Web Advanced: Javascript

“We will learn JavaScript properly. Then, we will learn useful design patterns. Then we will pick up useful tools to understand the modern world of coding.”

FALL 2022

SESSION #7

ASYNCHRONOUS EVENTS

jaink@newschool.edu

https://canvas.newschool.edu/courses/1661668

https://replit.com/@jaink/pgte-5505-f22

https://NewSchool.zoom.us/j/91939750510?pwd=dE5tM1dzeUlpelNlQTJYUUVBY003UT09

https://github.com/kujain/F22-5505_Javascript

RECAP

THE CALL STACK

The JavaScript engine (as provided by the browser), is a **single-threaded** interpreter comprising of a **heap** and a **single call stack**.

- ➔ It is single-threaded. Meaning it can only do one thing at a time.
- ➔ Code execution is synchronous.
- ➔ It works as a LIFO — Last In, First Out data structure.

The call stack is a data structure that temporarily stores and manage function calls.

```
function firstFunction(){
    console.log("Hello from firstFunction");
}

function secondFunction(){
    firstFunction();
    console.log("Hello from secondFunction");
}

function thirdFunction(){
    secondFunction();
    console.log("Hello from thirdFunction");

}

thirdFunction();
```

ASYNCHRONICITY

How to get around the single-threaded nature of Javascript

- ➔ Using the Worker API - moves long delaying code to a different browser process, external to Javascript.
- ➔ Using asynchronous JavaScript (such as callbacks, promises, and async/await), you can perform long network requests without blocking the main thread.
- ➔ Not waiting for the results of a long-running computationl - by using asynchronous functions eg.
  - ◆ setTimeout()
  - ◆ XMLHttpRequest()
  - ◆ Promise()
  - ◆ Fetch
  - ◆ async/await

AJAX

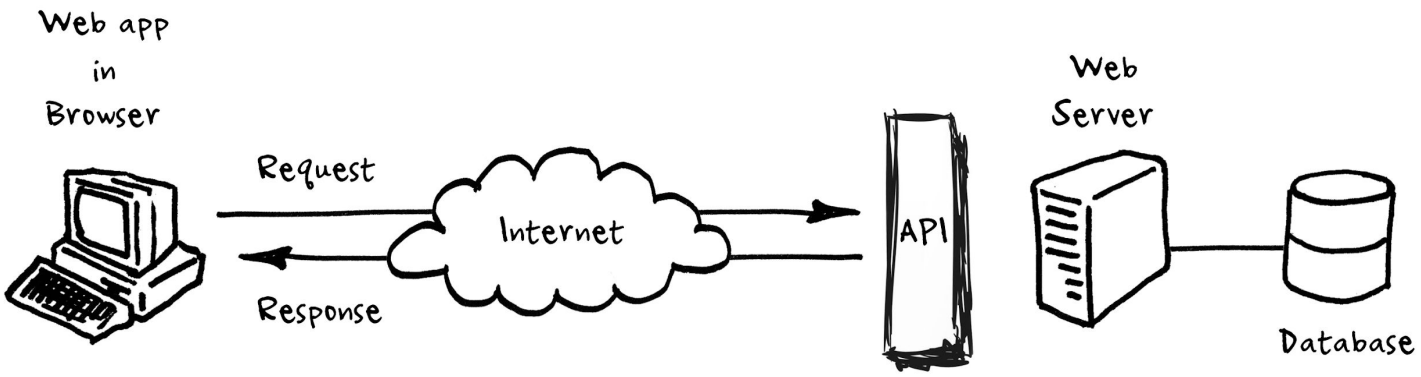
- ➔ Allows Javascript to take over the form submissions and process outside the user interface, transmitting the data asynchronously.
- ➔ Creates a more seamless UX since user doesn't have to wait for a page to keep refreshing after every action.
- ➔ Asynchronous
- ➔ Javascript
- ➔ XML (though mostly JSON now)

What is API?

- ➔ Internet is a web of accessible servers that you can connect using your client.
- ➔ Client makes requests and server returns data back.
- ➔ API -> Application Programming Interface is the intermediary in this process.
- ➔ API takes requests, translates, and returns responses - allowing two systems communicate with one another to access data based on a set of rules defined within the API.
- ➔ Allows standardized controlled (and possibly secure) exposure to a server's internal systems and data without exposing the internal source code, data structure etc.
- ➔ Stays independent of the source code and database structure - is not affected even if the entire server is rebuilt/overhauled.

What is REST?

- ➔ Built to handle client-server relationships ie client (browser) makes a request and server (web server) responds to it.
- ➔ Rest API is a set of rules that allow programs to communicate with each other using the REST structure. eg each rest api url should get a piece of data.
- ➔ Restful structure features:
  - ◆ All data required is provided in the request itself - no other global dependencies
  - ◆ Separation between client and server - completely independent and can be replaced.
- ➔ Standard url composition which is then documented and provided as reference. Also uses standard web methods to perform the actions:
  - ◆ GET: read or retrieve data/resource
  - ◆ POST: create new resource
  - ◆ PUT: update/create resource
  - ◆ DELETE: delete a resource

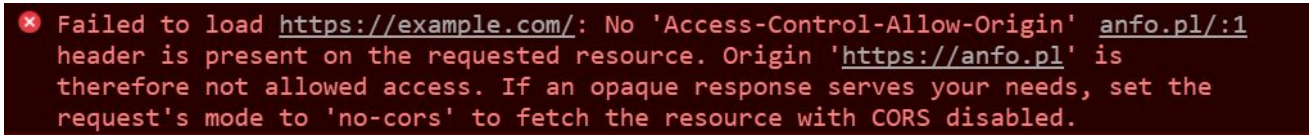


http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/

What is CORS?

- ➔ Due to security concerns, many APIs and sites have stopped allowing requests from external (ie different domain) Javascript clients.
- ➔ Basically to stop any malicious JavaScript being run from an external source.
- ➔ But most APIs are based on this foundation!
- ➔
- ➔ CORS is an implementation solution that allows requests to be made with some additional headers that indicate these restrictions - maybe specific domains, or authentication etc.
- ➔ With CORS implemented, requests will be allowed as per:
  - ◆ Different domain
  - ◆ Different subdomain
  - ◆ DIfferent port
- ➔ Example scenario:

This prevents attackers that plant scripts on various websites (eg. in ads displayed via Google Ads) to make an AJAX call to www.mybank.com and in case you were logged in making a transaction using *your* credentials.



JSON OVERVIEW

JSON is a string representation of the object literal notation:

```
var person = {
            "first_name": "John",
            "last_name": "Smith",
            "Schools": ["Parsons", "NYU"]
        };
```

Convert to string to pass as parameters:

```
var person_payload = JSON.stringify(person)
//
"{"firstName":"John","lastName":"Doe","age":30}"
```

Convert to string to pass as parameters:

```
var person_object = JSON.parse(person_payload)

// {firstName: "John", lastName: "Doe", age: 30}
```

XMLHttpRequest Object

AJAX uses the XMLHttpRequest (XHR) DOM object.
It can build HTTP requests, send them, and retrieve their results .

```
const xhr = new XMLHttpRequest();

xhr.open("GET", "https://google.com/search",
true); //3rd option is for async or sync

xhr.send("{'image_id':1}");
```

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

AJAX PROPERTIES

status:

```
200, 201 for a successful request
404 if it cannot find the endpoint
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

AJAX LISTENERS METHODS

setRequestHeader: sets the request data type: text, JSON etc.

```
xhr.setRequestHeader("Content-Type",
"application/json");
```

onerror: when the request couldn't be made, e.g. network down or invalid URL

onprogress: triggers periodically while the response is being downloaded

onload: when the request is complete (even if failed), and the response is fully downloaded

```
xhr.onload = callback_function;

eg.
xhr.onload = processXresponse;

function processXresponse() {
    if (xhr.status === 200) {
        // completed - do something with the response
    } else {
        // error - respond accordingly
    }
}
```

AJAX RESPONSE

response: returns the response sent back from the server

responseType: returns the type of data contained in the response

responseText: returns the text version of the response

AJAX METHODS - GET

```
let button = document.getElementById('GetUsers');

button.addEventListener("click", getUserData);

function getUserData() {
    var url = "https://reqres.in/api/users";
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
        if (xhr.status === 200) {
            document.getElementById("Output").innerHTML
= xhr.responseText;
        } else {
            document.getElementById("Output").innerHTML
= "There was an error";
        }

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

AJAX METHODS - POST

```
let button = document.getElementById('GetUsers');

button.addEventListener("click", sendUserData);
function sendUserData() {
    var url = "https://reqres.in/api/users";
    var xhr = new XMLHttpRequest();
    xhr.onload = function() {
        if (xhr.status === 201) {
            document.getElementById("Output").innerHTML
= xhr.responseText;
        } else {
            document.getElementById("Output").innerHTML
= "There was an error";
        }

    }
    xhr.open("POST", url, true);
    xhr.send("name=jason"); //data needs to be the
format expected, name=value pairs, json etc.
}
```

FORMDATA

Collects all data in a form in an object to be sent with AJAX:

```
var data = new FormData(form);
xhr.send(data);
```

To append:

```
data.append("name","name of person");
```

To loop:

```
for (let pair of data.entries()) {
     jsonObject[pair[0]] = pair[1];
}
```

To send:

```
xhr.send(data);
```

CALLBACKS

```
function printString(string, callback){
  const delay = Math.floor(Math.random() * 1000) + 1;
  setTimeout( function() {
      console.log(string, delay);
      callback();
    },
    delay
  )
}
// in parallel(ish)
function printAll(){
  printString("A");
  printString("B");
  printString("C");
}
// in turn
function printAll(){
  printString("A", function() {
    printString("B", function() {
      printString("C", function(){} )
    })
  })
}
```

PROMISE API

➔ **A new approach to avoiding callback hell: "Promises" to simplify the process**

```
new Promise(
    executorFunction(resolver, rejector) {
        … logic…
    }
);
```

➔ **A Promise can be in one of three states:**

- pending (when associated task is not finished yet)

- fulfilled (after and if task finishes successfully)

- rejected (after and if task fails). Once promise changes its state, it's done.

➔ **Promises can be chained - if each then() method yields and returns new promise. This allows to create elegant sequences of dependent tasks.**

```
const promise = new Promise( (resolve, reject) => {
    // initialization code goes here
    if (success) {
        resolve(value);
    } else {
        reject(error);
    }
});
```

PROMISE API

```
function printString(string){
     return new Promise( function(resolve, reject) {
          const delay = Math.floor(Math.random() * 1000) + 1;
          setTimeout( function() {
               console.log(string)
               resolve()
              },
             delay
          )
     })
}

function printAll() {
          const a = printString("A");
          const b = a.then( function() {
               return printString("B");
          });
          const c = b.then( function() {
               return printString("C");
          });
}
```

FETCH API - GET

Fetch is an adapted Promise designed to logically chain asynchronous responses together.

```
button.addEventListener("click", getUserData);

function getUserData() {
  let url = "https://reqres.in/api/users";
    fetch(url)
      .then(function(response) {
        return response.json();
      })
      .then(function(resp) {
        document.getElementById("Output").innerHTML =
JSON.stringify(resp.data);
      })
      .catch(function(resp) {
      document.getElementById("Output").innerHTML =
"There was an error";
      });
}
```

FETCH API - POST

```
const form = document.getElementById('createUser')
form.addEventListener("submit", saveUserData);

function saveUserData(e) {
  e.preventDefault();

  const url = "https://reqres.in/api/users";
  const FD  = new FormData(form);
  FD.append("name",form.first_name.value + ' ' +
form.last_name.value);
  let jsonObject = {};
  for (let pair  of FD.entries()) {
      jsonObject[pair[0]] = pair[1];
  }
  console.log(jsonObject);

  fetch(url, {
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify(jsonObject)
  })
    .then(function(response) {
        console.log(response.json());
        return response.json();
    })
    .then(function(data) {
        console.log('raw data',data);
        document.getElementById("Output").innerHTML =
"Successfully created id: "+data.id;
    })
    .catch(function(error) {
        document.getElementById("Output").innerHTML = "Th1ere was
an error "+error;
    });
}
```

ASYNC FUNCTIONS

Added in ES2017: a wrapper for calling Promise to further simplify the code:

```
// Promise object remains the same
function printString(string){
     return new Promise( function(resolve, reject) {
          const delay = Math.floor(Math.random() * 1000)
+ 1;
          setTimeout( function() {
               console.log(string)
               resolve()
              },
           delay
          )
     })
}
async function printAll(){
  await printString("A");
  await printString("B");
  await printString("C");
}
printAll();
```

Midterm Assignment

Next Steps

- ➔ Error Handling and Debugging
- ➔ OOP Concepts
- ➔ Midterm Discussion/Work in progress