Learning Objectives: Simple HTTP Client

Learners will be able to...

- **Understand the tools needed for web development (npm, json files)**
- **Assemble and deploy a simple HTTP client using Javascript for a server**
- **Construct and parse a URI using a JS library**
- **Formulate headers for the request, e.g. `Accept`**

info

Make Sure You Know

Students should be familiar with the Javascript programming language.

Limitations

We will not cover basic programming skills.

Tools and Dependencies

Tools and dependencies

The tools you need to access Web APIs are built into browsers. In the examples, in this assignment, we will be running our sample HTTP client as a stand-alone application using **Node.js**. **Node.js** is an open-source cross-platform run-time environment for developing server and network applications in JavaScript.

The **Node.js** run time environment is asynchronous, event-driven and single-threaded. This means that the server does not wait for a call to complete before making other API calls, for example, a time-intensive task such as input/output.

We'll use **Node Package Manager (npm)** to install any JavaScript libraries we need for the samples on the following pages. The package manager provides access to Open Source packages you can use to build your **Node.js** applications.

In this assignment we have specified the dependencies for our examples in the file **package.json** which you can see in the window on the left. The file specifies that we need the **node-fetch** library for the `fetch` command.

There are several other popular HTTP client libraries available through **npm**. We will not be exploring them in this course, here is a listing if you would like to learn more:

- axios
- got
- superagent
- needle

The simplest HTTP client

The JavaScript `fetch` API

In the last assignment we looked at a couple of examples of the `fetch` API. You can use the `fetch` library to create an HTTP client. We will learn about it in more detail in this assignment.

The **Fetch API** does what the name implies, it provides a way to "fetch" resources. There is only one **mandatory** argument to the `fetch` command, the path of the resource you are interested in. If the only argument you pass to fetch is a URL, it is by default a GET request.

The example on the left is a simple request for all the weather alerts from the government weather service. It lists all the alerts without filtering by state. As you can see it has a single argument, the URL for the active weather alerts: https://api.weather.gov/alerts/active.

Output of the `fetch` command

You can view the output of `fetch` in text format, or in a more structured fashion as a `json` object, depending on which method you use for the `response` command.

Output in text format `response.text()`

Here is the fetch command and the response in text format:

Output in json format `response.json()`

Here is the fetch command and the response in json format:

important

Installing the node modules

The first time you run something in each of the assignments in this course you need to make sure that you have all the proper node modules installed. To do that run the commands below in the terminal.

```
cd examples
npm i

</fieldset>


Run the code on the left from the command line in the
        terminal:

```bash
nodejs 01-weather-alerts-text.js | more
```

▼ What is the "| more"?

The output of the commands on this page are being "piped" to the Linux utility "more" to slow down the output. Type the space bar to scroll one screen or the letter 'q' to stop the output.

Run the json version:

```
nodejs 02-weather-alerts-json.js | more
```
```

Query parameters

Adding a query to the URI

We can add a query string to return values for a specific state. The query string is added to the URI.

In this example we have added `area=MA` as the query part of the URI:
- `area` is the name of the parameter
- `MA` is the value we want.

The URI looks as follows:
`https://api.weather.gov/alerts/active?area=MA`

Take a look at the URI with the query:

Run this from the command line in the terminal:

```
cd examples
nodejs 03-weather-alerts-static-filter.js
```

Constructing a query with user input

There are restrictions on which characters can be used in URI. You need to check for potential invalid characters when you are using user input to construct the URI or you might need to do something else with the user input before it can be added to a URI. Invalid characters can be encoded using URL encoding also known as percent-encoding.

All alphanumeric characters are valid in a URI. Examples of reserved characters are spaces and the following: `! * ' ( ) ; : @ & = + $ , / ? % # [ ]`. When you need to use a character that is on the reserved list in a URI it must be **percent-encoded**. The encoded character sequence is a percent sign followed by two hexadecimal digits that correspond to the byte value of the ASCII character.

In the case of , the caller passes the state they are interested in on the command line. The code on the left takes the two-letter string that is passed on the command line `ca` and upper cases it before the call to fetch. The API requires that the state name be in uppercase.

The example that uses the URLSearchParams object to manipulate the query string:

Try it out, paste this on the command line in the terminal.

```
nodejs 04-weather-alerts-query.js ca
```

Parsing URIs and working with them

Parsing a URI/URL is a common task, NodeJs provides the URL class to facilitate this process. This might need to be done in cases when the URI comes from a configuration file, environment variables or user input.

To parse the URL you use the constructor:

```
new URL(string)
```

If the string is a valid URL the result will be an object that has the following properties:
- `protocol` - the *scheme* part of the URI, for example `http:` or `https:`
- `hostname` - for example `www.example.com`
- `port` - can be empty or an integer like `8080`.
- `host` - a combination of `hostname` and `port`, e.g. `www.example.com:8080`. When the port is empty it is equal to `hostname`.
- `username` and `password` - can be specified in the URL for some applications but it is not recommended for security reasons.
- `origin` - is the combination of `protocol`, `hostname`, and `port`. A browser might compare the origin of a script and the origin of the resources the script is accessing, if the origins differ this may lead to a security problem. This is a read-only value.
- `pathname` - consists of a list of path segments separated by (/). HTTP and HTTPS URLs always have at least one path segment. If it is empty, it will be represented by a single slash resulting in two consecutive slashes (//) in the path component.
- `search` - a query string.
- `searchParams` - a parsed query component of a URI as `URLSearchParams`. This is a read only value, you can use the returned `URLSearchParams` object to modify the URL. The modified URL can be turned into a string by using the `toString` method as in the example on the left.

- `hash` - a fragment component of a URI.
- `href` contains the whole URL as a string.

POST method and the request body

In this example we are creating a `POST` rather than a `GET` request. The sentiment to be analyzed is passed in on the command line. This time the `fetch` call has more arguments than just the URL, it is posting the sentiment to be analyzed. The request method can be passed in the parameters along with the request body.

View the fetch command for a POST request:

The request body requires the URL-encoded format for this API, we can create that using the URLSearchParams object.

Using the URLSearchParams object to manipulate request string:

Try it out, paste this on the command line in the terminal. You can modify the sentiment string and see how it evaluates other text.

```
cd examples
nodejs 05-sentiment-post-body.js "I love pizza"
```

Adding the headers

In the previous examples we did not have to pass any headers because `fetch` adds some of the required headers automatically.

Normally when a request has a body, it's important to specify its format too. This can be done by adding the `Content-Type` header.
In the previous example the header was automatically added by `fetch`, it was `Content-Type: x-www-form-urlencoded`. That is how `fetch` knows to interpret the request body as`URLSearchParams`.

To adhere to REST requirements we need to send the format of the *request* body and also to specify which formats we can accept as the *response* body.

A server may have multiple versions of a piece of information, it may have it translated to a few different languages (English, French) and formats (JSON,XML).

These are the header values you can use to provide more detail about the format you want the information in:

```
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
```

This is a modified version of the weather example. Our previous example did not have a request body but we are expecting a JSON as a response, so now we will specify that using `'Accept': 'application/json'`.

Specifying the request body:

Try it out, paste this on the command line in the terminal. You can change the state argument to get data for other states.

```
cd examples
nodejs 06-weather-alerts-headers.js MA
```

In some cases the client can request multiple different formats for the response and the server will decide which is the best format to return the data in. This is referred to as **Content negotiation**