

# XML and JSON data sources

It is useful to understand how to read from XML and JSON data sources. These are common data formats for open data sources, which can be used to enrich your site because of the ability to use data from *other* sources.

Open data sources are resources which you can access for free. Many governments have open data sources you can use. A quick Google search will find many open data sources. Here are some open data catalogues to start:

- City of Toronto open data catalogue (<https://open.toronto.ca/catalogue/>)
- Ontario data catalogue (<https://data.ontario.ca/>)
- Government of Canada open data catalogue (<https://open.canada.ca/en/open-data>)

JSON is more popular now, but XML was used a lot for older resources. Both XML and JSON are plain-text formats and are platform agnostic (i.e. can be read on any computer). They are also machine-readable (i.e. easy for machines to parse) *and* human-readable (i.e. easy for people to parse). This document will not go over JSON as JSON is what we've already been using (e.g. when we tried the Trakt API earlier, that was JSON). A JSON data source is used in a similar way.

Regarding open data sources: you can load the data directly from the resource URL, but this can be slower and if you're expecting a lot of traffic, it's probably not the best idea. Instead, you can download the resource file and use it in your site. This has faster load times and avoids CORS issues. The drawback is that for types of data which are often refreshed, you'll have less fresh data depending on how often you refresh your data file. Fortunately, nowadays, for data which changes more often, most likely this will be accessed with an API rather than via a resource file and open data source files are usually used for data which doesn't change too often.

## XML basics

XML (Extensible Markup Language) is a markup language made up of elements and attributes. If you look at XML data, you'll see that it looks quite similar to HTML (see example below of an XML file's contents) but XML is used primarily for **data** structures.

```
<?xml version="1.0" encoding="utf-8" ?>
<people>
  <person id="1" job="accountant">
    <name>
      <first>John</first>
      <last>Doe</last>
    </name>
  </person>
  <person id="2" job="wizard">
    <name>
      <first>Harry</first>
      <last>Potter</last>
    </name>
  </person>
</people>
```

XML is defined by the following characteristics:

1. There is one root element (e.g. in an HTML file, the `<html>` element is the root element). This characteristic is the key to making it easy to parse for machines (it's much more difficult to query if there are multiple root elements).
2. All elements must be closed. If self-closing, the element must use the closing slash.
3. XML is not limited to a specific set of tags and semantics, although they are limited by user-defined rules if they exist. (If a developer creates a specific XML structure to be used, he/she also creates a set of rules other devs must follow when using the XML structure/other code that requires that XML structure.)
4. XML is case-sensitive.
5. Element/attribute names are defined by the creator of the particular XML data structure **provided it follows the XML naming rules**. The naming rules aren't so important for us since we're not focused on creating custom XML structures; we're just worried about using them. If you're curious, see the list below.

#### XML element naming rules

1. Element names **must not contain spaces** (e.g. `<first name>` is invalid).
2. Element names **must begin with letters or an underscore, but no other characters**. For example, `<1stName>` is invalid because it begins with a number. `< firstname>` is also invalid because it begins with a space (do **NOT** put leading whitespace before an element name). `<-firstname>` is invalid because it begins with a hyphen.
3. Element names may contain numbers, hypens, and underscores **as long as the first character is a letter or underscore**.
4. Element names **MUST NOT contain the special character ":"**.
5. Element names **MUST NOT begin with "xml"**. The case doesn't matter for this special word because any element name starting with "xml" or "Xml" or "XML" or even "xMI" and so on is forbidden because it is a reserved word.
6. Element names are **case-sensitive**. `<applicationname>` is **not** the same as `<ApplicationName>`.

## **XML DOM**

Previously, you've seen that HTML can be represented as a DOM tree. XML—since it's a markup language—can also be traversed using the DOM, but there are some things which are specific to HTML only in client-side JS (e.g. document, window, etc). Note that any XML-based language (e.g. KML) can also be traversed/queried and read using DOM methods and XPath.

**Useful properties are:**

- **parentNode**: get the parent node of the current element/node
- **parentElement**: get the parent element of the current element
- **attributes**: get the list of attributes for the current element
- **firstChildElement**: the first child which is an element
- **lastChildElement**: the last child which is an element

- **textContent**: the text within the element tags. This is probably the most used if just querying and using data.

Useful methods are:

- **xmlDoc.getElementsByTagName("<tag\_name>")**: returns a list of elements matching the specified tag name, where xmlDoc is an XML DOM document object
- **element.getAttribute("<attr\_name>")**: return value of a specified attribute, where element is an element object
- **xmlDoc.querySelector("<selector>")**: query using a selector format and get a single result (first match)
- **xmlDoc.querySelectorAll("<selector>")**: query using a selector format and get a list of results

If you want to generate XML, you could use the methods below, but we don't really need them if just reading XML data.

- **xmlDoc.createElement("<element\_name>")**: create an element on a given XML DOM document object
- **xmlDoc.createTextNode("<text\_content>")**: create a text node to be inserted in an element
- **node.appendChild(<node\_to\_append>)**: append an element or text node to a node (usually an element)
- **element.setAttribute("<attr\_name>", "<attr\_value>")**: create and add an attribute and attribute value to an element

## ***XPath***

XPath is a method of representing an XML node (can be element/attribute) as a **path** (i.e. think of how file paths look). Using XPath can be quite efficient and can be used alongside DOM methods as well. Where XPath shines is for complex queries (e.g. select a <person> element whose id attribute equals "50"). Using DOM methods alone (e.g. querySelector()) can be quite long and inefficient because you'd most likely need a loop.

Using the XML on page 1, you can represent the <person> whose first name value is "Harry" as the following path:

```
//person[./first/text()='Harry']
```

We'll go over the most useful path syntaxes.

**Path axes (each "chunk" in the path is referred to as a step in the path):**

- **//**: represents "descendant" (e.g. //first looks for a descendant element named "first"). If there is nothing before //, the query will return all matching elements which are descendants of the root of the document. If there is something before //, the query will return all matching descendants of the current **node**.
- **element**: text with no other symbols in a path represents an element name
- **@attr**: attributes are prefixed with @ (e.g. //person/@id selects the id attribute for all descendant <person> elements)

- `.`: represents the current node
- `..`: represents the parent element

### Predicates [<condition(s)>]:

Predicates allow you to define conditions. The result of a query is the last step in the path outside a predicate. For example, `//person[./first/text()='Harry']`, `<person>` is the last thing outside the predicate (it's the last step in the path) so the result will be a `<person>` element. The path is actually querying for a `<person>` element whose descendant `<first>` (`./first`) element's text content (`/text()`) equals 'Harry' (`= 'Harry'`).

You can use an **and** or **or** operation within a predicate using multiple conditions. You cannot use multiple sets of square brackets for a particular step in the path.

### Node tests:

- `text()`: This returns the text content for an element. In the XPath example above it has been used in a predicate (condition).
- `position()`: This returns a number identifying the position where the element is located in a list of elements (i.e. like the child number in CSS). This node test is not too useful in practical examples as getting an element by value is more useful than by position.

#### Selector and XPath cheat sheet

<https://devhints.io/xpath>

The above cheat sheet includes selector syntax and their equivalent XPath syntaxes.

#### Testing XPath queries in a browser

You can test XPath queries in your browser by using the console in dev tools.

1. Open the XML file in the browser.
2. Open the console in the browser.
3. Test a query by type `$x("<my_xpath>")`, replacing `<my_xpath>` with the XPath you want to query.

There's a DOM `evaluate()` method you can use to evaluate XPath.

<https://developer.mozilla.org/en-US/docs/Web/API/Document/evaluate>

## Reading XML sources

### Node.js

There are different ways to read XML in Node.js. You can install packages which use the standard DOM and XPath or you can find packages which convert to JSON.

XPath and standard DOM:

- [jsdom](#) (install with npm): can use DOM methods and properties and XPath (for `doc.evaluate()`, you need to specify XPathResult constant value—i.e. the number value)

Other packages to try:

- [fast-xml-parser](https://geshan.com.np/blog/2022/11/nodejs-xml-parser/) (install with npm): Convert XML to a Javascript object (see: <https://geshan.com.np/blog/2022/11/nodejs-xml-parser/>)

### Using jsdom:

We'll stick with using Fetch so that we're using something familiar. (If you wanted to load a local file, you could also use the fs module to read file streams.)

➤ `npm install --save jsdom`

The following is a snippet of code (in the example code, take a look at the parks component file in `/components/parks/index.js`).

```
const jsdom = require("jsdom");
const { JSDOM } = jsdom;

var xml;

async function loadXml() {
  if (xml == undefined) {
    //url from Toronto Open Data catalogue
    let response = await fetch(
      "http://localhost:8888/facilities-data.xml", //file in /public
      {
        method: "get",
        headers: {
          "Content-Type": "application/xml"
        }
      }
    );
    //convert XML string to XML DOM document
    data = new JSDOM(await response.text(), { contentType: "application/xml" });
    xml = data.window.document; //set the xml to the XML DOM document which we can
    query using DOM methods
  }
  return xml;
}

async function loadParks() {
  xml = await loadXml();
  return xml.querySelectorAll("Location");
}
```

```

}

module.exports = {
  loadParks
};

```

To query with DOM methods (you can use other DOM methods as, e.g. `getElementsByTagName()` or even `getElementById()` if elements have an id attribute):

```

//replace <element_name> with an element name from XML
xml.querySelectorAll("<element_name>"); //get a list of nodes
xml.querySelector("<element_name>"); //get first matching node

```

Note that instead of querying the entire XML DOM document (`xml`), you can query a node (e.g. if you've already selected an element, you can query inside that element). For example, in the page route:

```

const parks = require("../components/parks/");
...
app.get("/", async (request, response) => {
  let data = await parks.loadParks();
  response.render("index", { title: "Home", parks: data });
});

```

Now, in the template (.pug) file, we can query parks (which is a list of `<Location>` elements). The following code is query each `<Location>` element in the loop and retrieving that `<Location>`'s `<LocationName>` element's text (`.textContent` is the generic DOM property similar to the `innerHTML` property in the HTML DOM).

```

each park in parks
  - let parkName = park.querySelector("LocationName").textContent;
  div #{parkName}

```

We can extrapolate on this and also load each Location's `<LocationID>` element.

```

each park in parks
  - let parkId = park.querySelector("LocationID").textContent;
  - let parkName = park.querySelector("LocationName").textContent;
  div
    a(href=`/park/${parkId}`) #{parkName}

```

Note that we're creating a path for the links to lead to a page in our website named `/park/[LocationID_value]` (e.g. `/park/1`).

Now, we can add a function in `/components/parks/index.js` that gets a `<Location>` by id. We can use the `evaluate()` method to use XPath. (Don't forget to include this function in the export statement.)

```

async function getParkById(id) {
  xml = await loadParks();
  let result = xml.evaluate(
    `//Location[LocationID/text()='${id}']`, xml, parkNS, 4, null);
  //expecting only one result so iterateNext() only needs to be run once
  return result.iterateNext();
}

```

Now, in your page route:

```
app.get("/park/:id", async (request, response) => {
  let parkData = await parks.getParkById(request.params.id);
  console.log(parkData);
  response.render("park", { title: "Park", park: parkData });
});
```

In the above code, notice that we can specify a page route with a placeholder. In this case, there's a placeholder named "id" in the URI. This allows us to use this code for any page with the path /park/<some\_value>. To retrieve the placeholder value, use `request.params.<placeholder_name>`.

In *park.pug*:


```
extends layout

block main-content
  - let parkName = park.querySelector("LocationName").textContent;
  - let addr = park.querySelector("Address").textContent;
  - let pcode = park.querySelector("PostalCode").textContent;
  h1 #{parkName}
  p #{addr}
  p #{pcode}
```

## Namespaces

If using XML from other sites (like you would for open data sources), the XML would be using namespaces (see arrow pointing to a namespace in the XML snippet below). The namespace is defined by an `xmlns` property and is formatted like a URL. Although it looks like a URL, its actual purpose is as an identifier. This ensures that if a site is using multiple XML files from different sources/specs, the elements are differentiated by their different namespaces. To put it simply, namespaces are used to define a **context for your elements and attributes**. The URL format is the convention because domain names are unique, so this is how you can ensure that your own namespace (if you create one) is unique.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Locations xmlns="http://www.example.org/PFRMapData">
  ...
</Locations>
```



The above snippet declares a namespace on Locations and this implicitly places all of Locations's content elements and attributes under the same namespace.

Namespaces may be explicitly declared on elements and attributes using a namespace prefix (namespace prefixes highlighted below).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Locations xmlns:map="http://www.example.org/PFRMapData">
```

```
<map:Location>
  ...
</map:Location>
</Locations>
```

DOM methods for querying/selecting like `querySelector()` should work fine even on XML using namespaces, but you need to specify the namespace if using XPath. If the XML doesn't have a namespace, you can use `null` as the namespace resolver (third parameter for `document.evaluate()`). If the XML does have a namespace, we need to create a function or variable which returns the namespace. If there are multiple, the namespaces can be listed in a JSON object. See below.

**Read:** [https://developer.mozilla.org/en-US/docs/Web/XPath/Introduction\\_to\\_using\\_XPath\\_in\\_JavaScript#implementing\\_a\\_user\\_defined\\_namespace\\_resolver](https://developer.mozilla.org/en-US/docs/Web/XPath/Introduction_to_using_XPath_in_JavaScript#implementing_a_user_defined_namespace_resolver)

For example,

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Locations xmlns="http://www.example.org/PFRMapData">
  ...
</Locations>
```

In the JS:

```
//first param: XPath expression
//second param: what you're querying
//third param: namespace, if using
//fourth param: what type of result you want returned (note: XPathResult.ANY_TYPE
returns the natural type based on the XPath expression)
//fifth param: an existing XPathResult object to use for results (using null will
return a new object)
let result = xmldocument.evaluate('//Location[1]', xmldocument, nsResolver,
XPathResult.ANY_TYPE, null);
...
function nsResolver() {
  return "http://www.example.org/PFRMapData"; //return namespace
}
```

Notice that in the XPath, for vanilla JS, **you may have to use an arbitrary namespace prefix even if it doesn't exist in the XML**. This is to force the namespace. If you don't use a prefix, you'll get an error. If you use `null` in place of the namespace resolver, you'll get an error.

Using XPath:

```
//use null instead of xmlns if no namespace
//4 is the equivalent of XPathResult.UNORDERED_NODE_ITERATOR_TYPE
//(the usual resulting type if using ANY_TYPE)
let result = xml.evaluate(<xpath_expression>, xml, xmlns, 4, null);
let resNode = result.iterateNext();
```



## Vanilla JS

If using vanilla client-side JS, you need to run your site/code in a server environment (e.g. Apache or using Live Server in VS Code).

If your site has the following directory structure:

- *site\_directory/*
  - *example.html*
  - *js/*
    - *xmlread.js*
  - *xml/*
    - *people.xml* (containing the XML from page 1)

You can link to the *xmlread.js* from the HTML either using `async defer` or place the `<script>` tag at the end of `<body>` just before the `</body>` closing tag.

In *xmlread.js*, you can read using DOM methods or by using XPath via the XPath `document.evaluate()` method, where `document` refers to the DOM document being queried. It can be an HTML DOM document or an XML DOM document. (Read documentation: [https://developer.mozilla.org/en-US/docs/Web/XPath/Introduction\\_to\\_using\\_XPath\\_in\\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/XPath/Introduction_to_using_XPath_in_JavaScript))

```
//assuming using async defer for loading this script file
loadData();

async function loadData() {
  const response = await fetch(
    "/xml/facilities-data.xml",
    {
      method: "get",
      headers: {
        "Content-Type": "application/xml"
      }
    }
  );
  let parser = new DOMParser(); //use DOMParser
  data = parser.parseFromString(await response.text(), "application/xml"); //parse
  from XML string to actual XML DOM tree
  console.log(data); //see the XML DOM object in console

  let firstLocation = data.querySelector("Location:first-child");
  console.log(firstLocation); //print the first <Location> element to console

  //test XPath
  //document.evaluate([xpath_expression], [document_or_node], [namespace_resolver],
  [result_type], [result])
  //the "a:" is just a namespace
  let result = data.evaluate('//a:Location', data, nsResolver, XPathResult.ANY_TYPE,
  null);
  //console.log(result);
  let div = document.getElementById("results"); //get div#results from the HTML DOM
  (document)
  //the result of .evaluate() is an object not an actual list
  //the way to loop through results is using the object's iterateNext() method
```

```

let contents = "";
while (loc = result.iterateNext()) {
  //console.log(loc.querySelector("LocationName").textContent);
  //console.log(loc.querySelector("Address").getAttribute("specific"));
  let locName = loc.querySelector("LocationName").textContent;
  contents += `<li>${locName}</li>`;
}
div.innerHTML = `<ul>${contents}</ul>`;
}

```

## JSON

You'll come across JSON (JavaScript Object Notation) often when looking at open data and especially when we start looking at APIs. Quite simply, JSON is just another way to store/structure data and is in the form of a JavaScript object. As a refresher, take a look at the piece of data below:

```

{
  "firstname": "John",
  "lastname": "Doe"
}

```

The above is a JSON object containing two name-value pairs. The name-value pairs are for the bits of data identified as "firstname" (with the value "John") and the "lastname" (with the value "Doe"). This looks quite similar to how objects are defined in JavaScript. Note that the name-value pairs are separated by commas and are enclosed in curly brackets. The name (*key*) is separated from its value by a comma. Note that some JSON sources may not have the names in the name-value pairs enclosed in quotes.

JSON is used quite often because it is easily read by both humans and machines because it is platform-agnostic (like XML) but is lightweight and short to write (unlike XML).

You can define an array of pieces of data as well:

```

[
  {
    "firstname": "John",
    "lastname": "Doe"
  },
  {
    "firstname": "Hermione",
    "lastname": "Granger"
  }
]

```

Notice that to define an array of { ... } objects, you wrap the array with [ ... ] and separate each { ... } object with commas.

JSON data sources are pretty straight-forward as JSON is natively supported. If loading an online JSON resource using Fetch, you'd approach it the same way we've processed the response data before (response.json()). You can still load a file in your site directory using Fetch if you put it in your /public folder.

```

var cameras;

async function loadCameras() {
  if (cameras == undefined) {
    let response = await fetch(
      "http://localhost:8888/cameras.geojson",
      {
        method: "get",
        headers: {
          "Content-Type": "application/json"
        }
      }
    );
    cameras = await response.json();
    console.log(cameras.features[0].properties.MAINROAD);
  }
  return cameras;
}

module.exports = {
  loadCameras
};

```