LECTURE 6: DATA FORMATS AND MAPPING DATA

Reading Takeaways

- OpenStreetMap is awesome
- Really you should mess around with it
- Maps are important, change how we look at things
- Lots of data out there
- Get people involved
- Is Spatial Special?

What is data?

- What you or someone else decided was important to collect for a specific purpose
- Constrained by time and cost
- Constrained by the limitations of your sensor
- Unique snowflake

DATA FORMATS

Binary

- 0 and 1
- Data read directly from sensors is binary

CSV

- Comma-separated values
- Each record is one line of text
- Individual fields are delimited by a comma
- Other similar file formats change the delimiter, using a tab, space, or other symbol instead of a comma

CSV example

id,latitude,longitude,temperature,light

1,44.45,-123.05,23.0,300.0

2,44.46,-123.06,25.0,400.0

3,44.47,-123.07,26.0,200.0

XML

- Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable
- Consists mainly of the following constructs:
  - Tags: construct that begins with < and ends with >
    - start-tags <section>
    - end-tags </section>
    - empty-element tags <line-break />
  - Element
    - A logical document component which either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag
    - Data between start and end tag is the element's content
  - Attribute
    - A markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag
    - <record **id="3" latitude="44.47" longitude="-123.07"**>

XML example

```
<records>
  <record>
    <id>1</id>
    <latitude>44.45</latitude>
    <longitude>-123.05</longitude>
    <temperature>23.0</temperature>
    <light>300.0</light>
  </record>
  <record>
    <id type="int">2</id>
    <latitude type="double">44.46</latitude>
    <longitude type="double">-123.06</longitude>
    <temperature type="double">25.0</temperature>
    <light type="double">400.0</light>
  </record>
  <record id="3" latitude="44.47" longitude="-123.07" temperature="26.0" light="200.0" />
</records>
```

JSON

- JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs.
- JSON's basic types are:
  - Number - “numberType”: 5000
  - String - “stringType”: “I am a piece of text”
  - Boolean - “booleanType”: true
  - Array - “arrayType”: [1, 5, 10, “hello”, false]
  - Object - “objectType”: { “objectProperty1”: “text”, “objectProperty2”: 300 }
  - null — “nullType”: null

Json example

{ “records”: [{ "id": 1,

"latitude": 44.45,

"longitude": -123.05,

"temperature": 23.0,

"light": 300.0},

{ "id": 1,

"latitude": 44.46,

"longitude": -123.06,

"temperature": 25.0,

"light": 400.0},

{ "id": 1,

"latitude": 44.47,

"longitude": -123.07,

"temperature": 26.0,

"light": 200.0

}]}

GeoJSON

- JSON format for encoding a variety of geographic data structures
- Supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection
- Also supports features which are a geometry object with additional properties

GeoJson Point

- Type = “Point”
- Coordinates property consists of 1 coordinate pair
- {“type”: “Point”, “coordinates”: [-123.05, 44.45]}

GeoJson LineString

- Type = “LineString”
- Coordinates property is made up of an array of coordinate pairs
- {“type”: “LineString”, “coordinates”: [[-123.05, 44.45], [-123.06, 44.46], [-123.07, 44.47]]}
- The MultiPoint type has the same coordinates format but the type is “MultiPoint”

GeoJson Polygon

- Type = “Polygon”
- Coordinates property is made up of an array of an array of coordinate pairings
- The outer most array is contains at least one entry: the exterior ring. Other entries are interior rings that are cut out of the exterior ring.
- Each ring must have the same coordinate pair for the first and last entry.
- {“type”: “Polygon”, “coordinates”:[[[-123.05, 44.45], [-123.06, 44.46], [-123.07, 44.47], [-123.05, 44.45]]]
- The MultiLineString type has the same coordinates format as the polygon type except that each entry in the outer array consists of a line string and the first and last entry in the each line string array should not match
- The MultiPolygon type adds another outer array with each entry being a Polygon

GeoJson Feature

- The Feature type adds attributes called properties to each geometry
- Type = “Feature”
- Each feature must contain a geometry property and a properties property.
- The properties property is an anonymous object (key-value dictionary)
- {“type”: “Feature”, “geometry”: {“type”: “Point”, “coordinates”: [-123.05, 44.45]}, “properties”: { “name”: “Condon”, “yearBuilt”: 1925, “numberOfFloors”: 4, “height”: 45.0}}

GeoJson FeatureCollection

- A collection of features
- Type = “FeatureCollection”
- Must contain a property called features that is an array
- {“type”: “FeatureCollection”, “features”:[{ “type”: “Feature”, “geometry”: { “type”: “Point”, “coordinates”: [-123.05, 44.45], “properties”: {“name”: “Condon”}}, { “type”: “Feature”, “geometry” { “type”: “Point”, “coordinates”: [-123.06, 44.46], “properties” {“name”: “Lillis”}}]}

Data transformation

- Conversion of a set of data values from one data format to another
- Examples:
  - Removal of values
  - Addition of values
  - Flattening of data
  - Normalization of data

What was done in the run keeper example

- Get permission from user to access their data
- Access user's activities
- When user clicks the CSV link, the server accesses that activity (format is JSON) and then transforms into CSV format using a subset of attributes
- When user click the GeoJSON link, the server accesses that activity (format is JSON) and then transforms into GeoJSON format using a subset of attributes. The result is a FeatureCollection which contains multiply Point geometries and one LineString geometry made up of each point

Fitness Activity

- Distance array - distance, timestamp
- Path array - altitude, latitude, type, timestamp, longitude
- Other properties

Example

Convert to CSV

```
private List<FitnessActivityCsvDto> ConvertToCsv(FitnessActivity activity, int activityId)
        {
            var id = 1;
            var result = new List<FitnessActivityCsvDto>();
            foreach (var path in activity.Paths)
            {
                result.Add(new FitnessActivityCsvDto
                {
                    Altitude = path.Altitude,
                    ActivityId = activityId,
                    Id = id,
                    Latitude = path.Latitude,
                    Longitude = path.Longitude,
                    Timestamp = path.Timestamp,
                    Type = path.Type
                });
                id++;
            }
            return result;
        }
```

Convert to GeoJSON

```
private string ConvertToGeoJson(FitnessActivity activity, int activityId)
        {
            var id = 1;
            var geojson = new JObject();
            geojson.Add("type", "FeatureCollection");
            var features = new JArray();
            var lineCoordinates = new List<double[]>();
            foreach (var path in activity.Paths)
            {
                var point = new JObject();
                point.Add("type", "Point");
                point.Add("coordinates", JToken.FromObject(new double[]{path.Longitude, path.Latitude}));
                var feature = new JObject();
                feature.Add("type", "Feature");
                feature.Add("geometry", point);
                feature.Add("properties", JToken.FromObject(new Dictionary<string, object>
                {
                    {"Altitude", path.Altitude},
                    {"Id", id},
                    {"Timestamp", path.Timestamp},
                    {"Type", path.Type}
                }));
                lineCoordinates.Add(new double[]{path.Longitude, path.Latitude});
                features.Add(feature);
                id++;
            }
            var lineFeature = new JObject();
            lineFeature.Add("type", "Feature");
            var lineGeometry = new JObject();
            lineGeometry.Add("type", "LineString");
            lineGeometry.Add("coordinates", JToken.FromObject(lineCoordinates));
            lineFeature.Add("geometry", lineGeometry);
            var lineStringAttributes = new JObject();
            lineStringAttributes.Add("activityId", activityId);
            lineFeature.Add("properties", lineStringAttributes);
            features.Add(lineFeature);
            geojson.Add("features", features);
            return geojson.ToString(Formatting.Indented);
        }
```

Mapping Data

Evolution of web maps

- Single image, each zoom or pan requests new image and page reload (HTML)
- Traditional slippy map (Google), multiple images arrayed in a grid (usually 256x256 pixels). As the user pans or zooms around the map, images are asynchronously downloaded from server and displayed (JavaScript)
- Vector-based map (Google, Apple), geometries are downloaded from server and drawn client-side. This is how the native apps from Google and Apple now work on smart phones. (WebGL, OpenGL ES)

Map tiles (base map)

- Images are prebuilt (cached) on the server
- All tiles within an extent are created and stored on the server for all supported zoom levels
- Fast
- Depending on support map extent, large amounts of storage is needed
- For each zoom level that is supported, the storage needed quadruples

Other map image data

- Operational layers (data that changes regularly)
- Slower, not pre-created
- Images created on server from source data and styling on the fly and sent to client
- Overlaid on top of basemap tiles

Other vector data

- Retrieve actual geometries and attributes from a data source
- Draw geometries on the client by converting the real world coordinates into pixel coordinates
- Styling of features is done client
- The styling is usually built into the client application and fairly static but might be driven dynamically from a server source or included in the feature attributes

Example using SensorTag

Client - iPhone

- Search using Bluetooth for SensorTag
- Connect to SensorTag
- Discover services
- Connect to services that we are interested in, temperature and light sensor using unique identifier
- Discover characteristics of service
- Enable sensor data and value change notification
- Read binary data from sensor
- If needed, calculate result
- Upload data to server as JSON every second

Server

- Receive data from client
- Deserialize data from JSON to native object
- Store object to database
- Using redis as database
- Store each item in a sorted set using the timestamp as the “score”
- This makes it easy to retrieve time ranges
- Server exposes a “GET” endpoint to retrieve the last sent data

Client - Web

- Load map using leaflet
- Query web server every second for data
- Parse data
- Add point to map as a Leaflet marker
- On click of marker should data
- Also update text label of temperature and light