Tom MacWright

tom@macwright.org

More than you ever wanted to know about GeoJSON

Let's look at GeoJSON in a little more depth, from the ground up. Understanding these concepts will help you understand geospatial data in general, too: the basic concepts behind GeoJSON have been a part of geo since the very beginning.

This should be read along with the GeoJSON spec itself, which is authoritative and, for a formal specification of a format, pretty readable.

Coordinate

The most basic element of geographic data is the **coordinate**. This is a single number representing a single dimension: typically the dimensions are longitude and latitude. Sometimes there's also a coordinate for elevation. Time is a dimension but usually isn't represented in a coordinate because it's too complex to fit in a number.

Coordinates in GeoJSON are formatted like numbers in JSON: in a simple decimal format. Unlike geographic data for human consumption, data formats never use non-base-10 encodings like sexagesimal. As cool as 8° 10' 23" looks, it's just not a very good way to tell numbers to computers.

Position

A position is an array of coordinates in order: this is the smallest unit that we can really consider 'a place' since it can represent a point on earth. GeoJSON describes an order for coordinates: they should go, in order:

```
[longitude, latitude, elevation]
```

This order can be surprising. Historically, the order of coordinates is usually "latitude, longitude", and many people will assume that this is the case universally. Long hours have been wasted discussing which is better, but for this discussion, I'll summarize as such:

- longitude, latitude matches the X, Y order of math
- data formats usually use longitude, latitude order
- applications have tended to use latitude, longitude order

Here's a handy chart of what uses which ordering.

Before the current specification was released, GeoJSON allowed the storage of more than 3 coordinates per position, and sometimes people would use that to store time, heart rate, and so on. This wasn't well supported in GeoJSON tools, and is forbidden by the new specification.

Geometry

Geometries are shapes. All simple geometries in GeoJSON consist of a `type` and a collection of `coordinates`.

Points

```
- Point
  - Position
```

With a single position, we can make the simplest geometry: the **point**.

```
{ "type": "Point", "coordinates": [0, 0] }
```

Depending on the type, the kind of collection of coordinates differs: let's see how.

LineStrings

To represent a **line**, you'll need at least two places to connect:

```
{ "type": "LineString", "coordinates": [[0, 0], [10, 10]] }
```

```
- LineString
  - Positions..
```

These are the two simplest, most carefree kinds of geometry. Points and LineStrings don't have many geometric rules: a point can lie anywhere in a place, and a line can contain arrangement of points, even if it's self-crossing. Points and lines are also similar in that they **have no area**: there is no inside or outside.

Polygons

Polygons are where GeoJSON geometries become significantly more complex. They have area, so they have insides & outsides. And not only do they have an inside, they can also have holes in that inside.

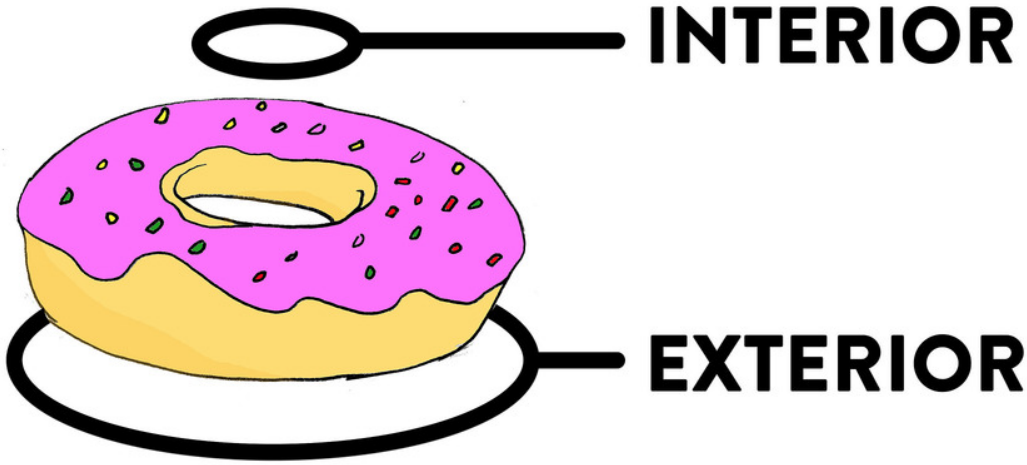
```
{
  "type": "Polygon",
  "coordinates": [
```

```
      [
        [0, 0], [10, 10], [10, 0], [0, 0]
      ]
    ]
  }
```

The list of `coordinates` for Polygons is nested one more level than that for LineStrings. For simple polygons, this might seem like overkill: what is a polygon but a closed line? But **holes** explain the jump in complexity: polygons in GeoJSON are not just closed areas, but can have cut-outs like donuts.

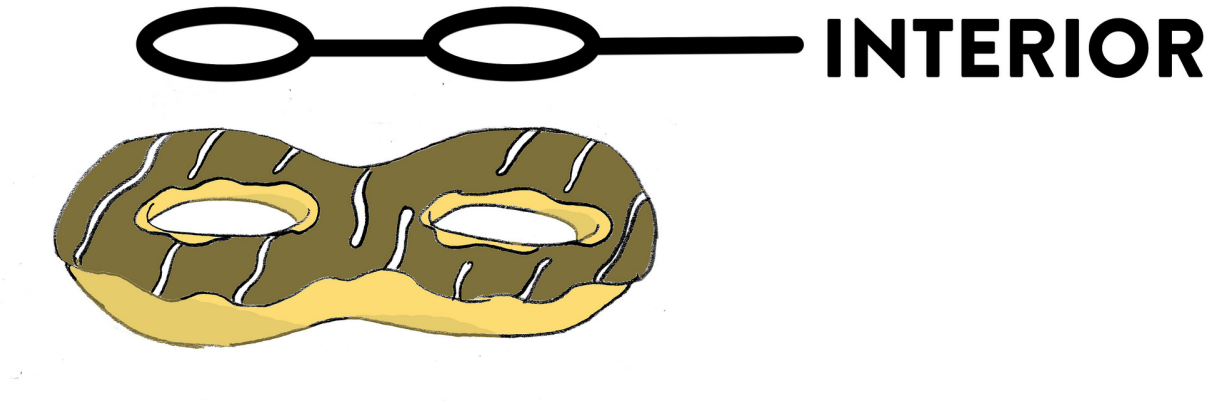
```
- Polygon
  - LinearRing (exterior)
    - Positions..
  - LinearRing (interior)
    - Positions...
  - LinearRing (interior)
    - Positions...
```

For this reason, polygons introduce a new term: the **LinearRing**. LinearRings are loops of positions.



LinearRings are either the **exterior** ring - positions that form the outside edge of the Polygon and define which parts are filled - or **interior** rings, which define the parts of the Polygon are empty. There

can only be one exterior ring, and it's always the first one.



There can be any number of interior rings, including zero. Zero interior rings just means that the polygon doesn't have any holes.

You'll also notice that the **first coordinate is repeated at the end** of each ring. There's no particular reason why this is necessary besides GeoJSON's heritage in older formats.

Coordinate Deepness

In this fun exploration, you may have noticed that there are four 'levels of depth' for the `coordinates` property of GeoJSON.

1. Points
2. MultiPoints & LineStrings
3. MultiLineStrings & Polygons
4. MultiPolygons

Features

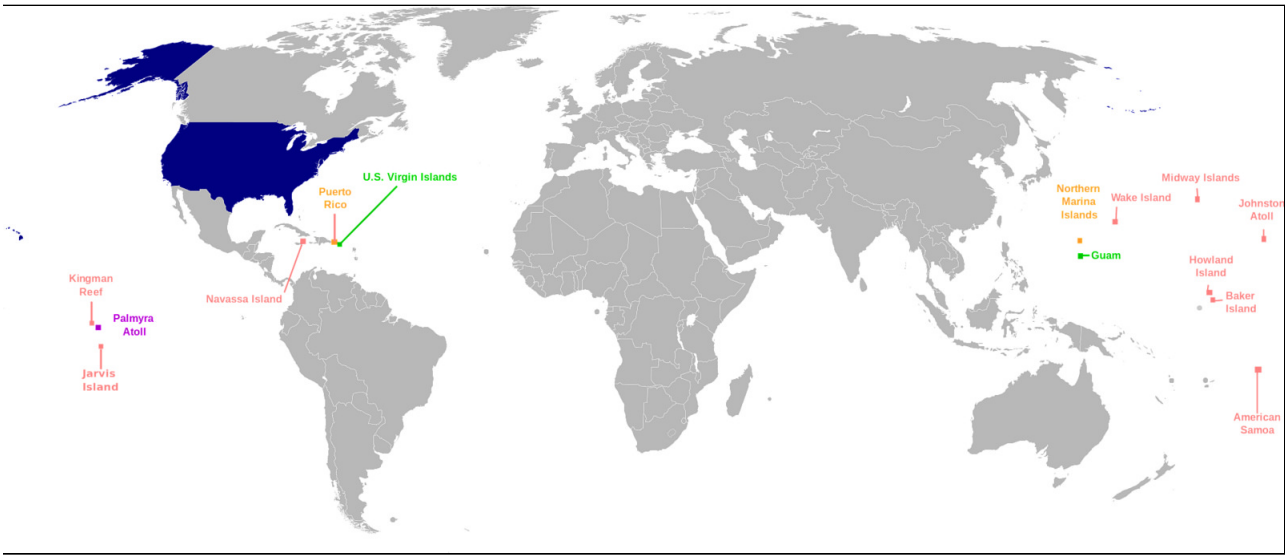
Geometries are shapes and nothing more. They're a central part of GeoJSON, but most data that has something to do with the world isn't simply a shape, but also has an identity and attributes. Some polygons are the White House, other polygons are the border of Australia, and it's important to know which is which.

Features are this combination of geometry and properties.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [0, 0]
  },
  "properties": {
    "name": "null island"
  }
}
```

The `properties` attached to a feature can be any kind of JSON object. That said, given the fact that no other prominent geospatial standard supports nested values, usually the properties object consists of single-depth key⇢ value mappings.

Multi Geometries



Now that we're talking about how data can describe the world, you might notice some limitations of this approach. Each of the basic `LineString`, `Polygon`, `Point` types is great for representing a single shape, but often the physical world contains entities that aren't just a single contiguous thing. For instance, the United States, along with many other countries, has multiple disconnected parts. We refer to all of them as "The United States", and software that wants to highlight "The United States" should be able to know this and also highlight Alaska, Hawaii and the rest.

This is where Multi Geometries come in. GeoJSON has versions of each of the three basic types with `Multi` stuck on the front: `MultiPolygon`, `MultiLineString`, `MultiPoint`. Together they give us something of a solution for this problem.

The way Multi features are created is the same across all the types: everything moves down a step of nesting. The coordinates of a single point are represented as `[0, 0]`, so a `MultiPoint` of that and another place might look like `[[0, 0], [1, 1]]`.

In rarer cases, you'll have a bunch of *different kinds of geometries* that all refer to the same thing. For that, GeoJSON has the `GeometryCollection` type, which works like this:

```
{
  "type": "Feature",
  "geometry": {
    "type": "GeometryCollection",
    "geometries": [{
      "type": "Point",
      "coordinates": [0, 0]
    }, {
      "type": "LineString",
      "coordinates": [[0, 0], [1, 0]]
    }]
  },
  "properties": {
    "name": "null island"
  }
}
```

GeometryCollections are relatively rare: most of the time when you have geometries of different types, you'll also have properties that will specifically apply to the individually. The current GeoJSON specification recommends against using GeometryCollections.

FeatureCollection

We've covered all the kinds of things that can be in GeoJSON but one: `FeatureCollection` is the most common thing you'll see at the top level of GeoJSON files in the field.

A `FeatureCollection` containing our "null island" example of a `Feature` looks like:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [0, 0]
      },
      "properties": {
        "name": "null island"
      }
    }
  ]
}
```

`FeatureCollection` is not much more than an object that has `"type": "FeatureCollection"` and then an array of Feature objects under the key `"features"`. As the name suggests, the array needs to contain Feature objects only - no raw geometries.

You might ask “why not just permit an array of GeoJSON objects”? FeatureCollections as objects makes a lot of sense in terms of the commonality between different GeoJSON types.

- GeoJSON objects are Objects, not Arrays or primitives
- GeoJSON objects have a `"type"` property

This is really nifty for implementations: they don’t need to guess about what kind of GeoJSON object they’re looking at - they just read the “type” property.

Winding

UPDATE: RFC 7946 GeoJSON now recommends right-hand rule winding order

LineString and Polygon geometries contain coordinates in an order: lines go in a certain direction, and polygon rings do too.

The direction of LineString often reflects the direction of something in real life: a GPS trace will go in the direction of movement, or a street in the direction of allowed traffic flows.

Polygon ring order is undefined in GeoJSON, but there’s a useful default to acquire: the **right hand rule**. Specifically this means that

- The exterior ring should be counterclockwise.
- Interior rings should be clockwise.

Why care? There are roughly two practical reasons:

1. The classic Chamberlain & Duquette algorithm for calculating the area of a polygon on a sphere has the nice property that counterclockwise-wound polygons have positive area and clockwise yield negative. If you ensure winding order, calculating the area of a polygon with holes is as simple as adding the areas of all rings.
2. Winding order also has a default meaning in Canvas and other drawing APIs: drawing a path with counterclockwise order within one with clockwise will cut it out of the filled image.

The 180th Meridian

The 180th meridian is one of the shames of geospatial technology. The story goes that given the rules of

- LineStrings and Polygons are represented as collections of positions
- Positions should be within -180° and 180° longitude and -90° and 90° latitude

It is simply impossible to tell the difference between a line that goes from -179° around the world to 179°, or one that just hops over the 180th meridian. That's one problem with Cartesian coordinates on a sphere.

A popular way to represent these lines is to break the second rule: a line that crosses the 180th meridian would be represented as 179° to 181° instead of 179° to -179°. By some definitions, this is invalid: 181° is out of the range of the EPSG:4326 datum. But most modern map technology tolerates this kind of data and helpfully draws the image you'd expect.

There's a clear need for a cleverer and cleaner solution to the 180th meridian problem: both at zero and at the dateline, even the most sophisticated tools exhibit eccentricities and bugs. The most

promising option in my opinion is delta-encoding, like in TopoJSON and Geobuf. Instead of representing coordinate pairs as in their full form, delta-encoded geodata will save a line as a series of directional steps: starting from `-73, 38`, it would say to move by `-3, -3`, instead of specifying that the next coordinate is `-76, 35`. Perhaps this gives a clear way to differentiate meridian wrapping from world-sized jumps without breaking the rules of a datum. But that's just a guess.

I wrote a whole article about the 180th meridian if you'd like to dig in even more.

What you can't do with GeoJSON

Much of GeoJSON's popularity derives from its simplicity, which makes it easy to implement, read, and share. So, like every other format, it has its limits.

- GeoJSON has no construct for **topology**, whether for compression, like TopoJSON, or semantics, like OSM XML and some proprietary formats. A topological layer on top of GeoJSON is possible but unimplemented.
- GeoJSON features have properties, which are JSON. Properties can use any of the JSON datatypes: numbers, strings, booleans, null, arrays, and objects. JSON doesn't support every data type: for instance, date values are supported by Shapefiles, but not in JSON.
- GeoJSON doesn't have a construct for styling features or specifying popup content like title & description. There are folk conventions for this, like simplestyle-spec and Leaflet's Path properties, but these aren't and won't be part of the spec. Most geo formats don't have styling support included either - KML stands out as prioritizing styling.
- GeoJSON doesn't have a circle geometry type, or any kind of curve. Only a few formats, like WKT, support curves and circles rather than straight-line geometries. Circles & curves are relatively tricky to implement, because a circle on a spheroid geoid planet is much more complex than a circle on a sheet of paper.
- Positions don't have attributes. If you have a LineString representation of a run, and your GPS watch logged 1,000 different points along that run, along with your heart rate and the duration at that time, there's no clear answer for how to represent that data. You can store additional data in positions as fourth and fifth coordinates, or in properties as an array with the same length as the coordinate array, but neither option is well-supported by the ecosystem of tools. The Simple Features Specification, which directly inspired GeoJSON and most GIS formats, doesn't support this notion of attributes-at-positions, and only two formats - GPX & OSM XML - do.

Projections

UPDATE: *2008 geojson.org GeoJSON supported alternative coordinate reference systems other than ESPG:4326, but this capability was removed in the current GeoJSON standard. So, this section is of historical interest but you shouldn't use the `crs` member or try to put projected data into GeoJSON: you should instead reproject it to WGS84 first.*

Anyway, here's what it looked like:

```
{ "type": "Point",
  "coordinates": [100.0, 0.0],
  "crs": {
    "type": "link",
    "properties": {
      "href": "http://example.com/crs/42",
      "type": "proj4"
    } } }
```

However, tools that interact with GeoJSON often disregard this feature, and the IETF draft specifically advises against using the `crs` property.

While there are other formats that support projections explicitly and have ecosystems with more focus on alternative CRSes, there are a few important things to remember in terms of projections.

projections in data are variously referred to as SRS, CRS, and just 'projections' with pedantic and poorly enumerated differences. consider the terms equivalent below

Map projections are not coordinate reference systems. You can rally against Web Mercator or Plate Carée, but that's entirely irrelevant to projections in data. Data can be stored in any projection and displayed in any other projection by the magic of *reprojection*, done seamlessly by libraries like proj4 that are integrated into virtually all tools. For instance, OpenStreetMap is typically displayed in Web Mercator, but is stored in EPSG:4326. By the magic of reprojection, you can render OpenStreetMap in any other projection.

Reprojection precision loss is real but tiny. If you're a surveyor and used a theodolite to determine a geographical position in centimeters relative to a landmark, and come up with a value in a state plane coordinate system, it's likely that you don't want to - and shouldn't - store your data in `EPSG:4326`. That's because computer calculations are typically fixed-point: instead of dividing 1 by 3 and getting ⅓ like you did in arithmetic, computers have a fixed number of decimal places - so most calculations are just slightly off from the absolute value.

Data projections are friction. If you aren't a surveyor and don't actually have centimeter-accuracy data, using projections adds friction for users: instead of simply downloading and using data, they need to determine the projection - sometimes manually - and occasionally even need to load in new projection definitions in order to use it. And they usually, off the bat, just convert it to `EPSG:4326`.

So, the take-home lesson of data projections is that they're useful for extremely-high precision datasets. But such data is rare, and usually the GeoJSON default of `EPSG:4326` is a better choice for sharing and storing data.

Performance

I've heard it said that GeoJSON isn't as efficient as binary formats like Shapefiles, or fancier-encoded formats like TopoJSON, or that you should always use PostGIS. Performance of formats and internet software is generally misunderstood and oversimplified. I don't have enough space or knowledge to cover all of it, but here are at least a few thoughts that might be enlightening.

Your first focus in terms of performance should always be bottlenecks. For instance, if you have a classic GeoJSON + Leaflet setup and performance issues, the bottleneck is almost always network or SVG. If it's SVG performance - the cost of Leaflet drawing polygons and lines in your browser - then the file format is irrelevant. Transfer the same data in an ultra-efficient format and you'll still end up with a slow map.

Let's say that network time is the bottleneck: the GeoJSON file is 20MB and takes 20 seconds to load. The approaches to solving that kind of issue are more general than any kind of file format:

Lossy compression

These tricks are employed by tools like simplify.js, TopoJSON, and Geobuf.

- **Removing attributes**: often GeoJSON data (and data in general) contains columns that are unused.
- **Quantization** means reducing the precision of coordinates in your data to a certain level that isn't noticeable on the map.
- **Simplification** will eliminate details that aren't visible at reasonable zooms - removing coordinates from LineStrings and Polygons that are super close together.

Lossy compression techniques are generally orthogonal to file formats: removing attributes and simplifying geometries will give you some performance savings, regardless of whether the data's a

Shapefile, GeoJSON, or something else.

Loading subsets

Maps and analysis typically display or analyze a fraction of the total dataset at a time. By tiling or implementing a protocol like WFS, you can specify smaller bites of data at a time, saving time.

The subsets approach generally *implies some sort of lossy compression*: if you don't compromise accuracy, zooming out on the map will load all of the data, yielding the same performance story you started out with. So things like the Mapbox Vector Tile spec specifies not just how to cut up data, but also how to simplify it.

File formats can support the trick of loading subsets by including *indexes*. Indexes like R* Trees and cells make it possible to efficiently query a file for a specific geographical area, rather than having to look at each feature in succession. While it's possible to index GeoJSON, there are no popular implementations, and their usefulness would be limited to Content-Range support in web servers, which is quite limited.

Streaming

Running an analysis across a gigantic dataset, like the 550GB+ OpenStreetMap Planet which you don't want load into memory, requires **streaming**. In a nutshell, streaming is a technique in which software will read datasets item-by-item, only keeping a tiny fraction of it in memory at a time.

Some formats are very amenable to streaming, like CSV, which you only need to split by newlines to process in this fashion. XML, awkward as it is, is also gifted with a number of high-quality streaming parsers that make streaming parsing of OSM XML doable.

While it's somewhat possible to parse GeoJSON with streams, it has a few drawbacks relative to some other formats:

- The order of properties in GeoJSON isn't defined: the "properties" part of a feature could come before or after the "id" and so on with every other part.
- JSON requires a single root object: you can't just "write a bunch of GeoJSON Features to a file" and be done with it: they would need to be wrapped in an array, in a FeatureCollection.

In other words, streaming benefits from simple separators between entries in data and well-defined types and orders. GeoJSON isn't perfect in this regard, due to its JSON lineage, but there's plenty of room to improve by taking advantage of the LD-JSON spec that proposes line-delimited JSON.

End notes

If you're diving into the technology around GeoJSON, I've compiled a list of utilities that convert, process, and analyze GeoJSON data. To tinker with GeoJSON and see how it relates to geographical features, try geojson.io, a tool that shows code and visual representation in two panes.

March 23, 2015 @tmcw

More things I've written about GeoJSON:

- Everything you need to know now about RFC 7946 GeoJSON
- Falsehoods developers believe about GeoJSON