Working with JSON

Tags

- JSON, is a text based data format following JavaScript object Syntax. (see MDN JavaScript objects for more on JS objects)
- JSON exists as a string - useful to transmit data across a network. It needs to be converted to native JS objects to access the data. JS provides a global JSON object that has methods available to convert between JSON to JS e.g. `stringify()`
- JSON files can store the same basic things as JS objects i.e. Strings, integers, boolean, arrays, null values
- This means you can also have data hiearchy in JSON files

```
{
"suqadNamne" : "Justice League",
"homeTown" : "metro City",
"formed" : 2016
"active" : true,
"members" : [
{
    "name" : "batman",
    "power" : null
},

{
    "name" : "superman",
    "power" : ["SuperSpeed", "Super Strength"]
},
```

```
]
}
```

- How is this useful? If we loaded this string into JS program and parsed into a variable called superHeroes, we could access the data inside it using the same dot/bracket notation

example:

```
superHero.homeTown;
superHeroes["active"];
```

- To access the hieracial structure in the JSON files you have to chain the required properties together in an array. Example: What is the second superpower of the second member of the superhero squad

```
superHeroes["members"][1]["power"][2];
```

Other Notes:

- JSON is purely a string with a specified data format — it contains only properties, no methods.
- JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.
- Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as

long as the generator program is working correctly). You can validate JSON using an application like JSONLint.

- JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON.
- Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

Exercise:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">

    <title>Our superheroes</title>

    <link href="https://fonts.googleapis.com/css?family=Faster+(
    <link rel="stylesheet" href="JSONexercise_heroes.css">
  </head>

  <body>

      <header>

      </header>

      <section>

      </section>

      <script>
```

```
        async function populate() {

            const requestURL = "https://mdn.github.io/learning-a
            const request = new Request(requestURL);
            const response = await fetch(request);
            const superHeroes = await response.json();

            populateHeader(superHeroes);
            populateHeroes(superHeroes);
        }
    </script>
  </body>
</html>
```

- To obtain JSON, we can use an API called Fetch. This APO allows us to make a network request to retrieve resources from a server via JS (e.g. images, JSON, even HTML snippets)
-

To obtain the JSON, we use an API called Fetch. This API allows us to make network requests to retrieve resources from a server via JavaScript (e.g. images, text, JSON, even HTML snippets), meaning that we can update small sections of content without having to reload the entire page.

In our function, the first four lines use the Fetch API to fetch the JSON from the server:

- we declare the `requestURL` variable to store the GitHub URL
- we use the URL to initialize a new `Request` object.
- we make the network request using the `fetch()` function, and this returns a `Response` object
- we retrieve the response as JSON using the `json()` function of the `Response` object.

Note: The `fetch()` API is **asynchronous**. We'll learn a lot about asynchronous functions in the next module, but for now, we'll just say that we need to add the keyword `async` before the name of the function that uses the fetch API, and add the keyword `await` before the calls to any asynchronous functions.

After all that, the `superHeroes` variable will contain the JavaScript object based on the JSON. We are then passing that object to two function calls — the first one fills the `<header>` with the correct data, while the second one creates an information card for each hero on the team, and inserts it into the `<section>`.

- Then we need to create the populateHeader and populateHeores Functions:

▼ **Code:**

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">

    <title>Our superheroes</title>

    <link href="https://fonts.googleapis.com/css?family=Faste
    <link rel="stylesheet" href="JSONexercise_heroes.css">
  </head>

  <body>

      <header>

      </header>

      <section>
```

```
    </section>

    <script>

      async function populate() {

          const requestURL = "https://mdn.github.io/learni
          const request = new Request(requestURL);
          const response = await fetch(request);
          const superHeroes = await response.json();
          populateHeader(superHeroes);
          populateHeroes(superHeroes);
      }

      function populateHeader(obj) {

          // Create the header
          const header = document.querySelector("header");
          const H1 = document.createElement("h1");
          H1.textContent = obj.squadName;
          header.appendChild(H1);

          const para = document.createElement("p");
          para.textContent = `Hometown: ${obj.homeTown} //
          header.appendChild(para);
      }


      function populateHeroes (obj) {

       const section = document.querySelector("section");
       const heroes = obj.members

       for (const hero of heroes) {

          const article = document.createElement("article"
```

```
        const H2 = document.createElement("h2");
        const P1 = document.createElement("p");
        const P2 = document.createElement("p");
        const P3 = document.createElement("p");
        const list = document.createElement("ul");


        H2.textContent = hero.name;
        P1.textContent = `Secret Identity: ${hero.secret
        P2.textContent = `Age: ${hero.age}`;
        P3.textContent = "Superpowers:";

        const superPowers = hero.powers;

        for (const power of superPowers) {
            const listItem = document.createElement("li"
            listItem.textContent = power;
            list.appendChild(listItem);
        }


        article.appendChild(H2);
        article.appendChild(P1);
        article.appendChild(P2);
        article.appendChild(P3);
        article.appendChild(list);

        section.appendChild(article);


    }



}
```

```
        populate();

      </script>
  </body>
</html>
```

- But in the above code we used network response directly into a JS object using response.json
- Sometimes we aren't so lucky and recieve raw JSON strings, and we need to convert to objects our selves
- And when we need to send JS objects across a network we need to convert to JSON before sending it.
- This is done using two methods:
  - **parse(); Accepts a json string as a parameter and returns a corresponding JS**

In the above example, we can do this:

```
function populate() {

const requestURL = "https://mdn.github.io/learning-area/javas
const request = new Request(requestURL);
const response = await fetch(request);

const superHeroText = await response.text();

const superHeores = JSON.parse(superHeroText);
populateHeader(superHeroes);
 populateHeroes(superHeroes);
```

```
}
```

- **stringify(); Accepts a JS object as a parameter and returns a corresponding JSON file**