

FETCH()

in JavaScript

Currently in JavaScript we can download the contents of files/URLs using **XMLHttpRequest** objects.

The asynchronous nature of the operation is handled by callbacks (i.e. the **onreadystatechange** event handler).

This minimal example downloads the content of a URL and displays it in the console.

```
var xhr = new XMLHttpRequest();

xhr.open("GET", "http://atlantis.cit.ie");

xhr.onreadystatechange = function()
{
    if (this.readyState == 4 && xhr.status == 200)
    {
        console.log(this.responseText);
    }
}

xhr.send();
```

A newer method for reading external resources uses Promises.

I.e. the **fetch()** method.

The **fetch()** method uses a **Promise** for retrieving the data, and we use another for processing the data.

```
fetch("http://atlantis.cit.ie")  
  
  .then(function(resp) { return resp.text(); })  
  
    .then(function(txt) { console.log(txt) });
```

This can be written even more concisely using arrow functions.

```
fetch("http://atlantis.cit.ie")  
  
  .then(resp => resp.text())  
  
    .then(text=>console.log(text))
```


fetch()

fetch() is a method of the global object in JavaScript (e.g. **window** in client-side JavaScript, **WorkerGlobalScope** in workers , etc).

This means you can use it without explicit reference to a receiver.

E.g.

```
fetch ("file.txt") ;
```


fetch() is passed a pathname (as a string) or **Request** object.

It returns a **Promise** that resolves when the the file/URL represented by the parameter has been retrieved.

It resolves to a **Response** object. This object contains and describes the data that was received.

Making the request

You can easily specify a resource you want to fetch by passing its pathname as a string.

```
fetch("file.json");
```

```
fetch("http://atlantis.cit.ie/getdata.php");
```

Alternatively, you can also create a **Request** object that contains the details of your request. You can then pass this to the **fetch()** method.

```
var req = new Request("getdata.php");  
  
fetch(req);
```


If you need to provide more information with your request you can provide an ***options object***.

An **options object** allows you specify various parameters by putting them all in a single object as object properties, rather than passing them to a function as individual parameters.

E.g. we can specify the extra data we need in order to make a POST request as follows:

```
{  
  "method": "POST",  
  "body": "x=5&y=4",  
  "headers":  
    { "Content-Type":  
      "application/x-www-form-urlencoded" }  
}
```


We can pass this to a **Request** constructor as the second parameter.


```
var req = new Request( "getdata.php",  
    {  
        "method": "POST",  
        "body": "x=5&y=4",  
        "headers":  
        { "Content-Type":  
            "application/x-www-form-urlencoded" }  
    }  
);  
  
fetch(req);
```

path

```
var req = new Request( "getdata.php",  
    {  
        "method": "POST",  
        "body": "x=5&y=4",  
        "headers":  
        { "Content-Type":  
            "application/x-www-form-urlencoded" }  
    }  
);  
  
fetch(req);
```


Options object

```
var req = new Request( "getdata.php",  
    {  
        "method": "POST",  
        "body": "x=5&y=4",  
        "headers":  
        { "Content-Type":  
            "application/x-www-form-urlencoded" }  
    }  
);  
  
fetch(req);
```



Note: The parameters you use to create a **Request** object can also be used directly in the **fetch()** command instead.

```
fetch( "http://atlantis.cit.ie/displayvalues.php",  
      {  
        "method": "POST",  
        "body": "x=5&y=4",  
        "headers": { "Content-Type": "application/x-www-form-urlencoded" }  
      }  
    );
```


Frequently the **Request** (and **Response**) objects you use in your code have been automatically created by JavaScript built-in objects.

Handling the Response

fetch() does not return the data from the request directly.

Instead it returns a promise that resolves when the data is received.

```
fetch ("file.txt")  
    .then ( <process response> )
```

This promise resolves to a **Response** object that we can pass to the then handler function.

```
fetch("file.txt")  
  .then ( function(resp) { <process response in resp> } )
```



The Response object has several properties describing the response.

This includes properties that let you know if the request that generated the request was successful.

You need to use these properties since the promise returned by **fetch()** is not rejected if the request fails.

The properties that describe the status of the response are:

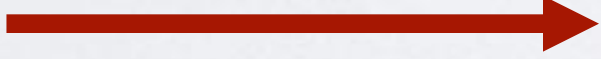
status

statusText

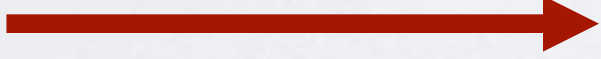
ok

status	contains the HTTP status code for the response. E.g. 200
statusText	contains the description of the status. E.g. OK
ok	contains true if the status code is between 200-299: E.g. true

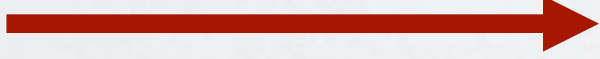
Now we can check the **Response** is valid before processing it.

```
fetch("file.json")  
  .then( function(resp) {  
  
        if (resp.ok)  
    {  
      <process response in resp>  
    }  
  
  }  
)
```

Now we can check the **Response** is valid before processing it.


```
fetch("file.json")  
  .then( function(resp) {  
     if (resp.status == 200)  
    {  
      <process response in resp>  
    }  
  }  
)
```


Now we can check the **Response** is valid before processing it.

```
fetch("file.json")  
  .then(function(resp) {  
     if (resp.statusText === "OK")  
    {  
      <process response in resp>  
    }  
  }  
)
```

If the server returns an error the **Promise** returned by **fetch()** will still resolve and provide you with the information about the error as we have seen.

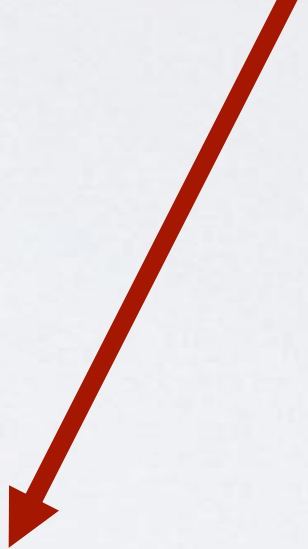
However, if there is an network error, or some issue with permissions, etc., the **Promise** returned by **fetch()** will reject.


```
fetch("file.json")  
  
  .then( function(resp) {  
  
    <process response in resp>  
  
  })  
   .catch(function() {  
  
    console.log("Network Error");  
  
  })
```

Response objects contain other information about the response.


E.g. information about the response's headers:

```
fetch("file.json")  
  
  .then( function(resp) {  
    console.log(resp.headers) ;  
  } )
```



The URL that was the source of the response:

```
fetch("file.json")  
  .then(function(resp) {  
    console.log(resp.url);  
  })
```



etc.

Accessing the Data

The data returned from a **fetch()** is not simply stored in a property as with **XMLHttpRequest** objects.

Instead you can access it as a **ReadableStream**.

This allows you read the data in chunks.

The stream is stored in the **body** property of the response.

You can use this property to get an instance of an object that can read the stream.

```
var reader = resp.body.getReader();
```


Now you can read the data from the stream in chunks.

You can use the **read()** method on the stream to get the next chunk.

```
reader.read()
```

Access to the content of a stream is asynchronous since we don't know how long the data will take to process.

As a result, streams use a **Promise** to return the data.

read() returns a **Promise** that resolves to an object with 2 properties:

- value** The content in the current chunk
- done** A boolean that will be true when all the content is read.

```
reader.read().then(function(data) {  
    if (data.done)  
        console.log("Finished");  
    else  
        console.log("Current = " + data.value);  
});
```

Notes:

To read a complete file you can repeatedly call **read()** until **done** is **true**.

If **done** is **true**, then **value** will be **undefined**.

To simplify their use, **Response** objects provide several methods that will process the stream for us and return the ***complete*** data in a **Promise**.

These methods can return data in more useful formats.

Of particular use are:

json()

text()

blob()

Here **text()** resolves with the text we fetched and we process it in the **then()** handler function.

```
fetch("file.txt")  
  
  .then(function(resp) {  
  
    resp.text() .then(function(txt)  
                                     {console.log(txt) }  
    ) } ) ;
```


This function is called when the **Promise** we got from **text()** resolves (i.e. when the text is ready).

This is the text we read from the file (resolved from the Promise returned by **text()**)

```
fetch("file.txt")
```

```
.then(function(resp) {
```

```
resp.text() .then(function(txt)  
    { console.log(txt) }  
    ) } ) ;
```

Response from **fetch()**

The **text()** function returns a **Promise**

Alternatively, we can return the **Promise** we get from **text()** and handle it the main chain .

```
fetch("file.txt")  
  
  .then(function(resp) { return resp.text() ; })  
  
    .then(function(txt) { console.log(txt) } ) ;
```


Cloning Responses

The body of a **Response** can only be used once.

I.e. you can only get a reader the one time, *or* you can call **json()** once *or* you can call **text()** once, etc.

You can check if the body has already been used by checking the (boolean) value of the **bodyUsed** property of the **Response**.

```
if (!resp.bodyUsed) { pr = resp.text() }
```


If you need to use the data in the body of a response more than once you can either:

- 1) retrieve the data from the body and make a copy of that data (assuming you want it in the same format).
- 2) clone the **Response** and use the clone to retrieve the data again.

Here we want to get a stream reader object and the **text()** from a **Response**. Since that requires using the body twice, we must clone the **Response** first.

```
fetch("getData.php")  
  
  .then(function(resp) {  
  
    var respClone = resp.clone();  
  
    doSomething(resp.text());  
  
    doSomethingElse(respClone.body.getReader());  
  
  });
```


Remember to clone the response **before you try to use it.**