-Ajax/05-CallbackHell/index1.html

|  |
| --- |
| let now = new Date();  // Set-up our call-back functions to perform OP1, OP2 and OP3 in sequence  setTimeout(()=>{  console.log(`Performing OP1. Time passed: ${new Date() - now}.`);  setTimeout(()=>{  console.log(`Performing OP2. Time passed: ${new Date() - now}.`);  setTimeout(()=>{  console.log(`Performing OP3. Time passed: ${new Date() - now}.`);  }, 3000);  }, 2000);  }, 1000);  console.log('Main thread is here and is entering the Event Loop'); |

For the data fetch application, assume that we extend the prototype of our GET function as follows:

|  |
| --- |
| GET(url, callbackForSuccess, callbackForFailure); |

This GET function now takes the URL of the resource to fetch, and two callback functions: One function to be called if the operation is successful, and one function to be called if the operation fails, e.g., the Web server is down, there is a network problem, the resource does not exist, etc. Using this method, we can implement our fetch application as follows:

|  |
| --- |
| GET(url1, (data)=>{  console.log(`${url1} is successfully fetched. Here is the data: ${data}`);  GET(url2, (data)=>{  console.log(`${url2} is successfully fetched. Here is the data: ${data}`);  GET(url3, (data)=>{  console.log(`${url3} is successfully fetched. Here is the data: ${data}`);  }, (err)=>{  console.log(`Error fetching ${url3}: `, err);  });  }, (err)=>{  console.log(`Error fetching ${url2}: `, err);  });  }, (err)=>{  console.log(`Error fetching ${url1}: `, err);  }); |

Here, the main thread initiates the GET operation for “url1”, which runs asynchronously in another thread. After initiating the request, the main thread continues with the Event loop. When the GET operation completes, either successfully or with failure, the appropriate call-back function is invoked. If the operation is successful, then we initiate the next GET operation for “url2”. When that completes we finally initiate GET for “url3”. Notice that the implementation of this application is very similar to our previous application with both having the same structure.

Notice from above that we have a nested callback (a triangle) structure in both applications. As the number of nested callbacks gets bigger, this will get more unreadable! This is called the callback hell!

Look at: 07-Ajax/05-CallbackHell/index2.html

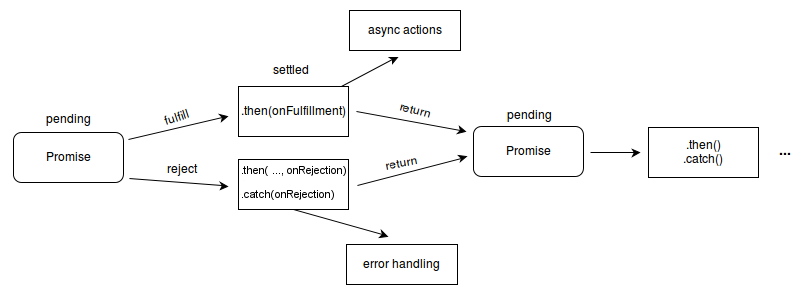
**Promises [**<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>**]**

To solve the callback hell problem, ES6 introduced a new construct called a **Promise**. A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value. The idea is very simple: You create a Promise and initialize it with two callback functions: One that is called if the Promise completes successfully (this is called holding or resolving the promise), and another that is called if the Promise fails (this is called rejecting the promise). To handle the case where the promise is resolved, you write .then clause, and to handle the case where the promise is rejected, you write a .catch clause. Here is an example:

|  |
| --- |
| Let p = new Promise((resolve, reject)=>{…});  p.then((res)=>{console.log(res);}).catch((err)=>{console.log(err);}); |

A Promise is in one of these states:

* **pending**: initial state, neither fulfilled nor rejected.
* **fulfilled**: meaning that the operation completed successfully.
* **rejected**: meaning that the operation failed.



Here is an example, where we create two promises, one of which resolves and the other fails.

Look at: 07-Ajax/06-Promise/index1.html

Here is a more reasonable example: Assume that we would like to write an assertion function that succeeds if the assertion is true, and fails if the assertion is false. Here is how we can implement this using a Promise:

|  |
| --- |
| function CheckIfLessThan5(val){  return new Promise((resolve, reject)=>{  if (val < 5) resolve(`Succeeded. ${val} is < 5`);  else reject(`Failed. ${val} is >= 5`);  });  } //end-CheckIfLessThan5  CheckIfLessThan5(3).then((ret)=>{console.log(ret);}).catch((err)=>{console.log(err);}); |

Look at: 07-Ajax/06-Promise/index2.html

Now, let’s get back to our original application, where we performed three operations one after the other. We can now implement this using a Promise as shown in the following example. We will re-implement our fetch data application using a Promise in the chapter when we cover AJAX.

Look at: 07-Ajax/06-Promise/index3.html

In the previous application, we have three promises that run sequentially one after the other. What if we want to run all these promises in parallel, and get notified when ALL of these promises either get fulfilled or rejected? This can be done using Promise.all as follows:

Look at: 07-Ajax/06-Promise/index4.html

The problem with Promise.all is that even if one of the promises fails, then Promise.all fails for all promises. It is ALL or NOTHING. This is illustrated in the following example:

Look at: 07-Ajax/06-Promise/index5.html

**async/await**

ES8 took asynchronous programming with Promises one step further and introduced two keywords “async” and “await” to further simplify writing asynchronous code. Instead of using “.then” to handle the resolution of a Promise, we simply call the promise with “await” and get the result. We then process the result and if necessary make further asynchronous calls with Promises. In order to catch the errors returned by Promises, we enclose the Promise calls inside a “try/catch” block. The error is then caught inside the “catch” block. To use the await keyword, the function within which it is used must be labeled as an “async” function. Here is an example:

Look at: 07-Ajax/06-Promise/index6.html

**Promise-based XMLHttpRequest handling**

Now that we know that Promises help us avoid the callback hell problem, we will implement a Promise-based GET function to make asynchronous calls to a Web server. We will implement a GET function that returns a Promise, which will resolve when the request completes successfully, and reject when the request fails. We will then use .then and .catch methods of the Promise to handle the return values. Here is how GET is implemented using a Promise:

|  |
| --- |
| function GET(url){  return new Promise((resolve, reject)=>{  let xhr = new XMLHttpRequest();  xhr.onload = function (){  if (this.status == 200) {  // Convert the JSON object to JS object  const res = JSON.parse(this.responseText);  resolve(res);  } else if (this.status >= 400){  reject(this.status);  } //end-else  } //end-onload  xhr.onerror = function(ev){  reject("GET Error!");  } //end-onerror  // Initialize the xhr object and send the request  xhr.open("GET", url, true);  xhr.send();  });  } //end-GET |

Now, we use this GET function to fetch a JSON object from a Web server, and access the returned JSON object in our program using .then, and catch the errors using .catch as follows:

|  |
| --- |
| GET("cd.json")  .then((cd)=>{  console.log(cd);  }).catch(err=>console.log(err)); |

Here is the entire application:

Look at 07-Ajax/07-PromiseBasedXHR/index1.html

We now return to the application where we performed 3 GET operations in sequence. Using our GET function with Promises, it is very easy to implement. In the next example, we use our GET function to fetch “cd.json”, then “cds.json” and finally “person.json” one after the other, and chain the handling of our call-backs using “.then”:

Look at 07-Ajax/07-PromiseBasedXHR/index2.html

**Fetch API [**<https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>**]**

The Promise-based API to **XMLHttpRequest** object that we described in the previous section has already been incorporated into JS. It is called the Fetch API (<https://flaviocopes.com/fetch-api/>), and provides a more powerful and flexible feature set.

For making a request and fetching a resource, use the fetch() method, which is part of the window object. This means that you may use “fetch” directly without specifying the containing object (just like document, which is also part of the window object). The fetch() method takes one mandatory argument, the path to the resource you want to fetch. It returns a **Promise** that resolves to the **Response** to that request, whether it is successful or not. You can also optionally pass an init options object as the second argument (see Request).

To use the fetch API, simply pass the resource you want to fetch, and then use the returned Promise to check whether the result was successful:

|  |
| --- |
| fetch("file.html")  .then(function(res){  console.log(res); // This is a Response object  return res.text(); // Return another promise that contains the actual data  }).then(function(data){  console.log(data);  document.getElementById("para1").innerHTML = data;  }).catch(function(err){  console.log("Fetch error: " + err);  }); |

Look at 07-Ajax/08-Fetch/index1.html

Alternatively, you can use arrow functions to achieve the same thing:

|  |
| --- |
| fetch("file.html")  .then((res)=>res.text())  .then((data)=>{document.getElementById("para1").innerHTML = data;}  .catch((err)=>{  console.log("Fetch error: " + err);  }); |

Look at 07-Ajax/08-Fetch/index2.html

For our previous app that fetches a CD or an array of CD objects from the local Web server:

Look at 07-Ajax/08-Fetch/index3.html

For our previous app that fetches a CD, an array of CD objects, and a Person from the local Web server one after the other:

Look at 07-Ajax/08-Fetch/index4.html

For an app that sends GET, POST, PUT, PATCH and DELETE requests to a remote server:

Look at 07-Ajax/08-Fetch/index5.html

**Axios Library [**<https://github.com/axios/axios>**]**

Axios is a Promise-based HTTP client for the browser and node.js, and can be used as an alternative to the fetch API. It is very similar to the fetch API, but has some better features that may come handy.

To use Axios, you can either download it or include a CDN as follows:

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.19.0/axios.min.js"></script> |

For an implementation of the app (07-Ajax/09-Axios/index4.html) that we did with the Fetch API using Axios:

Look at 07-Ajax/09-Axios/index.html

**JSON Server [**https://www.npmjs.com/package/json-server**]**

[Net Ninja’s JSON Server tutorial: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9i2v2ZqJgydXIcRq_ZizIdD>]

[<https://medium.com/codingthesmartway-com-blog/create-a-rest-api-with-json-server-36da8680136d>]

A common task for front-end developers is to simulate a backend REST service to deliver some data in JSON format to the front-end application and make sure everything is working as expected. Of course you can setup a full backend server, e.g. by using Node.js, Express, and MongoDB. However this takes some time and a much simpler approach can help to speed up front-end development time. JSON Server is a simple project that helps you to setup a REST API with CRUD operations very fast. The project website can be found at <https://github.com/typicode/json-server>.

Before you can use json-server, you need to install node and then globally install json-server as follows:

|  |
| --- |
| % npm install -g json-server |

The next step is to create a database of records in JSON format. Let’s call the name of the file db.json and copy the first 10 todos from <https://jsonplaceholder.typicode.com/todos>. Look at 07-Ajax/10-JSON-Server/db.json for details.

To run json-server, open up a terminal and type:

|  |
| --- |
| % json-server --watch db.json |

By default, json-server runs at port 3000. If you want your server to run at a different port, specify the port number from the command line as follows:

|  |
| --- |
| % json-server --watch db.json --port 8000 |

After the json-server starts running, we have following endpoints:

|  |
| --- |
| GET /todos  GET /todos /{id}  POST /todos  PUT /todos /{id}  PATCH /todos /{id}  DELETE /todos /{id} |

We can either use post-man or visual studio code Thunder Client extension to talk to our endpoints. After installing Thunder Client extension, its logo will appear on the left sidebar. Click on the icon and then New Request. Here, you can perform all HTTP requests that you used to perform with postman.

We can also talk to these endpoints from within our JS programs. Here is a re-implementation of the previous axios example that talks to our JSON-server running at port 8000.

Look at 07-Ajax/10-JSON-Server/index.html

For an implementation of our ItemListApp implemented in vanilla JS that uses the JSON-server to Create and Delete Items. Before running this app, make sure that you start json-server at the default port 3000 with data/db.json as the database.

Look at 07-Ajax/11-MyItemListApp

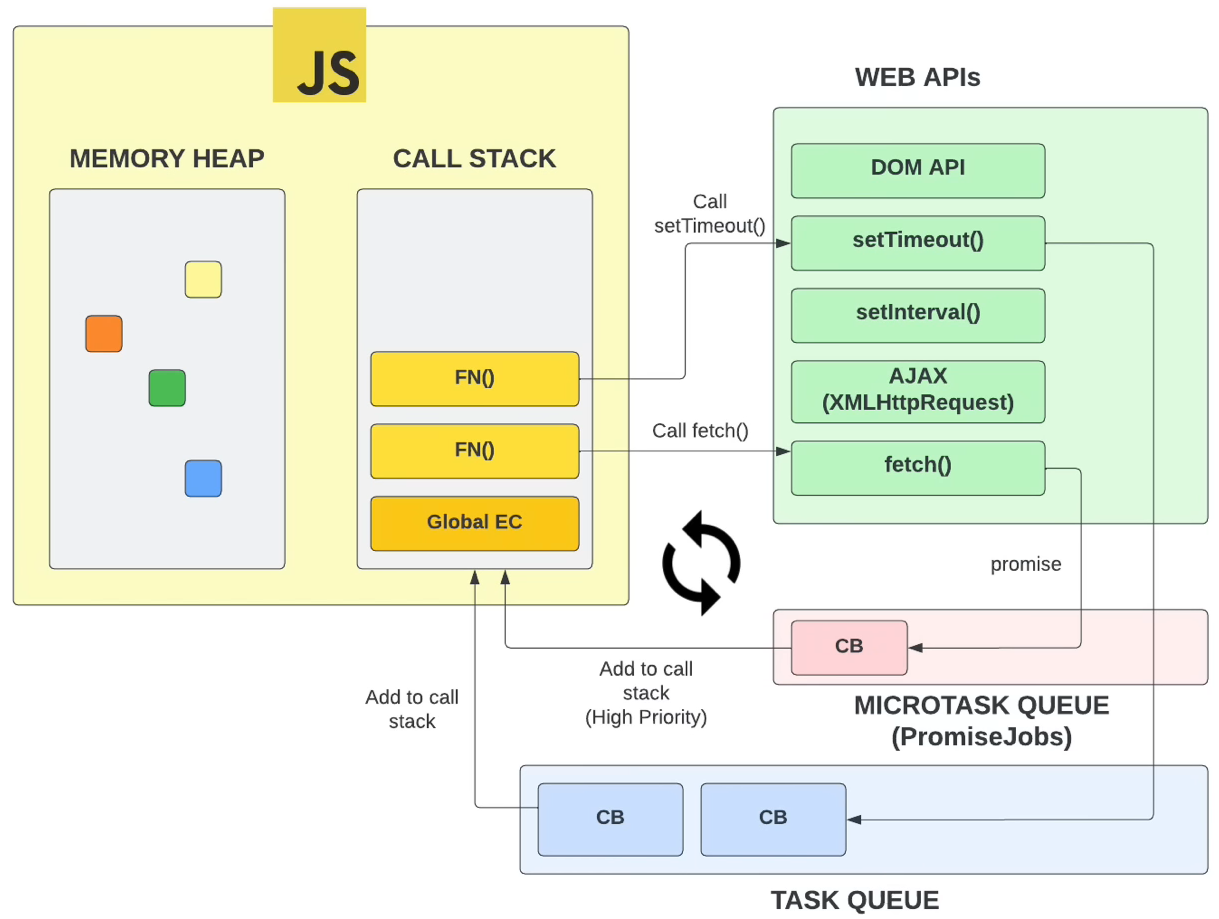
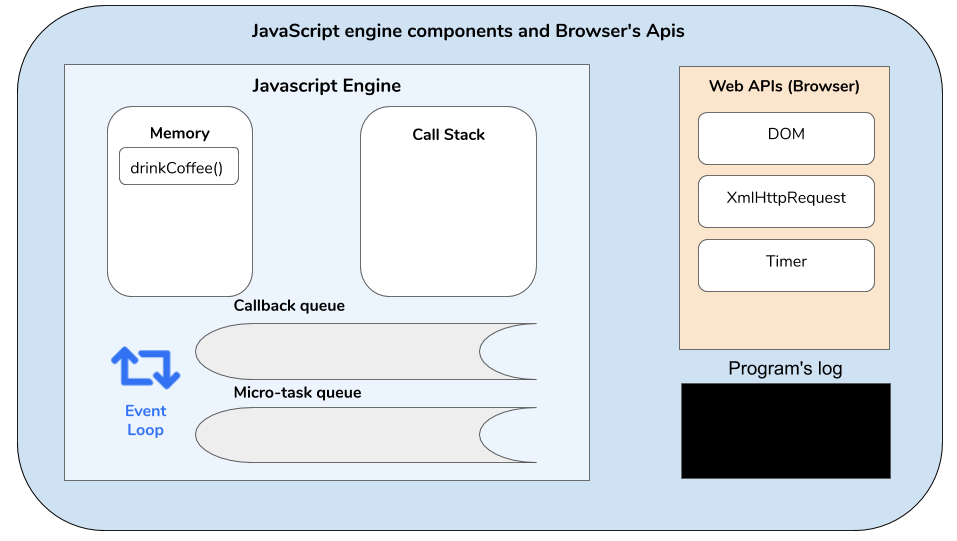
**The JS Event Loop [**<https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>**]: How JS actually works**

(The event loop: <https://www.youtube.com/watch?v=cCOL7MC4Pl0> by Jake Archibald)

(What the heck is the event loop anyway? <https://www.youtube.com/watch?v=8aGhZQkoFbQ> by Philip Roberts)

(JavaScript: Under The Hood: <https://www.youtube.com/playlist?list=PLillGF-Rfqbars4vKNtpcWVDUpVOVTlgB> by Traversy Media)

(The JavaScript Event Loop: Explained: <https://towardsdev.com/event-loop-in-javascript-672c07618dc9> )

The figures above taken from <https://www.youtube.com/watch?v=28AXSTCpsyU> and https://medium.com/better-programming/how-does-the-browser-execute-asynchronous-code-3808dff05ca4 respectively show different components of the JS engine and the event loop, and the Web APIs provided by the Browser. The JS engine has a **heap** which keeps track of the space used by different objects in your app, a **stack** that maintains a stack of execution frames for the currently executing functions, and two task queues: A **macro-task** queue that contains functions (callbacks) to be executed. These macro-tasks are such events as user-registered callback functions for such events as button clicks, mouse movements, timeout events. It also has a **micro-task** queue that also contains functions to be executed. But these are callbacks for Promises and have more priority over macro tasks. That is, if there are tasks in macro and micro task queues at the same time, the JS engine will always choose a task from the micro-task queue. It is only after all tasks in the micro-task queue is finished that the JS engine will start executing the tasks in the macro-task queue.

We already mentioned that JS is a single-threaded, asynchronous PL. The single main thread is what executes the code inside <script> tags after the page is loaded. When this main thread reaches the end of the script, it enters into what is called the “event loop”, where it waits for the next message (task) to arrive and then executes it. The event loop got its name because of how it's usually implemented, which usually resembles:

|  |
| --- |
| while (queue.waitForMessage()) {  queue.processNextMessage()  } |

queue.waitForMessage() waits synchronously for a message to arrive (if one is not already available and waiting to be handled). Each message is processed completely before any other message is processed. This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be pre-empted and will run entirely before any other code runs (and can modify data the function manipulates). This differs from C, for instance, where if a function runs in a thread, it may be stopped at any point by the runtime system to run some other code in another thread.

A downside of this model is that if a message takes too long to complete, the web application will be unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short, and if possible, cut down one message into several messages.

In web browsers, messages are added anytime an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message—likewise with any other event.

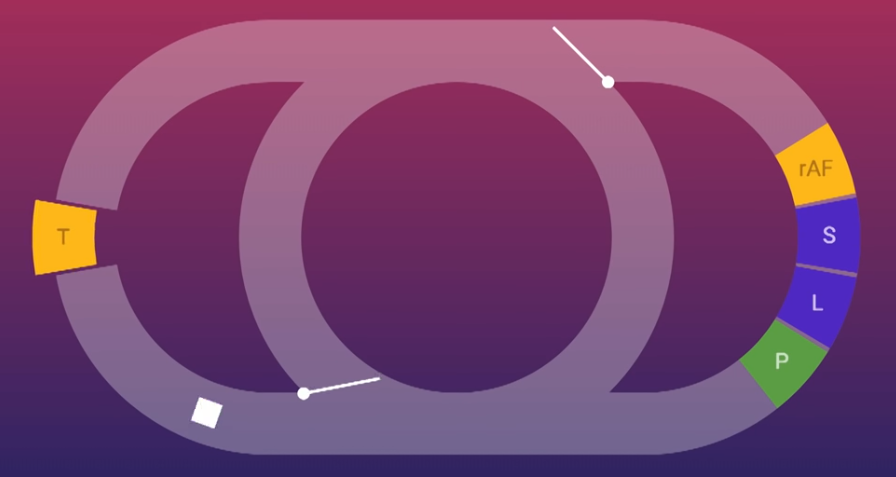
In the following example, the function setTimeout is called with 2 arguments: a message to add to the queue, and a time value (optional; defaults to 0). The time value represents the (minimum) delay after which the message will actually be pushed into the queue. If there is no other message in the queue, and the stack is empty, the message is processed right after the delay. However, if there are messages, the setTimeout message will have to wait for other messages to be processed. For this reason, the second argument indicates a minimum time—not a guaranteed time.

Here is an example that demonstrates this concept (setTimeout does not run immediately after its timer expires):

|  |
| --- |
| const s = new Date().getSeconds();  setTimeout(function() {  // prints out "2", meaning that the callback is not called immediately after 500 milliseconds.  console.log("Ran after " + (new Date().getSeconds() - s) + " seconds");  }, 500)  while (true) {  if (new Date().getSeconds() - s >= 2) {  console.log("Good, looped for 2 seconds")  break;  }  } |

Look at: 07-Ajax/12-EventLoop/index1.html

In fact, things are more complicated than this: The event loop must also update the browser screen (the DOM) if necessary. The browser screen update frequency is usually 60 frames/sec. Therefore, every 16.66 ms, the browser screen is re-rendered. For a detailed talk on the JS event loop, and how it works, listen to the following great talk by Jake Archibald: <https://www.youtube.com/watch?v=cCOL7MC4Pl0>. The following figure, taken from this talk, illustrates the event loop:



If I should just summarize this figure, the main event loop is the middle circle. When you load a page for the first time, the main loop executes your initial JS script until the end synchronously in this main event loop. Then every 16.66 ms, the event loop is supposed to go through the path to the right, and execute **ALL** requestAnimationFrame (rAF) tasks seen as the yellow rectangle on the right. It then re-renders the page by doing styling computation (CSS-S), then layout computation (L), and finally page generation (P); that is, the actual painting of the page.

When the main event loop is waiting in the main event loop (the middle circle) and if a task is added to the task queue, i.e., a message with a call-back function is added to the task queue, it then goes through the left loop, takes ONE task from the task queue (the yellow rectangle on the left) and executes it. It then comes back to the main loop. If it is now time to re-render the page, then it goes through the right-loop and re-renders the page. Having gone through the main loop, if there are more tasks in the task queue, the main loop again goes through the left loop and executes another user task. This continues forever. For another very explanatory talk on the JS Event loop, listen to the following talk by Philip Roberts: <https://www.youtube.com/watch?v=8aGhZQkoFbQ>. The speaker gives many examples to show how the event loop works to execute different macro and micro-tasks.

Look at: 07-Ajax/12-EventLoop/index2.html