MONGODB GEOLOCATION QUERY EXAMPLES

Here we explain how to query and working with geolocation data in MongoDB.

(This article is part of our MongoDB Guide. Use the right-hand menu to navigate.)

GeoJSON is a universally-accepted standard to represent a location by its latitude and longitude, aka coordinates. MongoDB uses the same syntax as the US Geological Survey or other persons working with maps would use. For example, a simple point is represented like this:

```
"location" : {
                "type" : "Point",
                "coordinates" :
        }
```

On the globe, longitude < 0 mean west of Greenwich, England, by convention. Numbers > 0 are to the east. To remember which is which, think of longitude lines as being **long** since they go all the way from the North to the South Pole. This is an arc of 180 degrees. By convention is is divided into two arcs of 90 degrees each. North is positive and South negative.

The latitude is the circle that goes a full 360 degrees around the Earth. By convention that ranges between -180 (West) and 180 (East) degrees.

The equator is the point where the latitude is 0.

Note that you can use Google Maps and point to a location on a map and right click the mouse and select **What's Here** and it gives the longitude and latitude. But it gives it backwards from convention, giving the latitude first and the longitude second.

Note also that while sailors use degrees (°), minutes ("), and seconds ('), based on the number 360, GeoJSON uses the base 10 decimal system, which is obviously easier to deal with when doing arithmetic.

With GeoJSON you can also represent an area, rather than a point, by drawing any number of line segments around it like this, which is called a **Polygon**, although this does not mean it has to have 5 sides, as does a Polygon is university geometry classes. Each pair of coordinates in a GeoJSON polygon is a **line segment**.

Like this:

```
location: {
      type: "Polygon",
       coordinates:
 ,
   ,
   ,
   ,
   ]]
}
```

MongoDB is well suited to store Geolocation data because it provides points, polygons, and other Geolocation objects as well as built-in query methods to query data based on its proximity to a point on a map, within a certain distance of that point, or within lines drawn around that location.

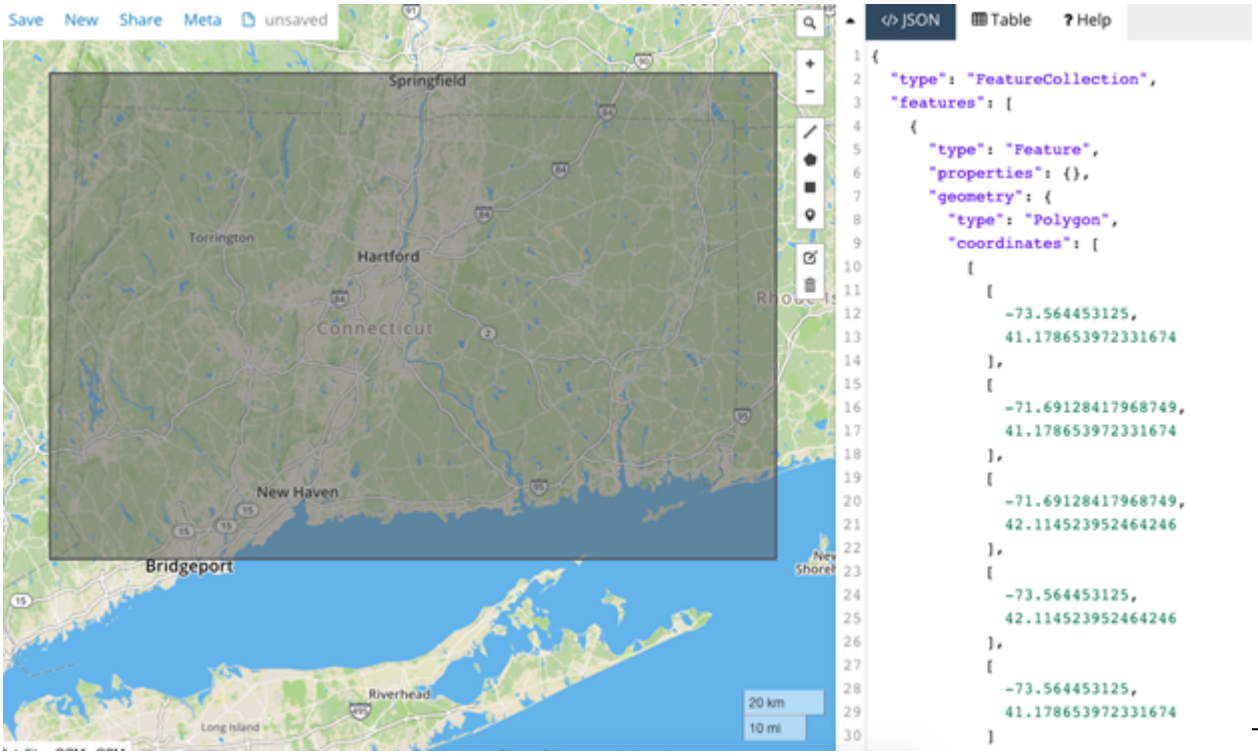
We illustrate that by using data provided by the Town of Fairfield and other towns in Connecticut and assembled by OpenAddresses. (The full license is here.) Download the data from here. Then unzip it:

```
tar xvfz ct.json.tar
```

Then load it into a MongoDB instance like this, where **geo** is the database and **address** is the collection:

```
mongoimport --db geo --collection address --file ct.csv.json
```

Now, we can use this clever tool to draw a map around a certain location and it will generate a polygon in GeoJSON format. Notice in the graphic below that since Connecticut is roughly a rectangle, we can draw a rectangle of sufficient size to roughly define the location Connecticut.



Type **mongo** to open the MongoDB shell and then show one record as shown below and not the coordinates as a type **Point**:

```
db.address.findOne()
{
        "_id" : ObjectId("5c55b6b46c6e2fc66db81675"),
        "state" : "CT",
        "postcode" : "06037",
        "street" : "Parish Dr",
        "district" : "",
        "unit" : "",
        "location" : {
                "type" : "Point",
                "coordinates" :
        },
        "region" : "Hartford",
        "number" : "51",
        "city" : "Berlin"
}
```

Now we create an index. We use the parameter **2dsphere** meaning we want to use points on a sphere and not a Euclidean, i.e., flat, plane. That way when we work with distances MongoDB takes into consideration the curvature of the Earth:

```
db.address.createIndex( { "location": "2dsphere"} )
```

Now let's find addresses located within 4 meters of the coordinates shown below using the **$near** operator:

```
db.address.find ({
  location: {
```

```
     $near: {
       $geometry: {
          type: "Point" ,
          coordinates:
       },
       $maxDistance: 4,
       $minDistance: 0
     }
   }
})
```

Now let's find all address within the square that defines Connecticut that we drew above using $geoWithin. Note that you have to put an extra set of brackets [] around the coordinate pairs or MongoDB will complain that that is not an Array. Note by convention that the first and last line segment are the same. That's why it takes 5 lines to define and not 4, as you would think.

```
db.address.find ({location:
     {$geoWithin:
       {$geometry: {
       type: "Polygon",
       coordinates:
 ,
   ,
   ,
   ,
   ]]
}}}});
```

And since Mongo understands JavaScript we can define Connecticut like this:

```
var connecticut = db.address.find ({location:
     {$geoWithin:
       {$geometry: {
       type: "Polygon",
       coordinates:
 ,
   ,
   ,
   ,
   ]]
}}}});
```

So there are 918,356 addresses there:

```
connecticut.count()
918356
```