

Front-End Coding: JavaScript and GeoJSON

Bo Zhao

The Department of Geography

Outline

- JavaScript
- GeoJSON

JavaScript: A high-level definition

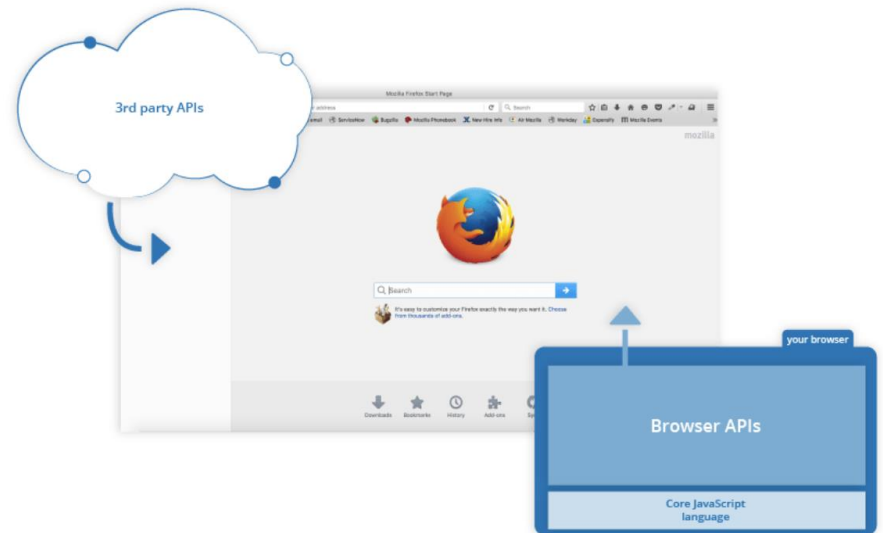
JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved. It is the third layer of the layer cake of standard web technologies, two of which (HTML and CSS) we have covered in much more detail in other parts of the Learning Area.

Interpreted versus compiled code

- You might hear the terms interpreted and compiled in the context of programming. In interpreted languages, the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it. The code is received in its programmer-friendly text form and processed directly from that.
- Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer, such as C/C++.
- **JavaScript is a lightweight interpreted programming language.** The web browser receives the JavaScript code in its original text form and runs the script from that. From a technical standpoint, most modern JavaScript interpreters actually use a technique called just-in-time compiling to improve performance; the JavaScript source code gets compiled into a faster, binary format while the script is being used, so that it can be run as quickly as possible. However, JavaScript is still considered an interpreted language, since the compilation is handled at run time, rather than ahead of time.

Application Programming Interfaces (APIs)

- What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called Application Programming Interfaces (APIs) provide you with extra superpowers to use in your JavaScript code.
- APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement.
- Two categories: Browser APIs and Third-party APIs



Browser APIs

built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things.

- The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed (as we saw above in our simple demo) for example, that's the DOM in action.
- The Geolocation API retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.
- The Canvas and WebGL APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies —see Chrome Experiments and [webgl.samples](#).
- Audio and Video APIs like `HTMLMediaElement` and WebRTC allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer (try our simple Snapshot demo to get the idea).

Third party APIs

built into the browser by default, and you generally have to grab their code and information from somewhere on the Web.

- The Twitter API allows you to do things like displaying your latest tweets on your website.
- The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example,

```
const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {
  let name = prompt('Enter a new name');
  para.textContent = 'Player 1: ' + name;
}
```

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the browser developer console — `TypeError: para is undefined`. This means that the `para` object does not exist yet, so we can't add an event listener to it.

Two hands-on experiments

- Guess the number game
- Image gallery

This two examples are from MDN Web Docs.

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Image_gallery

GeoJSON {}

JSON

- JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax.
- It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).
- JSON exists as a string — useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. This is not a big issue — JavaScript provides a global JSON object that has methods available for converting between the two.
- A JSON string can be stored in its own file, which is basically just a text file with an extension of .json, and a MIME type of application/json.

{JSON}

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

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.

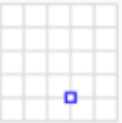
GeoJSON

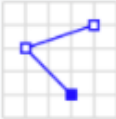

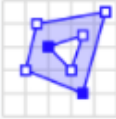
- GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes. It is based on the JSON format.
- The features include points (therefore addresses and locations), line strings (therefore streets, highways and boundaries), polygons (countries, provinces, tracts of land), and multi-part collections of these types. GeoJSON features need not represent entities of the physical world only; mobile routing and navigation apps, for example, might describe their service coverage using GeoJSON.

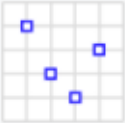

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
      },
      "properties": {
        "prop0": "value0"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
            [100.0, 1.0], [100.0, 0.0]
          ]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": { "this": "that" }
      }
    }
  ]
}
```

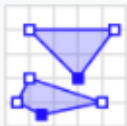
Geometries

- Points are [x, y] or [x, y, z]. They may be [longitude, latitude] or [eastings, northings]. Elevation is an optional third number. They are decimal numbers.
- For example, London (51.5074° North, 0.1278° West) is [-0.1278, 51.5074]

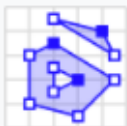
Point		<pre>{ "type": "Point", "coordinates": [30.0, 10.0] }</pre>
-------	---	---

<div data-bbox="46 287 197 322" data-label="Text"> <p>LineString</p> </div>		<pre data-bbox="401 191 1263 429">{ "type": "LineString", "coordinates": [[30.0, 10.0], [10.0, 30.0], [40.0, 40.0]] }</pre>
<div data-bbox="46 915 166 951" data-label="Text"> <p>Polygon</p> </div>		<pre data-bbox="401 582 1798 821">{ "type": "Polygon", "coordinates": [[[30.0, 10.0], [40.0, 40.0], [20.0, 40.0], [10.0, 20.0], [30.0, 10.0]]] }</pre>
		<pre data-bbox="401 972 1673 1296">{ "type": "Polygon", "coordinates": [[[35.0, 10.0], [45.0, 45.0], [15.0, 40.0], [10.0, 20.0], [35.0, 10.0]], [[20.0, 30.0], [35.0, 35.0], [30.0, 20.0], [20.0, 30.0]]] }</pre>

Type	Examples	
MultiPoint		<pre data-bbox="504 358 1895 682"> { "type": "MultiPoint", "coordinates": [[10.0, 40.0], [40.0, 30.0], [20.0, 20.0], [30.0, 10.0]] }</pre>
MultiLineString		<pre data-bbox="504 765 1895 1089"> { "type": "MultiLineString", "coordinates": [[[10.0, 10.0], [20.0, 20.0], [10.0, 40.0]], [[40.0, 40.0], [30.0, 30.0], [40.0, 20.0], [30.0, 10.0]]] }</pre>



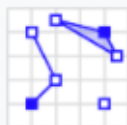
```
{  
  "type": "MultiPolygon",  
  "coordinates": [  
    [[30.0, 20.0], [45.0, 40.0], [10.0, 40.0], [30.0, 20.0]],  
    [[15.0, 5.0], [40.0, 10.0], [10.0, 20.0], [5.0, 10.0],  
    [15.0, 5.0]]  
  ]  
}
```



```
{  
  "type": "MultiPolygon",  
  "coordinates": [  
    [[40.0, 40.0], [20.0, 45.0], [45.0, 30.0], [40.0, 40.0]],  
    [[20.0, 35.0], [10.0, 30.0], [10.0, 10.0], [30.0, 5.0],  
    [45.0, 20.0], [20.0, 35.0]],  
    [[30.0, 20.0], [20.0, 15.0], [20.0, 25.0], [30.0, 20.0]]  
  ]  
}
```

MultiPolygon

GeometryCollection



```
{
  "type": "GeometryCollection",
  "geometries": [
    {
      "type": "Point",
      "coordinates": [40.0, 10.0]
    },
    {
      "type": "LineString",
      "coordinates": [
        [10.0, 10.0], [20.0, 20.0], [10.0, 40.0]
      ]
    },
    {
      "type": "Polygon",
      "coordinates": [
        [[40.0, 40.0], [20.0, 45.0], [45.0, 30.0], [40.0,
40.0]]
      ]
    }
  ]
}
```

TopoJSON

- A notable offspring of GeoJSON is TopoJSON, an extension of GeoJSON that encodes geospatial topology and that typically provides smaller file sizes.
- Rather than representing geometries discretely, geometries in TopoJSON files are stitched together from shared line segments called *arcs*.
- *Arcs* are sequences of points, while line strings and polygons are defined as sequences of arcs. Each arc is defined only once, but can be referenced several times by different shapes, thus reducing redundancy and decreasing the file size.
- In addition, TopoJSON facilitates applications that use topology, such as topology-preserving shape simplification, automatic map coloring, and cartograms.

Hands-on experiments

- Create GeoJSONs on geojson.io
- Convert a shapefile to geojson on QGIS
- Using mapshaper to simplify a shapefile