Retrieving data with JavaScript

Sections in this chapter:

1. XMLHttpRequest
2. JSON
3. fetch

6-1. XMLHttpRequest

With JavaScript we can **interact with servers**. *6-1-1*

We can do HTTP requests and retrieve data from urls.

We can also use other protocols, such as `file` and `ftp` *6-1-2*

To do this with pure JavaScript we use `XMLHttpRequest` (XHR) objects. *6-1-3*

Although we have **XML** in the name `XMLHttpRequest`, we can **retrieve data of any type**, JSON for example, not only XML. *6-1-4*

We must call the `XMLHttpRequest` before making any other methods calls. *6-1-5*

```
let request = new XMLHttpRequest();
```

The `XMLHttpRequest` contains several methods to perform different actions, and properties with info regarding the request. 6-1-6

- `onreadystatechange`
- `response`
- `.open()`
- `.send()`

Above are a few examples, but there are many more.

You can find a list of available properties and methods on MDN. 6-1-7

Make a request 6-1-8

As stated before, we must create a new object using the `XMLHttpRequest()` constructor.

We initialize the request using the `.open()` method.

.open() 6-1-9

We specify what kind of request, and to what url, we want to make by passing this as parameters to the open() method.

```
request.open("GET", "getTheData.txt");
```

GET request to getTheData.txt

The first argument is what HTTP method we want to use, such as: 6-1-10

- `GET`
- `POST`
- `PUT`
- `DELETE`

The second argument is to what **url we want to send the request**, a server or a file for example. 6-1-11

There are other syntax for how to use the `.open()` method with more options. 6-1-12

```
XMLHttpRequest.open(method, url)
XMLHttpRequest.open(method, url, async)
XMLHttpRequest.open(method, url, async, user)
XMLHttpRequest.open(method, url, async, user, password)
```

.send()

6-1-13

To set the request in motion we use the `send()` method.

```
request.send();
```

A full example of a request reading a file could look like:

6-1-14

```
let request = new XMLHttpRequest();

request.onreadystatechange = () => {
    if (request.readyState == XMLHttpRequest.DONE) {
       if (request.status == 200) {
           // request succeeded
           console.log(request.responseText)
       }
       else {
           console.log('something else other than 200 was returned');
       }
    }
};

request.open("GET", "some_file.txt");
request.send();
```

6-2. JSON

Previously, data was sent between frontend and backend as **XML inside strings**. 6-2-1

This was a **horrible** time to be alive.

For instance, if you wanted to represent a customer in XML it would look like this: 6-2-2

```
<Customer>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <email>john.doe@mail.com</email>
</Customer>
```

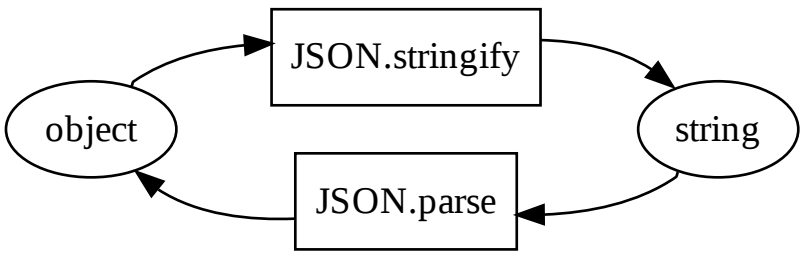
Now, we use **JSON** instead! JSON stands for **J**avaScript **O**bject **N**otation, and consists of the simple but brilliant idea to use **JavaScript object as a data format**. 6-2-3

JSON, a brief history 6-2-4

Douglas Crockford wrote the specification on JSON, in 2006.

By that time, he (and others) had already used it for several years. Already in 2005, **Yahoo!** was offering some web services in JSON. In 2006, **Google** started offering JSON as part of their GData protocol.

Today JSON is one of the most popular data exchange formats on the Net, supported by most major **web APIs** out there. **Big browsers** provide JSON parsing as part of their JavaScript implementations. Usage consists of using the **`parse` and `stringify` methods** on the `JSON` object: 6-2-5



The `JSON.parse()` method takes a JSON string as an argument and *parses* it, meaning it will create a JavaScript object or value described by the string it receives. 6-2-6

```
let customer = '{"firstName": "John", "lastName": "Doe", "email": "john.doe@mail.com"

let customerObject = JSON.parse(customer);
console.log(typeof customer); // string
console.log(typeof customerObject); //object
```

`JSON.stringify` does the opposite, it converts a JavaScript object or value to a string. 6-2-7

```
let customer  = {
    firstName: "John",
    lastName: "Doe",
    email: "john.doe@mail.com"
}

let customerString = JSON.stringify(customer);
console.log(typeof customer); // object
console.log(typeof customerString); // string
```

6-3. fetch

As you saw earlier, we can make AJAX requests with **XMLHttpRequest**. *6-3-1*

```
let request = new XMLHttpRequest();
```

However, when we need multiple requests, this gets messy quickly. *6-3-2*

People therefore often made a **Promise wrapper** around `XMLHttpRequest`.

Now there is a native API that does that: **fetch**! *6-3-3*

```
fetch(url, options).then(handler);
```

As you see, the `fetch` call returns a promise which we can attach a handler to using `.then`.

This means we can parallelise multiple requests: *6-3-4*

```
let getAll = Promise.all([
  fetch(url1),
  fetch(url2),
  fetch(url3)
]).then(handleAll);
```

The handler is called with a **response** object that contains... *6-3-5*

- data about the state of the request
- methods for extracting the data

Most commonly you'll extract **JSON** from the response using the `.json` method. That returns another promise! 6-3-6

```
fetch(url).then(resp => resp.json().then(data => /* use data here */));
```

Because it is all promises we can **chain** them instead:

```
fetch(url)
  .then(response => response.json())
  .then(data => /* use data here */)
```

Q So what about **browser support** - can we safely use `fetch` today? 6-3-7

A Well, it depends. Show the Can-i-use metrics to the client and talk about money! 6-3-8

As of this writing, it is supported pretty much everywhere except IE.

Read more here: 6-3-9

- reference: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- response: https://developer.mozilla.org/en-US/docs/Web/API/Response
- guide: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch