codecademy

Requests

HTTP GET request

HTTP `GET` requests are made with the intention of retrieving information or data from a source (server) over the web.

`GET` requests have no *body*, so the information that the source requires, in order to return the proper response, must be included in the request URL path or query string.

JSON: JavaScript Object Notation

JSON or *JavaScript Object Notation* is a data format suitable for transporting data to and from a server.
It is essentially a slightly stricter version of a Javascript object. A JSON object should be enclosed in curly braces and may contain one or more property-value pairs. JSON names require double quotes, while standard Javascript objects do not.

```
const jsonObj = {
  "name": "Rick",
  "id": "11A",
  "level": 4
};
```

HTTP POST request

HTTP `POST` requests are made with the intention of sending new information to the source (server) that will receive it.

For a `POST` request, the new information is stored in the *body* of the request.

Asynchronous calls with XMLHttpRequest

AJAX enables HTTP requests to be made not only during the load time of a web page but also anytime after a page initially loads. This allows adding dynamic behavior to a webpage. This is essential for giving a good user experience without reloading the webpage for transferring data to and from the web server.
The XMLHttpRequest (XHR) web API provides the ability to make the actual asynchronous request and uses AJAX to handle the data from the request.
The given code block is a basic example of how an HTTP GET request is made to the specified URL.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'mysite.com/api/getjson');
```

The query string in a URL

Query strings are used to send additional information to the server during an HTTP GET request.
The query string is separated from the original URL using the question mark character `?`.
In a query string, there can be one or more key-value pairs joined by the equal character `=`.
For separating multiple key-value pairs, an ampersand character `&` is used.
Query strings should be url-encoded in case of the presence of URL unsafe characters.

```
const requestUrl = 'http://mysite.com/api/vendor?
name=kavin&id=35412';
```

`XMLHttpRequest` GET Request Requirements

The request type, response type, request URL, and handler for the response data must be provided in order to make an HTTP GET request with the JavaScript `XMLHttpRequest` API.
The URL may contain additional data in the query string. For an HTTP GET request, the request type must be `GET`.

```
const req = new XMLHttpRequest();
req.responseType = 'json';
req.open('GET', '/myendpoint/getdata?id=65');
req.onload = () => {
  console.log(xhr.response);
};

req.send();
```

HTTP POST request with the `XMLHttpRequest` API

To make an HTTP POST request with the JavaScript `XMLHttpRequest` API, a request type, response type, request URL, request body, and handler for the response data must be provided. The request body is essential because the information sent via the POST method is not visible in the URL. The request type must be `POST` for this case. The response type can be a variety of types including array buffer, json, etc.

```
const data = {
  fish: 'Salmon',
  weight: '1.5 KG',
  units: 5
};
const xhr = new XMLHttpRequest();
xhr.open('POST', '/inventory/add');
xhr.responseType = 'json';
xhr.send(JSON.stringify(data));

xhr.onload = () => {
  console.log(xhr.response);
};
```

ok property fetch api

In a Fetch API function `fetch()` the `ok` property of a response checks to see if it evaluates to `true` or `false`. In the code example the `.ok` property will be `true` when the HTTP request is successful. The `.ok` property will be `false` when the HTTP request is unsuccessful.

```
    fetch(url, {
    method: 'POST',
    headers: {
      'Content-type': 'application/json',
      'apikey': apiKey
    },
    body: data
  }).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
  }, networkError => {
    console.log(networkError.message)
  })
}
```

JSON Formatted response body

The `.json()` method will resolve a returned promise to a JSON object, parsing the body text as JSON.

The example block of code shows `.json()` method that returns a promise that resolves to a JSON-formatted response body as a JavaScript object.

```
fetch('url-that-returns-JSON')
.then(response => response.json())
.then(jsonResponse => {
  console.log(jsonResponse);
});
```

promise url parameter fetch api

A JavaScript Fetch API is used to access and manipulate requests and responses within the HTTP pipeline, fetching resources asynchronously across a network.

A basic `fetch()` request will accept a URL parameter, send a request and contain a success and failure promise handler function.

In the example, the block of code begins by calling the `fetch()` function. Then a `then()` method is chained to the end of the `fetch()`. It ends with the response callback to handle success and the rejection callback to handle failure.

```
fetch('url')
.then(
  response  => {
    console.log(response);
  },
 rejection => {
    console.error(rejection.message);
);
```

Fetch API Function

The Fetch API function `fetch()` can be used to create requests. Though accepting additional arguments, the request can be customized. This can be used to change the request type, headers, specify a request body, and much more as shown in the example block of code.

```
fetch('https://api-to-call.com/endpoint', {
  method: 'POST',
 body: JSON.stringify({id: "200"})
}).then(response => {
  if(response.ok){
      return response.json();
  }
    throw new Error('Request failed!');
}, networkError => {
  console.log(networkError.message);
}).then(jsonResponse => {
  console.log(jsonResponse);
})
```

async await syntax

The **async...await** syntax is used with the JS Fetch API `fetch()` to work with promises. In the code block example we see the keyword `async` placed the function. This means that the function will return a promise. The keyword `await` makes the JavaScript wait until the problem is resolved.

```
const getSuggestions = async () => {
  const wordQuery = inputField.value;
  const endpoint
= `${url}${queryParams}${wordQuery}`;
  try{
const response = __~await~__ __~fetch(endpoint,
{cache: 'no-cache'});
    if(response.ok){
      const jsonResponse = await response.json()
    }
  }
  catch(error){
    console.log(error)
  }
}
```