Open Charge Map

Data Cleaning workflows for global EV charging stations

UIUC MCS-DS: CS 513 Data Cleaning
Summer 2020

RYAN RICKERTS
Email: ryanjr3@illinois.edu
Repo: github.com/theRocket/ocm-data-mongonest

Open Charge Map

Data Cleaning workflows for global EV charging stations

Table of Contents

Project Overview

From the Github README for the OCM project: "Open Charge Map is the global public registry of electric vehicle charging locations. OCM was first established in 2011 and aims to crowdsource a high quality, well maintained open data set with the greatest breadth possible. Our data set is a mixture of manually entered information and imported open data sources (such as government-run registries and charging networks with an open data license). OCM provides data to drivers via hundreds of apps and sites, as well to researchers, EV market analysts, and government policy makers."

From their Project Objectives web page: "Our primary objective is to provide and maintain a quality Open Data set detailing the non-residential charging station infrastructure available to electric vehicle drivers globally."

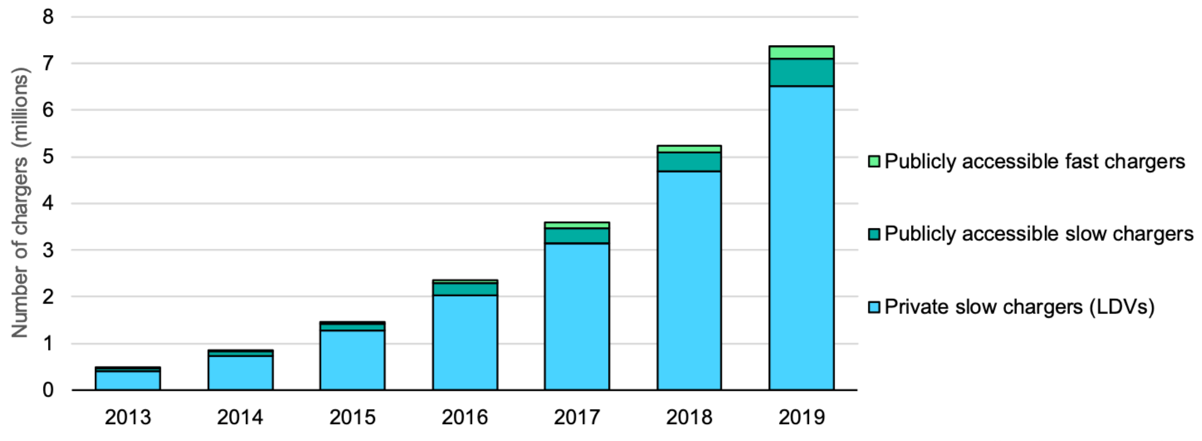
"To achieve this goal we provide:

- Access to an Open Data set of Charging Locations, Equipment details and user contributed supporting information. Note that the data licensing is mixed and specific to each applicable Data Provider.
- Tools and techniques (Web, Apps etc.) for accepting and managing data contributions and information maintenance.
- Web/API services to publish the data to all interested parties.
- Community support and discussions."

They also explicitly state that database support and editing of information is provided by volunteers, of which I am now one!

While the majority of EV charging worldwide occurs privately (such as the home or workplace), according to the International Energy Agency's Global EV Outlook 2020 report, there were approximately 862,000 publicly accessible chargers for Light Duty Vehicles (LDV) worldwide in 2019, a quarter of which were fast chargers. This figure has doubled since 2017 (see Figure 1.7 from the report below):

Figure 1.7 Global stock of electric LDV chargers, 2013-19



IEA 2020. All rights reserved.

While private enterprise has a huge role in implementation of a useful and accessible charging network to enable a shift away from our fossil fuel dependence, the role of reliable and open data sets for providing readily accessible information for all EV drivers regardless of their choice of EV car manufacturer, smartphone maker, or even navigation app for locating the best chargers for their route is obvious, so contributions from the sector of our economy that is not motivated primarily by profit (e.g. academia) is readily apparent.

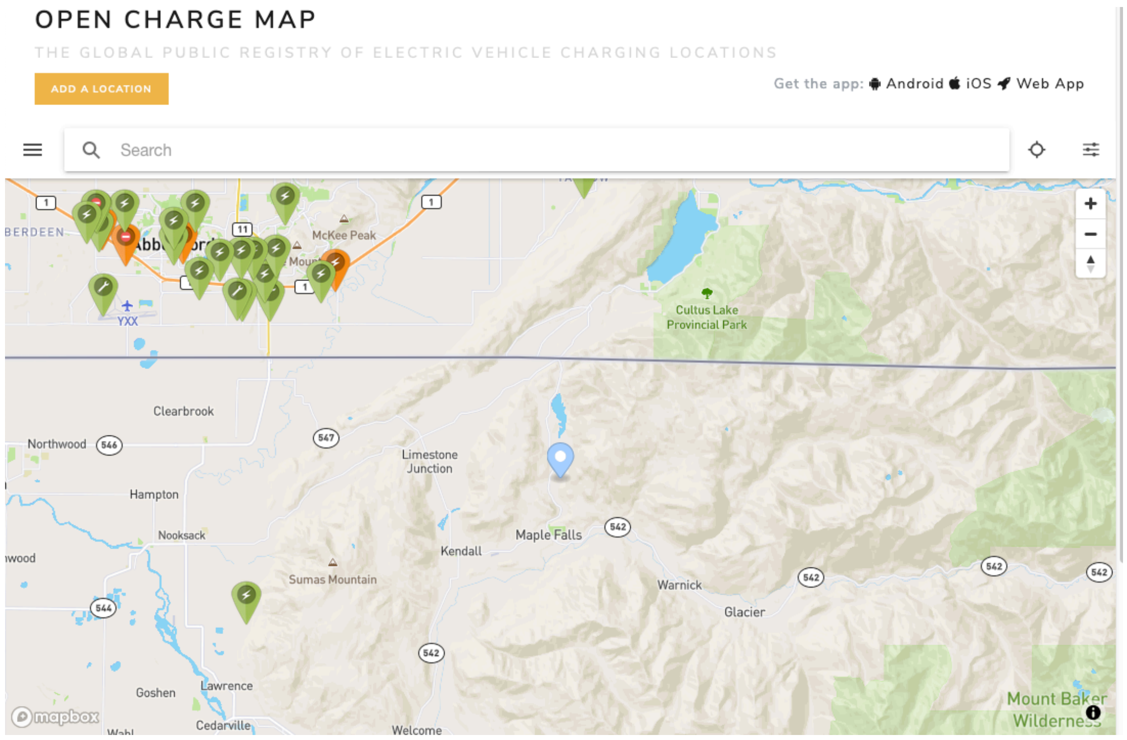
Data Cleaning Objectives

Combining the aspirations of the OCM project to "crowdsource a high quality, well maintained open data set with the greatest breadth possible" with the rapid pace of growth of the data elements themselves (doubling every two years) as well as the international characteristics of this data set (different charging standards, systems of measurement, ownership models, etc.) seems to create a prime candidate for contribution from the academic data science community.

My aim, in line with the stated aim of this project and course, is to verify the project has a standardized, well-documented data cleaning workflow that can be applied to incoming data, especially those imported from large data sources where automation would be of greatest advantage. If not, I hope to provide one for them.

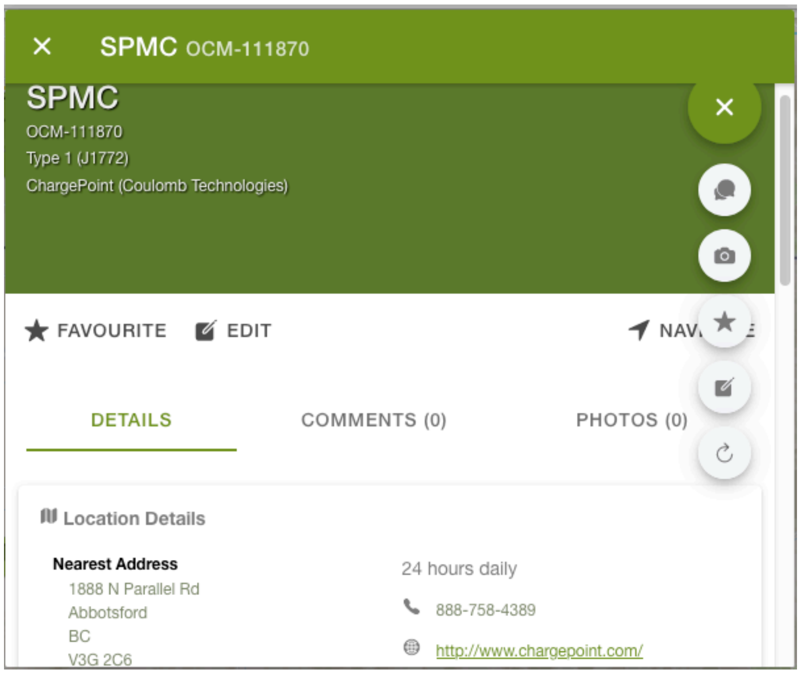
Use Cases

One obvious use case for this dataset is on the home page of their website, opencharegemap.org.



Open Charge Map homepage with my home location centered on map

After allowing location access in my browser (presumably based on my internet provider location or if my home address has been preconfigured in a browser setting), my home location has already been loaded up on a map powered by MapBox (another open standard) indicated by a blue balloon. Zooming out to a sufficiently wide area (since I live in a rural location with no nearby chargers), I soon see a set of charger locations with two different color codes, orange and green, and symbols such as a lightning bolt, a wrench, and a minus sign (there is no legend for these indicators). Clicking on a green icon, I get a popup with extensive information (if available):


SPMC OCM-111870

SPMC
OCM-111870
Type 1 (J1772)
ChargePoint (Coulomb Technologies)

Location Details

Nearest Address
1888 N Parallel Rd
Abbotsford
BC
V3G 2C6

24 hours daily
888-758-4389
http://www.chargepoint.com/

Scrolling is required to see all data points, but I can see values such as Location Details (address, lat/long, hours, phone, website), Equipment Details, Usage Restrictions, Network/Operator, and Additional Information. This should all correspond to values we see in the database schema in the next section.

Some locations show images for the connection types, usage costs, and in some cases filler data like “Planned For Future Date” or “Operational Status: Unknown.” This suggests there is a minimum dataset for which the charging point — aka Point of Interest (or POI) — can be added to the database, but the full set of information is not required. For example, they may not have a website associated, or they may not have software enabled that would indicate whether the site is currently in use (i.e. a car is plugged in).

Their website clearly states several other use cases for this data besides their own mapping solution:

Get Charging Station Data

Vehicle Manufacturers, Sat Nav vendors, App Developers, Website Owners: You can access and download our latest charging locations using our API.

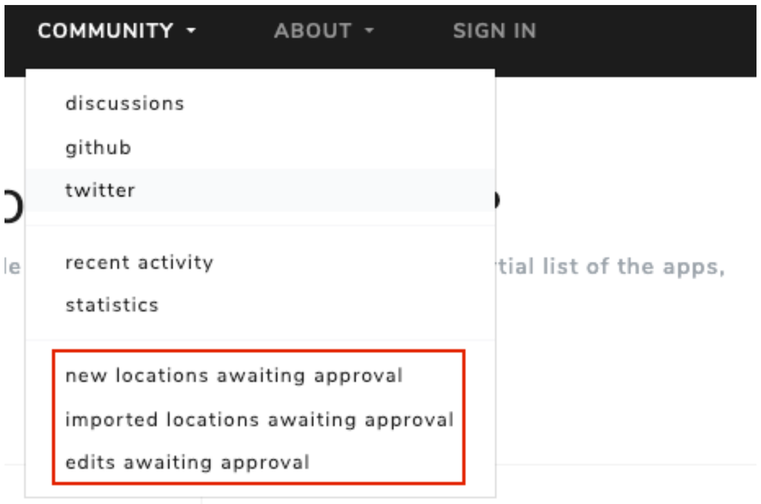
If you operate a website and would like to include a charging location map, you can embed a map on your own website.

Many of these are already in use, since they are advertising an Android and iOS app on their homepage, as well as a list of apps already using the data:

- Open Vehicle Monitoring System
- LEAF Control
- Place To Plug
- Charge/ by MyOxygen
- A Better Route Planner
- EV Trip Planner
- Tesla Apps
- Zero:Net
- and many others

Clearly there are many high quality sites and apps relying on this data, so it should be expected to uphold the highest quality standards possible for an open, community-curated project.

Speaking of community contributions, they do offer these links off their main menu:



This suggests it is possible for community members to validate manually entered data, automatically generated data (aka imports), and any edits. This is an amazing project!

After signing up for a login, I was able to view the edits awaiting approval. There were only two entry queued for 7/31/2020, and it appears in a table like this:

Field	Old Value	New Value
AddressInfo.Title	Cobscook Community Learning Center	Cobscook Institute
AddressInfo.AccessComments	Free guest wi-fi available, indoor and outdoor seating, outdoor park and hiking trails on the CCLC's beautiful 50 acre campus. The CCLC is a local non-profit with a mission to create responsive educational opportunities that strengthen personal, community, and global well-being.	Cobscook Institute's EV charger is open, although buildings on campus are currently closed to the public. We have free Wi-Fi available in the parking lot. No cost. Guest wi-fi available, outdoor seating, outdoor park and hiking trails on Cobscook Institute's beautiful 50 acre campus. Cobscook Institute is a local non-profit with a mission to create responsive educational opportunities that strengthen personal, community, and global well-being. More info at www.thecclc.org Formerly Cobscook Community Learning Center (CCLC)

We see the field names (Title and AccessComments in the AddressInfo table), the Old Value and the New Value to determine if the changes are valid. However, based on the status of "Awaiting Country Editor Approval," it appears only the designated editor for that country can approve the changes.

By clicking on the username of the data edit submitter, I get a profile of the user with stats such as date joined, a location, an optional bio and website, and community points (which I presume is based on contributions). The scope of all this data combined with the number of users changing it and the project's established workflows to keep it open and decentralized makes the project very interesting & challenging.

I can manipulate the query parameters on the web page to see past edits already processed within a date range. I choose the past month (July 2020) and see dozens of edits, who approved them, and when. This enables me to see all sorts of data processing results, but I still cannot determine their workflow. Is it automated, or manual? For instance, there are many revisions attributed to the same user with the same timestamp. Is he just very quick? Or was he using a tool?

```
Edit Processed by Simon Hewison 7/29/2020 2:06:00 PM
```

Here is another edit that provides a clue to foreign key relationships I will be discussing in the next section on data referential integrity:

Field	Old Value	New Value
OperatorInfo	(null)	OCM.API.Common.Model.OperatorInfo
OperatorID	(null)	199

Initial Assessment

Structure and Content

The project adds daily backups of their dataset to their Github repo at openchargemap/ocm-data. It includes a file called reference.json that is "an export of our current reference data (connection types, network operators, countries, etc.)" which is better known for our purposes as the schema file. However, since this is MongoDB dataset, often called NoSQL for not having a schema, the file must be included separately (and is likely maintained independently of the data flowing in, since MongoDB does not validate schemas by default).

The JSON is not easily viewed at the link provided above, but with a bit of formatting from a text editor, we can see it is a tree of data with 16 primary nodes of the following labels:

- `ChargerTypes`
- `ConnectionTypes`
- `CurrentTypes`
- `Countries`
- `DataProviders`
- `Operators`
- `StatusTypes`
- `SubmissionStatusTypes`
- `UsageTypes`
- `UserCommentTypes`
- `CheckinStatusTypes`
- `DataTypes`
- `MetadataGroups`
- `UserProfile`
- `ChargePoint`
- `UserComment`

For example in the ConnectionTypes, we see an element like this one:

```
{
    "ID": 3,
    "Title": "BS1363 3 Pin 13 Amp",
    "FormalName": "BS1363 / Type G",
    "IsDiscontinued": null,
    "IsObsolete": null
}
```

This record is given an ID field that does not match the MongoDB convention of generating a long string of random alphanumeric characters. I suspect when a charging station is added to the database, they use this ID as a foreign key reference to identify a valid connection type. This raises the question of what happens when a new connection type is introduced into the dataset on import before it has been included in the reference data. Are those elements rejected and reviewed by hand for a new standard by someone on the team? It seems likely, but this is not clear from their public documentation. What happens if a connection type that has been discontinued is discovered in the data? This seems like a prime candidate for a prospective provenance workflow, so we know what is intended to happen to these data elements so the entire team is aware of expectations for maintaining data cleanliness.

I also found a more descriptive JSON file at the ocm-docs / Model path to accompany the reference.json found in the backups repo. It is called poi.schema.json and has a top-level element (or key) called "description" with the value:

```
A POI (Point of Interest) is the top-level set of information regarding a Site with
electric vehicle charging. With the exception of the AddressInfo property, other
object properties may not be populated in a compact result set and instead only the
associated reference IDs will be set (e.g. UsageTypeID, DataProviderID etc).
```

This confirms my suspicion stated earlier, that references will be added by their corresponding ID. This means a manual entry must include a lookup (or memorization of IDs) but imports must perform the lookup in some sort of scripted mechanism, since these IDs are internal to the project (not a global standard ID). This should be explicitly stated in the provenance workflow.

The documentation entry for ConnectionType (the example above) is:

```
    "ConnectionType": {
        "$id": "#/properties/Connections/items/properties/ConnectionType",
        "type": "object",
        "title": "The Connectiontype Schema",
        "required": [
          "FormalName",
          "IsDiscontinued",
          "IsObsolete",
          "ID",
          "Title"
         ],
        "properties": {
           "FormalName": {
           "$id": "#/properties/Connections/items/properties/ConnectionType/properties/
FormalName",
           "type": "string",
            "title": "The Formalname Schema",
            "default": "",
            "examples": [
                "IEC 62196-2 Type 2"
             ],
            "pattern": "^(.*)$"
        },
        "IsDiscontinued": {
            "$id": "#/properties/Connections/items/properties/ConnectionType/
properties/IsDiscontinued",
            "type": "boolean",
            "title": "The Isdiscontinued Schema",
            "default": false,
            "examples": [
                 false
             ]
        },
        "IsObsolete": {
            "$id": "#/properties/Connections/items/properties/ConnectionType/
properties/IsObsolete",
            "type": "boolean",
            "title": "The Isobsolete Schema",
            "default": false,
            "examples": [
                false
             ]
         },
        "ID": {
            "$id": "#/properties/Connections/items/properties/ConnectionType/
properties/ID",
            "type": "integer",
            "title": "The Id Schema",
            "default": 0,
            "examples": [
                 25
             ]
          },
         "Title": {
            "$id": "#/properties/Connections/items/properties/ConnectionType/
properties/Title",
            "type": "string",
            "title": "The Title Schema",
            "default": "",
            "examples": [
                 "Type 2 (Socket Only)"
             ],
            "pattern": "^(.*)$"
          }
         }
        …
```

This is very helpful information! It gives data types (such as booleans), default values, and even pattern matching! This will be great for validating data (but it still does not describe what happens to the rejects nor much about referential integrity).

The parent of the ConnectionType element in the tree becomes apparent in this block, so we are getting closer to a complete picture of the schema:

```
    "Connections": {
          "$id": "#/properties/Connections",
          "type": "array",
          "title": "The Connections Schema",
          "items": {
            "$id": "#/properties/Connections/items",
            "type": "object",
            "title": "The Items Schema",
            "required": [
              "ID",
              "ConnectionTypeID",
              "ConnectionType",
              "Reference",
              "StatusTypeID",
              "StatusType",
              "LevelID",
              "Level",
              "Amps",
              "Voltage",
              "PowerKW",
              "CurrentTypeID",
              "CurrentType",
              "Quantity",
              "Comments"
            ],
```

Data integrity enforcements in these relationships are more apparent by viewing the SQL versions of their dataset, which is available in their ocm-docs repository under the Database / Scripts path. These have limited use for current workflows since it has not been updated in 4 years. Given the growth rate described in the project introduction, it seems likely there have been enough changes and additions to the standards in that time frame to render it obsolete. Hopefully the data structure is somewhat similar, so this will at least provide some clues from a traditional row-column view of data relationships.

Under ReferenceData, we see 17 .sql files corresponding to data for inserting into tables with the following names:

- `ChargerType`
- `CheckinStatusType`
- `ConnectionType`
- `Country`
- `CurrentType`
- `DataProvider`
- `DataProviderStatusType`
- `DataType`
- `EntityType`
- `MetadataField`
- `MetadataFieldOption`
- `MetadataGroup`
- `Operator`
- `StatusType`
- `SubmissionStatusType`
- `UsageType`
- `UserCommentType`

This seems to correspond the reference.json file we started the assessment with on page 3 above. Besides just being named as singular (while the MongoDB nodes are plural), there are differences in some of the tables, such as DataProviderStatusType and MetadataFieldOption (both missing above). I won't bother trying to determine what happened to those elements in 4 years time. Instead, I will examine the structure of data we will be working with in our JSON dataset in MongoDB without a strict schema like this one.

On Row 6 of the dbo.ConnectionType.Table.sql file, we have:

```
   INSERT [dbo].[ConnectionType] ([ID], [Title], [FormalName], [IsDiscontinued],
[IsObsolete]) VALUES (3, N'BS1363 3 Pin 13 Amp', N'BS1363 / Type G', NULL, NULL)
```

This matches the sample JSON data block we provided previously, value for value. That is encouraging. It has 35 rows to be inserted, where-as the reference.json has 41 records, so that confirms my suspicion that connection type standards are being added (and likely discontinued) continuously. This highlights an opportunity to do some reporting on which EV stations have discontinued connection types. Should they be contacted by a vendor to discuss upgrade opportunities? Have they already converted but did not report the new plug option? Do we have their contact info in the related table? This is putting our dataset to good, practical use!

The Schema folder (next to the ReferenceData folder in the same SQL reference section) has many more .sql files which would create the actual tables we need, and likely the foreign key references. For example, in the parent element described before, dbo.ChargePoint.Table.sql:

```
    CREATE TABLE [dbo].[ChargePoint](
              [ID] [int] IDENTITY(1,1) NOT NULL,
              [UUID] [nvarchar](100) NOT NULL,
              [ParentChargePointID] [int] NULL,
              [DataProviderID] [int] NOT NULL,
              [DataProvidersReference] [nvarchar](100) NULL,
              [OperatorID] [int] NULL,
              [OperatorsReference] [nvarchar](100) NULL,
              [UsageTypeID] [int] NULL,
              [AddressInfoID] [int] NULL,
              [NumberOfPoints] [int] NULL,
              [GeneralComments] [nvarchar](max) NULL,
              [DatePlanned] [smalldatetime] NULL,
              [DateLastConfirmed] [smalldatetime] NULL,
              [StatusTypeID] [int] NULL,
              [DateLastStatusUpdate] [smalldatetime] NULL,
              [DataQualityLevel] [int] NULL,
              [DateCreated] [smalldatetime] NULL,
              [SubmissionStatusTypeID] [int] NULL,
              [UsageCost] [nvarchar](200) NULL,
              …
   CREATE TABLE [dbo].[ConnectionInfo](
              [ID] [int] IDENTITY(1,1) NOT NULL,
              [ChargePointID] [int] NOT NULL,
              [ConnectionTypeID] [int] NOT NULL,
              [Reference] [nvarchar](100) NULL,
              [StatusTypeID] [int] NULL,
              [Amps] [int] NULL,
              [Voltage] [int] NULL,
              [PowerKW] [float] NULL,
              [LevelTypeID] [int] NULL,
              [Quantity] [int] NULL,
              [Comments] [nvarchar](max) NULL,
             [CurrentTypeID] [smallint] NULL
             …
   ALTER TABLE [dbo].[ConnectionInfo]  WITH CHECK ADD  CONSTRAINT
[FK_ConnectionInfo_ConnectorType] FOREIGN KEY([ConnectionTypeID])
   REFERENCES [dbo].[ConnectionType] ([ID])
```

So there we have it. The ConnectionInfo table is likely the bridge (or a join table) between a ChargePoint record and a ConnectionType record, referenced by internal ID. At least, that was the design 4 years ago in SQL!

To understand how they were manipulating data using SQL, let's look at a few of their stored procedure, also available in this directory, such as dbo.procInsertOrUpdateChargePoint.StoredProcedure.sql. An important block of logic is as follows:

```
--check if charge point exists already based on Operator reference or DataProvider ref
 SELECT @ChargePointID =ID FROM ChargePoint
 WHERE OperatorID=@OperatorID AND OperatorsReference=@OperatorsReference

 IF @ChargePointID IS NULL BEGIN
       SELECT @ChargePointID =ID FROM ChargePoint
       WHERE DataProviderID=@DataProviderID AND
DataProvidersReference=@DataProvidersReference
 END

 --if chargepoint doesn't exist, add new charge point details
 IF @ChargePointID IS NULL
 BEGIN
       --add new charge point
       INSERT INTO ChargePoint(DataProviderID, DataProvidersReference, OperatorID,
OperatorsReference, UsageTypeID, UsageCost, AddressInfoID, NumberOfPoints,
GeneralComments, DatePlanned, DateLastConfirmed, StatusTypeID, DateLastStatusUpdate,
DateCreated, SubmissionStatusTypeID)
       VALUES (@DataProviderID, @DataProvidersReference, @OperatorID,
@OperatorsReference, @UsageTypeID, @UsageCost, NULL, @NumberOfPoints,
@ChargePointComments, @DatePlanned, @DateLastConfirmed, @StatusTypeID,
@DateLastStatusUpdate, GETDATE(), 100 --imported and published
             )
  SET @ChargePointID=SCOPE_IDENTITY()
```

This answers my question about some of their data entry workflows to maintain relational integrity. As described nicely in the comments, the logic first checks if the Chargepoint already exists (first by unique combination of OperatorID and OperatorsReference, then by DataProviderID). If the ChargePointID is still a NULL value, it creates a new one and stores the new primary key generated by the database engine in a local variable for further use.

There are other interesting transformations here, such as one to "convert text string to title case proper case" (a step we have performed in OpenRefine in our assignments), but since this appears to be a deprecated workflow (or at least not updated on Github regularly), I will leave the analysis at this depth and focus on what can be done with MongoDB, since the scripting language for manipulating JSON data is far different (Javascript vs SQL) and the engine is also drastically different. Luckily, I have a bit of experience with this database type from my previous employment.

Data Cleaning Workflows

Once extracted from the Gzip format, the poi.json file from the data backup is nearly a 500MB file. This obviously cannot be kept within the repo (due to Github limits of 100MB), so it is included in the .gitignore file. The poi.json file will open in a text editor and shows 145,179 lines of code, each a JSON block which I presume is one entry in a MongoDB collection. It is difficult to scroll through the data let alone grasp its structure viewing it this way.

Step 2. Data cleaning with OpenRefine

The project specification asks that we process this first with OpenRefine. There will be several differences from our Week 2 assignment learning this software tool. First off, editing a text document of this size challenges my OS memory using just a text editor (especially with JSON formatting applied), so I was suspicious that a free tool running in the Java Runtime Environment could handle it. Therefore, to sample the data with OpenRefine, I first truncated the data at 3,000 rows. This reduced the file size to 10MB and allows me to try the tool without bogging it down unnecessarily.

The second challenge loading this into OpenRefine is the difference between row-column based data structures (table-based) we have been using in lectures & assignments versus JSON or XML data structures (tree-based). The first would load nicely in a tool like Excel or Sqlite. The latter is better viewed in a navigational style where parent-child relationships (or node-leaf) can be collapsed and expanded. This is easily achieved using their API in a browser or API tool that formats JSON, like Postman.

For example, the following API call:

https://api.openchargemap.io/v3/poi

submits the API version number (3) to the POI endpoint and returns 100 collapsed data elements.

The documentation for this API is provided here: https://openchargemap.org/site/develop/api with suggestions like:

> The default output contains a lot of information. Here is the same call as above, but with the most compact output (formatting removed, reference data as IDs instead of full objects, null fields skipped): `https://`

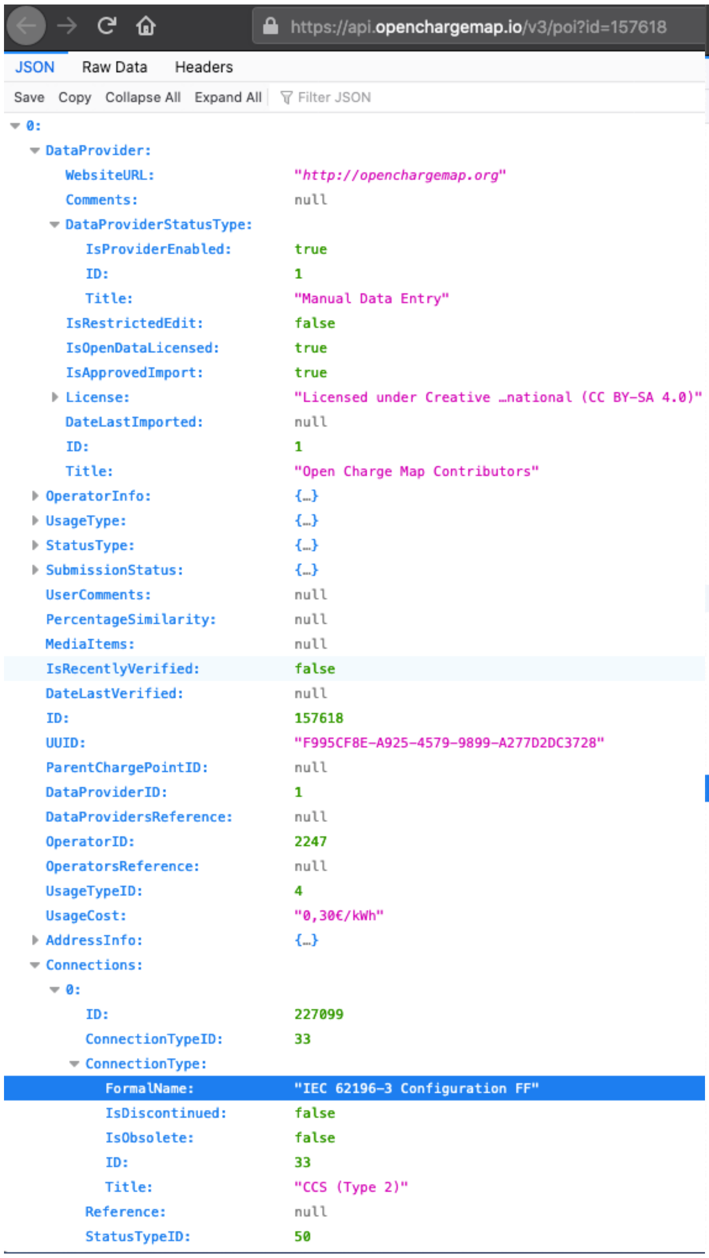
[api.opencharge map.io/v3/poi/?
output=json&countrycode=US&maxresults=100&compact=true&verbose=false]

There are 28 possible parameters you can specify in the URL, all documented on that webpage. There is also a link to the endpoint for core reference data, which appears to the same reference.json file we have already discussed at length in the previous section.

Specifying a particular POI by its ID, such as:

https://api.openchargemap.io/v3/poi?id=157618

results in the following displays in Firefox browser (compact and less verbose version on the right, which suppresses null values and shows only the most interesting data):

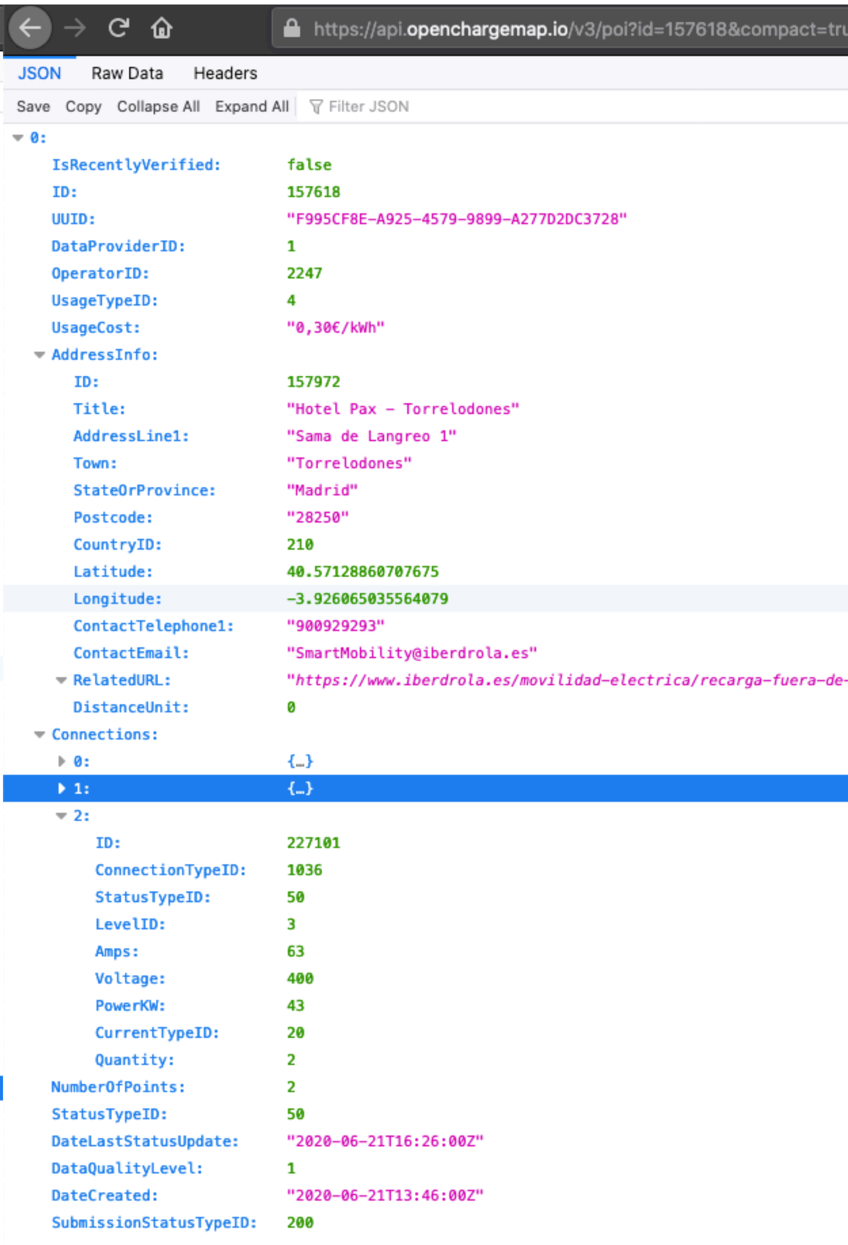


https://api.openchargemap.io/v3/poi?id=157618

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
▼ 0:
  ▼ DataProvider:
      WebsiteURL:                 "http://openchargemap.org"
      Comments:                   null
    ▼ DataProviderStatusType:
        IsProviderEnabled:        true
        ID:                       1
        Title:                    "Manual Data Entry"
      IsRestrictedEdit:           false
      IsOpenDataLicensed:         true
      IsApprovedImport:           true
    ▶ License:                    "Licensed under Creative …national (CC BY-SA 4.0)"
      DateLastImported:           null
      ID:                         1
      Title:                      "Open Charge Map Contributors"
  ▶ OperatorInfo:                 {…}
  ▶ UsageType:                    {…}
  ▶ StatusType:                   {…}
  ▶ SubmissionStatus:             {…}
    UserComments:                 null
    PercentageSimilarity:         null
    MediaItems:                   null
    IsRecentlyVerified:           false
    DateLastVerified:             null
    ID:                           157618
    UUID:                         "F995CF8E-A925-4579-9899-A277D2DC3728"
    ParentChargePointID:          null
    DataProviderID:               1
    DataProvidersReference:       null
    OperatorID:                   2247
    OperatorsReference:           null
    UsageTypeID:                  4
    UsageCost:                    "0,30€/kWh"
  ▶ AddressInfo:                  {…}
  ▼ Connections:
    ▼ 0:
        ID:                       227099
        ConnectionTypeID:         33
      ▼ ConnectionType:
          FormalName:             "IEC 62196-3 Configuration FF"
          IsDiscontinued:         false
          IsObsolete:             false
          ID:                     33
          Title:                  "CCS (Type 2)"
        Reference:                null
        StatusTypeID:             50
```



https://api.openchargemap.io/v3/poi?id=157618&compact=tru

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

```
▼ 0:
    IsRecentlyVerified:           false
    ID:                           157618
    UUID:                         "F995CF8E-A925-4579-9899-A277D2DC3728"
    DataProviderID:               1
    OperatorID:                   2247
    UsageTypeID:                  4
    UsageCost:                    "0,30€/kWh"
  ▼ AddressInfo:
      ID:                         157972
      Title:                      "Hotel Pax - Torrelodones"
      AddressLine1:               "Sama de Langreo 1"
      Town:                       "Torrelodones"
      StateOrProvince:            "Madrid"
      Postcode:                   "28250"
      CountryID:                  210
      Latitude:                   40.57128860707675
      Longitude:                  -3.926065035564079
      ContactTelephone1:          "900929293"
      ContactEmail:               "SmartMobility@iberdrola.es"
    ▼ RelatedURL:                 "https://www.iberdrola.es/movilidad-electrica/recarga-fuera-de-
      DistanceUnit:               0
  ▼ Connections:
    ▶ 0:                          {…}
    ▶ 1:                          {…}
    ▼ 2:
        ID:                       227101
        ConnectionTypeID:         1036
        StatusTypeID:             50
        LevelID:                  3
        Amps:                     63
        Voltage:                  400
        PowerKW:                  43
        CurrentTypeID:            20
        Quantity:                 2
    NumberOfPoints:               2
    StatusTypeID:                 50
    DateLastStatusUpdate:         "2020-06-21T16:26:00Z"
    DataQualityLevel:             1
    DateCreated:                  "2020-06-21T13:46:00Z"
    SubmissionStatusTypeID:       200
```

While OpenRefine accepts JSON as an input type for project imports, it is not likely this data can be transformed well into an OpenRefine table. For example, just in the table preview, the UsageType node is split into several columns for each of its fields with column titles like:

- UsageType - ID
- UsageType - Title
- UsageType - IsPayAtLocation
- UsageType - IsAccessKeyRequired
- UsageType - IsMembershipRequired

If I select none of the data, I have a confusing user experience where "Please specify a record path first" is reported as an unresolvable error (filed on Github as a bug report).

OpenRefine Workflow Steps for **AddressInfo**:

1) Selecting a child or leaf node like AddressInfo allows me to continue and create a flattened table.

2) After proving I can do this for 3,000 rows, I try again with the full 500MB file. OpenRefine actually performs quite well with this! I am able to create a project with 145,179 rows. I named it OCMData-POIs-AddressInfo.

3) Once the data is imported, I begin by renaming and reordering the columns. I strip off the leading "AddressInfo - " string from each column, since it just clutters the view. I also put related elements near each other, such as Latitude & Longitude, since importing JSON seems to order them randomly.

4) I run a transform on AddressLine2 to strip any leading or trailing whitespace, then apply a filter on it using a regular expression of '\S+' to fetch only those entries with non-whitespace in them. This results in 8,741 rows, so I keep the column.

5) I apply a facet to GeneralComments and it reports all values are blank, so I delete this column.

6) I try the same for ContactTelephone2 and get 649 choices total, so this one I keep.

7) I apply the text filter for non-whitespace on ContactEmail and examine the data quality on the resulting 6,000 rows. I apply a general regular expression for valid emails and get only 5,772 rows, so this makes me curious how to find the invalid ones. I hit 'invert' and get 139,407 rows, so this is not the 228 records I'm looking for. I try a more complex regular expression "that 99.9% works" and get 5,791 records, so now I'm suspicious of any claims on these crowdsourced matching patterns. I remove the filter and make sure any trailing & leading whitespace is removed. This affects 14 rows. I ultimately settle on this regex pattern, which matches on 5,787 rows:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$
```

It is still unclear how I would find the rows that didn't match this expression but are not blank.

8) I do the same operation for RelatedURL, but with a generally accepted URL regex from StackOverflow:

```
    https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\+~#=]{1,256}\.[a-zA-Z0-9()]{1,6}\b([-a-zA-Z0-9()@:%_\
    +.~#?&//=]*)
```

This results in 74,757 rows. Non-blanks is 74,853 rows, so we're pretty close on that one (same problem as previous step on how to identify the non-compliant ones). I try a facet here to see how many URLs are identical. I get 13,121 unique counts, so it appears there are many duplicates.

I try a text filter on "http://www.mychargepoint.net" and get 1,338 rows. Curiously, that link redirects to https://pnrstatus.vip/ which is for checking "Indian Railways train live PNR status online."

9) I run a filter on Distance and they are all blank. I apply a facet to the DistanceUnit and they are all zero. So I delete these columns. Note, I did find a SQL function in their repo that calculates these values using the following logic relying on a GEOGRAPHY library:

```
IF @Latitude1 IS NULL OR @Longitude1 IS NULL OR @Latitude2 IS NULL OR @Longitude2 IS
NULL RETURN NULL;

DECLARE @sourceLocation GEOGRAPHY = GEOGRAPHY::Point(@Latitude1,@Longitude1, 4326) ;

SELECT  @distanceKM = @sourceLocation.STDistance(GEOGRAPHY::Point(@Latitude2,
@Longitude2, 4326))/1000;
```

10) We have some additional columns related to Country. There is a duplicate one for Country ID, a Country Title, ISO Code, and Continent. This seems non-normalized if there is in fact a Country table with this information and a primary key that corresponds to the foreign key here. We'll examine that further in the SQL workflow (Step 4).

I save the steps above that are traceable in a file called **Open_Refine_History_addrinfo.json** attached to this report.

OpenRefine Workflow Steps for **ConnectionType**:

I repeat this process for a different data node, one that is simpler and more standardized to match the reference data. I perform the same steps on column names, stripping the "ConnectionType - " part on each. Now we can run some more interesting facets and see informative record counts.

I am noticing that the foreign key on ConnectionType ID is present as well as all the related data from what would be a related table. This makes the data non-normalized in the SQL world, but perhaps in their MongoDB, they pump the data values into each instance since this all in one giant collection.

Therefore, when I apply a text facet to the Title column, I get only 38 choices. In the Structure and Content section of the Initial Assessment (p. 12), we found 41 possible values in the reference data for these ConnectionTypes. So there are 3 that are not referenced, and they could be ones marked IsObsolete or IsDiscontinued, but I see records with those values in this dataset by applying a text facet:

IsObsolete: 13 marked 'true'; 133,866 marked 'false'; and 80,918 left blank.
IsDiscontinued: 203 marked 'true'; 133,676 marked 'false'; 80,918 left blank.

I get the following results on the Title text facet:

```
Avcon Connector          55
Blue Commando (2P+E)     327
BS1363 3 Pin 13 Amp      2,048
CCS (Type 1)             4,655
CCS (Type 2)             9,539
CEE 3 Pin                965
CEE 5 Pin                1,257
CEE 7/4-Schuko-Type F    11,881
CEE 7/                   5698
CEE+ 7 Pin               10
CHAdeMO                  17,760
Europlug 2-Pin(CEE 7/16) 228
IEC 60309 3-pin          18
IEC 60309 5-pin          47
LP Inductive             2
NEMA 14-30               6
NEMA 14-50               149
NEMA 5-15R               64
NEMA 5-20R               2,491
NEMA 6-15                11
NEMA 6-20                13
NEMA TT-30R              1
SCAME Type 3A(Low Power) 2,158
SCAME Type 3C (Schneider-Legrand)       1,111
SP Inductive             11
T13 - SEC1011 ( Swiss domestic 3-pin ) - Type J        86
Tesla (Model S/X)        18,029
Tesla (Roadster)         135
Tesla Battery Swap       2
Tesla Supercharger       4,209
Three Phase 5-Pin (AS/NZ 3123)          20
Type 1 (J1772)           42,392
Type 2 (Socket Only)     67,876
Type 2 (Tethered Connector)             8,026
Type I (AS 3112)         120
Unknown                  18,391
Wireless Charging        5
XLR Plug (4 pin)         1
```

Now it is apparent which connectors are the most common and which are quite rare (it seems most connector types fit into one or the other camps). We also have the troubling statistic of 18,391 "unknown" connector types. This seems like a significant data hole that should be addressed.

We could also run an analysis to determine if the IDs provided match the reference dataset, or if the non-normalized data has got us into trouble with data quality. For example, if a Connection Type is marked obsolete in the reference data, are all the records here now matching?

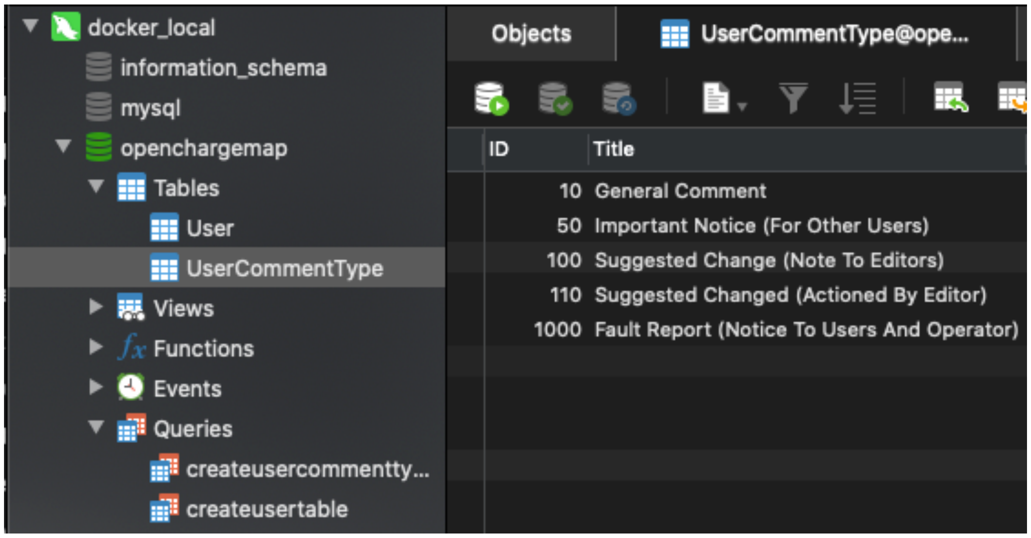
See Open_Refine_History_connecttype.json for the alteration history on this table (not much) and the .csv and openrefine.tar.gz files for an export of these results.

Step 4. Creating a SQL schema

As discussed in the Initial Assessment on page 11, the project already has a SQL representation of this data hosted on Github (although not updated in 4 years). In this section, I will attempt to load the data into MySQL 5.7 in a Docker container so queries can be performed on it. It is not clear from the Github repo which order they should be loaded in so the foreign key references are valid, so here we go!

The first table I attempted to create is User. It became clear very quickly that these scripts are based on Microsoft SQL Server (data types like smalldatetime are only available for that product or Azure). I simplified the commands so that I could just get a basic table created.

Next table I try is UserCommentType because it has no foreign key references either and is quite simple. I am also able to add the reference data to my query to pre-load it with the default data. You can see it here in my MySQL editing tool, Navicat. You can also see my saved queries on the lower left sidebar:



ID	Title
10	General Comment
50	Important Notice (For Other Users)
100	Suggested Change (Note To Editors)
110	Suggested Changed (Actioned By Editor)
1000	Fault Report (Notice To Users And Operator)

It occurs to me now that all the reference data (or I would call it seed data) are prime candidates for first tables to create, so I continue with all the table specs for the reference data: Countries, CurrentType, ConnectionType, StatusType, etc.

I get through several tables and realize this is all possible for recreating the SQL-based schema, but will be somewhat manual to convert their scripts from MS-SQL to MySQL compliant syntax. It does not seem likely that a scripted routine could perform this work, as the differences between the two databases are too drastic, from data types to default values to field properties. It is clear that since they seeded IDs for the reference data, that the foreign key references in the related tables could be enforced, but I would have to build out the whole database manually to prove this.

I did model this data at a high level (tables and relations, but not all fields) and it is easy to surmise the relationships from convention of "ForeignTablenameID" for the foreign key field name. I can also see the relations added in the SQL table creation routines, such as (abbreviated):

```
ALTER TABLE UserComment ADD FOREIGN KEY (UserID) REFERENCES User(ID);
```

One reference data table of interest at the schema level is the one called DataType. The list is as follows. It would be interesting to see how this is applied in logic (I'm guessing as how to format them for display).

```
Single Line Text
Multi Line Text
Number
Date
Boolean
Option List
```

I also learn through this process that there are about 250 pre-populated Operators and their website URL, such as:

```
'ChargeNow', 'https://www.chargenow.com/'
```

I did not include data provenance related tables (which appear to be for logging or archiving), nor views (aggregations of tables), nor stored procedures (likely for reporting, auditing, & cleaning) in this schema diagram, merely for simplicity. I really just wanted to understand how the model is "shaped" as the core framework of data. The views give examples of how to query the data, since they are basically SELECT statements with a bunch of JOIN statements. One example for displaying the full location info for a charge point is called dbo.ViewAllLocations.View.sql but is too long to display here.

This schema diagram available in the resources section of this report as a file named openchargermap_datarelationships.png). The diagram is also temporarily hosted in the Gliffy cloud here: https://go.gliffy.com/go/publish/13337984

Some intervention must be made by the database architect to decide accurately how to pivot from MS-SQL syntax into the MySQL engine without compromising data integrity. This seems unnecessary given that the project has already pivoted from SQL to NoSQL, so we'll move on to the MongoDB based workflows which are the most current. I feel understanding the historical model will still be helpful once we move into a schema-less database, so the diagram creation was a worthwhile exercise.

Step 5. Creating a workflow model for MongoDB

This workflow will correspond to the Gliffy diagram hosted on their cloud site here: https://go.gliffy.com/go/publish/13337014 (also in resources as ocm-dataworkflow.png).

The repository with this code is hosted on Github at: github.com/theRocket/ocm-data-mongonest.

Step 1) Since this backup file of their MongoDB database — poi.json.gz — is a daily or twice daily snapshot, it would not be generally useful to start with older data, so my first step is to obtain the latest copy of this file and extract it. The copy I am starting with is from the following commit:

```
commit c499aff6a821d404fbe6acae22229d59f1260d6f (HEAD -> master, origin/master,
origin/HEAD)
Author: OCM <data@openchargemap.org>
Date:   Thu Jul 30 12:15:56 2020 +0000
```

This file needs to be unzipped (gunzip) and dropped into the project directory docker-entrypoint-initdb.d next to the seed.js file that initializes the database, collection, and user in the next step.

The file reference.json would be useful if initializing a brand new database, but the values should be included in the full backup, so it is not needed in this workflow except for actual reference.

Step 2) The next step is to fire up a MongoDB database in a Docker container and related services. The docker-compose.yml file contains all the necessary code to make this work. Key among that code is the part that copies our seed.js and poi.json files into the container. Those lines are:

```
volumes:
 - ./docker-entrypoint-initdb.d/seed.js:/docker-entrypoint-initdb.d/seed.js:ro
 - ./docker-entrypoint-initdb.d/poi.json:/docker-entrypoint-initdb.d/poi.json:ro
```

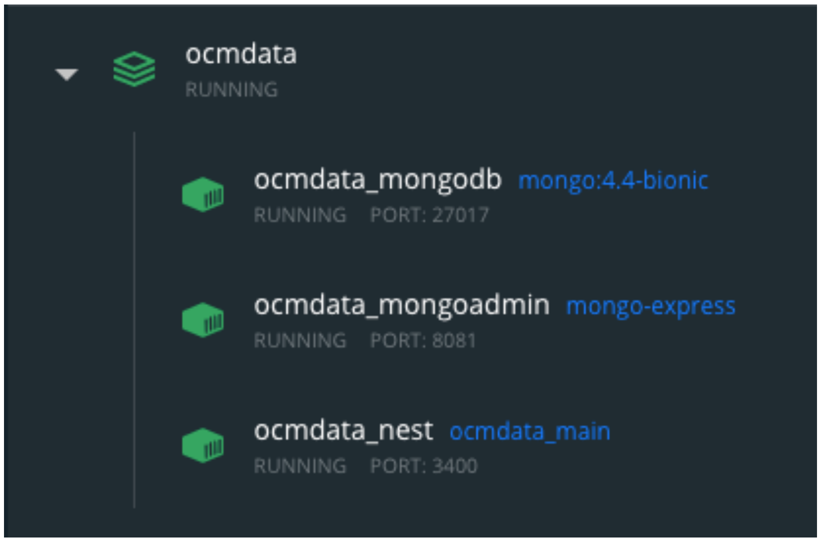
Step 3) Run the following Docker-compose command (presumes you have Docker installed and filled out the .env or config/environments/*.env files with proper DB credentials) will build out 3 containers, all linked to each other on one service called **ocm-data-mongonest**, and running on their own local network called **ocm-data-mongonest_netty:**

```
ocm-data-mongonest git:(master) ✗ docker-compose --project-name ocm-data-
mongonest up —build --renew-anon-volumes --detach

Creating network "ocmdata_netty" with the default driver
Building main
Step 1/8 : FROM node:12.13-buster as development
...

Successfully built c914ed1bcaf6
Successfully tagged ocmdata_main:latest
Creating ocmdata_mongodb ... done
Creating ocmdata_mongoadmin ... done
Creating ocmdata_nest       ... done
```

The primary container, **ocmdata_mongodb** holds the database. The next one is a web-based admin interface called Mongo Express for interacting with the data in a GUI, which we will cover in Step 6. The third is a NestJS framework (Typescript version of NodeJS) application that will allow us to program against the data using a Javascript-based MongoDB database driver called MongooseJS. This is more apparent looking at the Dashboard feature in Docker Desktop (on a Mac):


ocmdata
RUNNING
ocmdata_mongodb mongo:4.4-bionic
RUNNING PORT: 27017
ocmdata_mongoadmin mongo-express
RUNNING PORT: 8081
ocmdata_nest ocmdata_main
RUNNING PORT: 3400

Step 4) Now we will enter the container to run a mongo import routine that will initialize the database with the 500MB of JSON above. This presumes the **seed.js** file was successful in running according to their documentation on Dockerhub (see Initializing a fresh instance) which adds our database, user authentication for the app, and the collection. This can be verified with the docker-compose logs for the container. For example:

```
{"remote":"127.0.0.1:38664","client":"conn3","doc":{"application":
{"name":"MongoDB Shell"},"driver":{"name":"MongoDB Internal
Client","version":"4.4.0"},"os":
{"type":"Linux","name":"Ubuntu","architecture":"x86_64","version":"18.04"}}}}
Successfully added user: {
  "user" : "nest_user",
  "roles" : [
   {
    "role" : "readWrite",
    "db" : "ocmdata"
   }
  ]
 }
```

We can also execute a bash session in the container and connect with the **mongo** client:

```
ocm-data-mongonest git:(master) ✗ docker exec -it ocmdata_mongodb bash

root@45f6e97826fd:/docker-entrypoint-initdb.d# mongo
MongoDB shell version v4.4.0
connecting to: mongodb://127.0.0.1:27017/?
compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("4aa8e999-e921-4ada-b392-a5995d15d1cc") }
MongoDB server version: 4.4.0
Welcome to the MongoDB shell.

> use ocmdata
```

```
switched to db ocmdata
> db.auth('nest_user','generic')
1
> db.pois.find({})
> exit
bye
```

The query 'db.poi.find({})' returns no results, proving that the database is empty. We exit the mongo client before executing the next command within the container.

Step 5) From within the /docker-entrypoint-initdb.d directory (at the root of containers file system after success with Step 2) we can execute the **mongoimport** command with the above credentials, like so:

```
root@45f6e97826fd:/docker-entrypoint-initdb.d# mongoimport --username nest_user --
password generic --authenticationDatabase ocmdata --db ocmdata --collection pois --
type json --file ./poi.json
```

```
2020-08-05T00:21:18.880+0000	connected to: mongodb://localhost/
2020-08-05T00:21:21.881+0000	[###.....................] ocmdata.poi	63.5MB/476MB
(13.3%)
...
2020-08-05T00:21:39.847+0000	[###################.....] ocmdata.poi	395MB/476MB
(83.1%)
2020-08-05T00:21:42.848+0000	[######################..] ocmdata.poi	449MB/476MB
(94.3%)
2020-08-05T00:21:44.284+0000	[########################] ocmdata.poi	476MB/476MB
(100.0%)
2020-08-05T00:21:44.285+0000	145179 document(s) imported successfully. 0
document(s) failed to import.
```

Looking at the container logs, we see the entries with matching timestamps at the beginning and end of the **mongoimport** operation:

```
{"t":{"$date":"2020-08-05T00:21:18.857+00:00"},"s":"I", "c":"NETWORK",
"id":51800, "ctx":"conn18","msg":"client metadata","attr":
{"remote":"127.0.0.1:38704","client":"conn18","doc":{"driver":{"name":"mongo-go-
driver","version":"v1.4.0-rc0"},"os":
{"type":"linux","architecture":"amd64"},"platform":"go1.12.17","application":
{"name":"mongoimport"}}}}

{"t":{"$date":"2020-08-05T00:21:18.876+00:00"},"s":"I", "c":"ACCESS", "id":20250,
"ctx":"conn18","msg":"Successful authentication","attr":{"mechanism":"SCRAM-
SHA-256","principalName":"nest_user","authenticationDatabase":"ocmdata","client":
"127.0.0.1:38704"}}

{"t":{"$date":"2020-08-05T00:21:44.286+00:00"},"s":"I", "c":"-", "id":20883,
"ctx":"conn17","msg":"Interrupted operation as its client disconnected","attr":
{"opId":9691}}
```

In other words, success! Now we can execute the mongo client session as in step 4 and begin interacting with the data (after repeating the authentication steps):

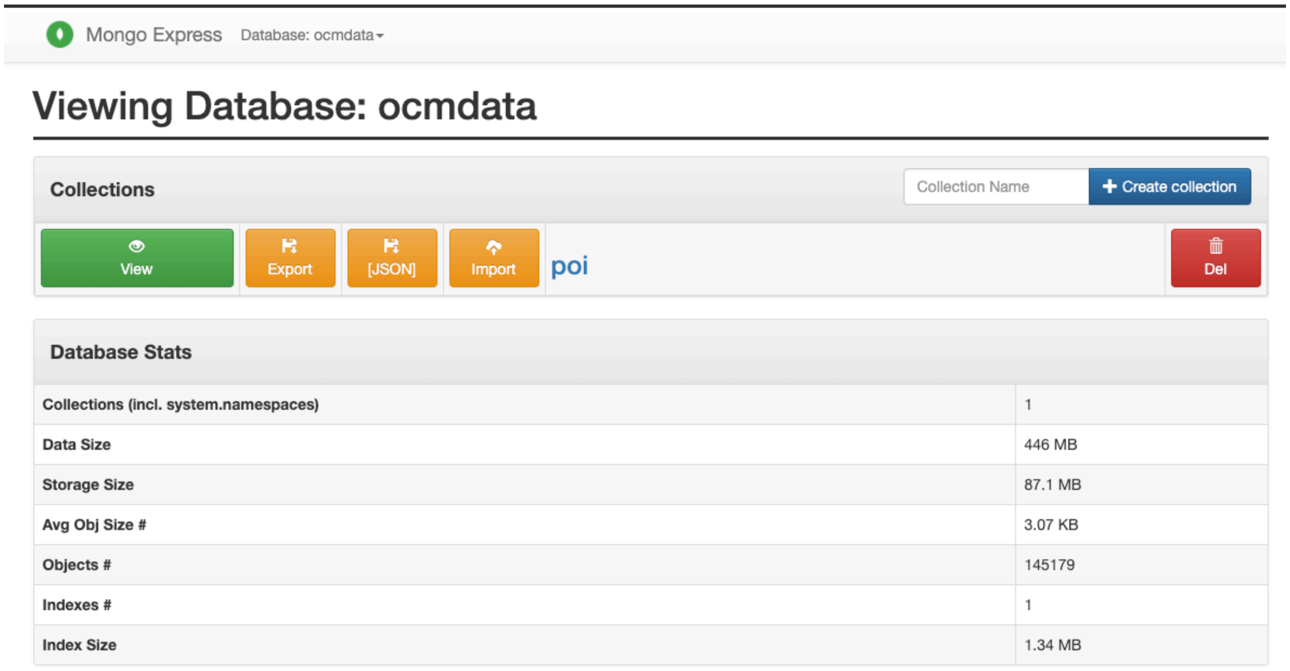
```
> db.pois.find({}).count()

145179
```

We find the document count matches the import logs. This is highly encouraging!

Step 6) I fire up the Mongo Express web interface using the address shown in the Docker container info (localhost:8081) and supply the root user credentials (stored in the .env file in the project but included in .gitignore for security reasons).

I am now able to view the stats for the ocmdata collection. Notice it is slightly smaller than the raw JSON file itself, by about 53 MB. The storage size is drastically smaller at 87MB, showing the optimizations in this open source database are incredible! We also see the average object size (3.07 KB) and the number of objects match. We have only one collection (POI) and one index (likely _ID, the mongo default):



Viewing Database: ocmdata

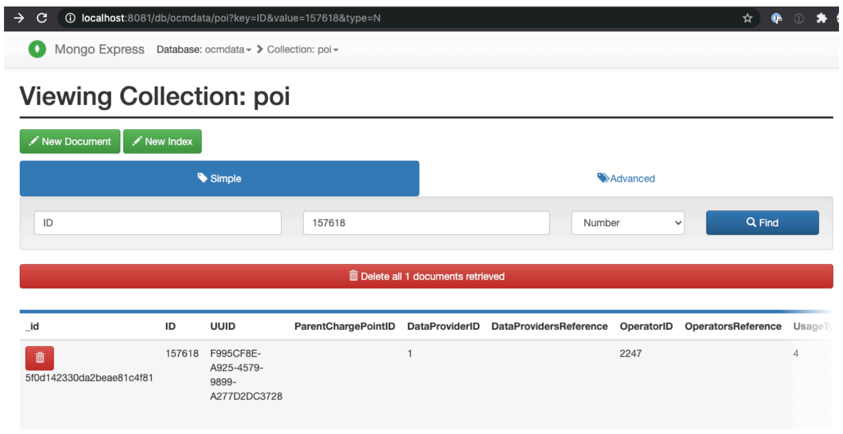
Collections

poi

Database Stats

Collections (incl. system.namespaces)	1
Data Size	446 MB
Storage Size	87.1 MB
Avg Obj Size #	3.07 KB
Objects #	145179
Indexes #	1
Index Size	1.34 MB

I am able to run a query in this interface for the explicit numeric ID for a Chargepoint provided by the database designer, the same one shown in the API call in the Firefox browser on p. 15, and obtain the same result:



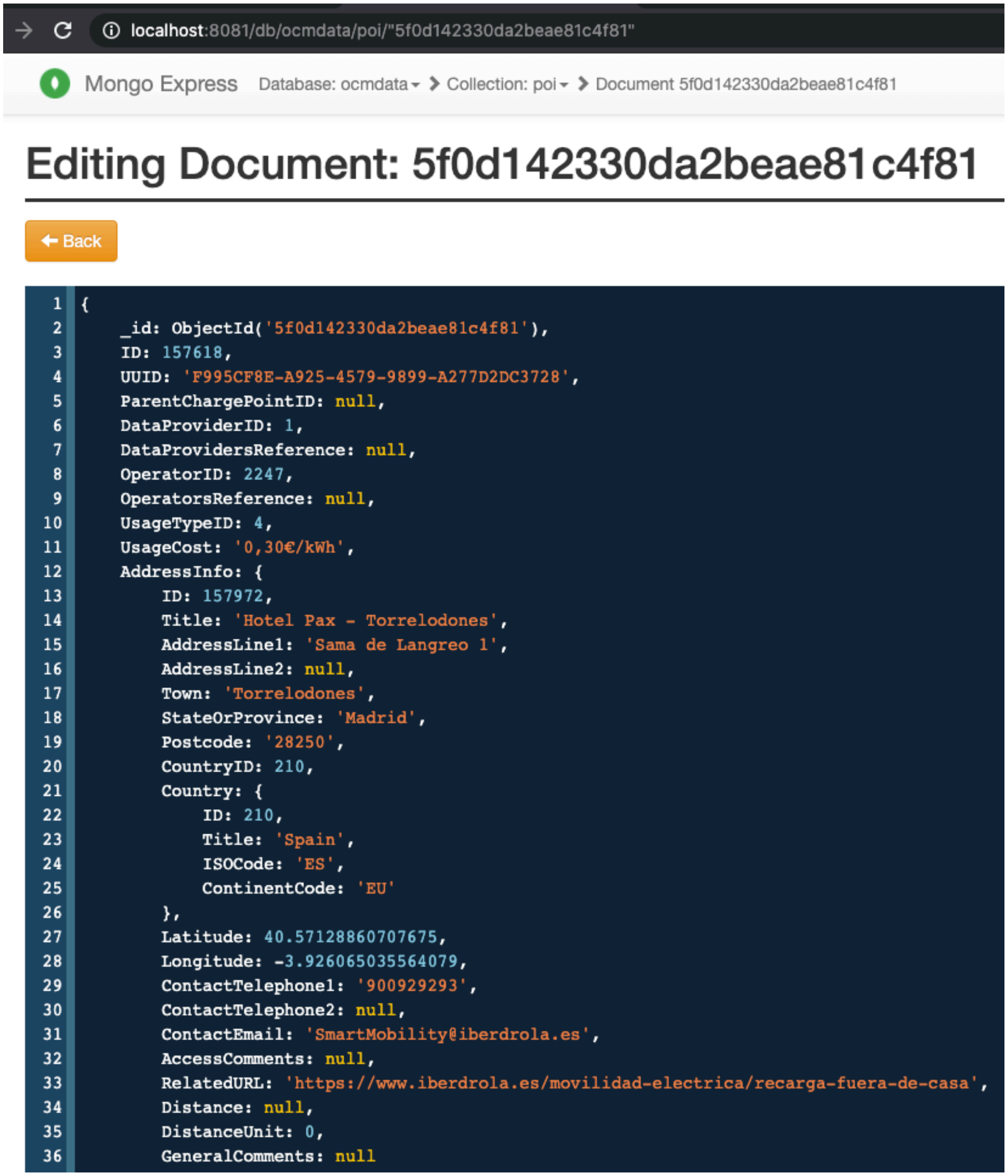
localhost:8081/db/ocmdata/poi?key=ID&value=157618&type=N

Viewing Collection: poi

ID | 157618 | Number

Delete all 1 documents retrieved

_id	ID	UUID	ParentChargePointID	DataProviderID	DataProvidersReference	OperatorID	OperatorsReference	UsageT
5f0d142330da2beae81c4f81	157618	F995CF8E-A925-4579-9899-A277D2DC3728		1		2247		4


localhost:8081/db/ocmdata/poi/"5f0d142330da2beae81c4f81"
Mongo Express
Database: ocmdata
Collection: poi
Document 5f0d142330da2beae81c4f81
Editing Document: 5f0d142330da2beae81c4f81
Back

```
{
    _id: ObjectId('5f0d142330da2beae81c4f81'),
    ID: 157618,
    UUID: 'F995CF8E-A925-4579-9899-A277D2DC3728',
    ParentChargePointID: null,
    DataProviderID: 1,
    DataProvidersReference: null,
    OperatorID: 2247,
    OperatorsReference: null,
    UsageTypeID: 4,
    UsageCost: '0,30€/kWh',
    AddressInfo: {
        ID: 157972,
        Title: 'Hotel Pax - Torrelodones',
        AddressLine1: 'Sama de Langreo 1',
        AddressLine2: null,
        Town: 'Torrelodones',
        StateOrProvince: 'Madrid',
        Postcode: '28250',
        CountryID: 210,
        Country: {
            ID: 210,
            Title: 'Spain',
            ISOCode: 'ES',
            ContinentCode: 'EU'
        },
        Latitude: 40.57128860707675,
        Longitude: -3.92606503556 4079,
        ContactTelephone1: '900929293',
        ContactTelephone2: null,
        ContactEmail: 'SmartMobility@iberdrola.es',
        AccessComments: null,
        RelatedURL: 'https://www.iberdrola.es/movilidad-electrica/recarga-fuera-de-casa',
        Distance: null,
        DistanceUnit: 0,
        GeneralComments: null
```

This does not prove I have the exact same data as the live website, but it's rather convincing! I will not likely have the same powerful queries via the API that I can run directly using mongo, such as record (or document) counts.

Step 7) Let's try a few using the mongo client session as we did in Step 4. Here I will query for all the Chargepoints that have "IsOperational=true" status in their nested Connections datapoint:

```
> db.poi.find({"Connections.StatusType.IsOperational":true}).count()

69260
```

So out of 145,179 points, less than half are operational! That is interesting. Perhaps we'd like to do some data cleaning on that to winnow the collection down to only active connections.

I run a query to see the opposite dataset, those where this value is false (or more interestingly, totally absent). Here is a subset, selecting only their numeric ID fields ("_id" returned by default):

```
> db.poi.find({"Connections.StatusType.IsOperational":false},{ID:1})
{ "_id" : ObjectId("5f0d111bf866c35e22065f92"), "ID" : 2961 }
{ "_id" : ObjectId("5f0d111ef866c35e22066159"), "ID" : 3622 }
{ "_id" : ObjectId("5f0d111ef866c35e220661ce"), "ID" : 4134 }
{ "_id" : ObjectId("5f0d111ef866c35e220662c5"), "ID" : 4391 }
{ "_id" : ObjectId("5f0d111ef866c35e2206631e"), "ID" : 4480 }
{ "_id" : ObjectId("5f0d111ef866c35e22066394"), "ID" : 4599 }
...
```

I load up record with ID 2961 in Mongo Express (as well as make a call to the live API with Postman) and see the following block:

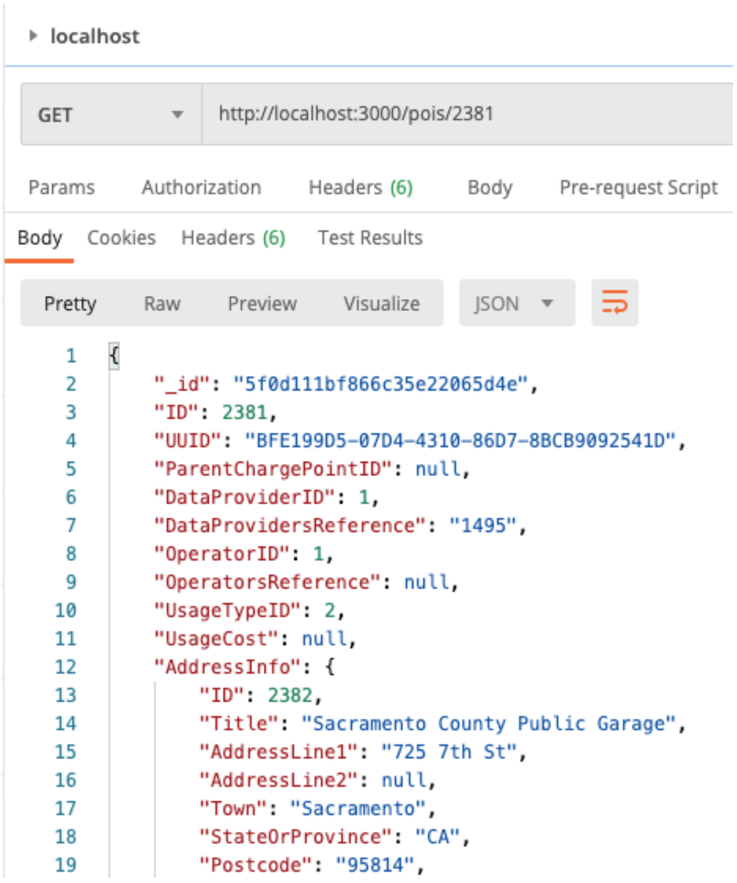
```
"StatusType": {
    "IsOperational": false,
    "IsUserSelectable": true,
    "ID": 100,
    "Title": "Not Operational"
}
```

Looking at reference.json, we see a matching block (this could also be checked in the older SQL version of the reference data):

```
"StatusTypes": [
    {
        "ID": 0,
        "Title": "Unknown",
        "IsOperational": null,
        "IsUserSelectable": true
    },
    {
        "ID": 10,
        "Title": "Currently Available (Automated Status)",
        "IsOperational": true,
        "IsUserSelectable": false
    },
   …
    {
        "ID": 100,
        "Title": "Not Operational",
        "IsOperational": false,
        "IsUserSelectable": true
    },
```

So while the data is non-normalized in this JSON representation, it should be possible to load this reference.json data in a separate collection in this database and run queries to match the IDs to make sure the Title and other values match on every document.

Step 8) Now we can finally get into some application development and 'scripting' with NestJS and Typescript. I have set up a local API so I can query the database using REST routes, either in my browser or again in the Postman app:



```
http://localhost:3000/pois/2381
```

```
{
    "_id": "5f0d111bf866c35e22065d4e",
    "ID": 2381,
    "UUID": "BFE199D5-07D4-4310-86D7-8BCB9092541D",
    "ParentChargePointID": null,
    "DataProviderID": 1,
    "DataProvidersReference": "1495",
    "OperatorID": 1,
    "OperatorsReference": null,
    "UsageTypeID": 2,
    "UsageCost": null,
    "AddressInfo": {
        "ID": 2382,
        "Title": "Sacramento County Public Garage",
        "AddressLine1": "725 7th St",
        "AddressLine2": null,
        "Town": "Sacramento",
        "StateOrProvince": "CA",
        "Postcode": "95814",
```

The code for this (found in the src/poi/poi.service.ts file) is:

```
async findByID(id:number): Promise<POI> {
    return this.poiModel.findOne({"ID": id}).exec();
  }
```

Now I will go back and forth between writing queries in the mongo shell and implementing service methods that do the same via API calls. For example, finding the number of distinct UsageCost strings, we find:

```
> db.pois.distinct("UsageCost").length
2731
```

I implement this in Typescript and Mongoose as:

```
async findUsageCosts(): Promise<string[]> {
    return this.poiModel.distinct("UsageCost").exec();
  }
```

The number of returned values from executing this API call is too many to display here, but looking them over in the browser, I see an opportunity for data cleaning, since I find variations on the same rate in the same currency and unit, like:

```
"$0.2 / kWh"
"$0.20 per kWh"
"$0.20/KWh"
"$0.20/kWh"
"$0.20/kWh; other tariffs for older cars"
```

Using the powerful MongoDB aggregation queries, I am able to group these and sort by their count with the following query to see the top UsageCosts:

```
> db.pois.aggregate(
... { $sortByCount:  "$UsageCost" }
... )
{ "_id" : null, "count" : 99667 }
{ "_id" : "0.00 jaarabonnement", "count" : 11519 }
{ "_id" : "Free", "count" : 11238 }
{ "_id" : "0", "count" : 1534 }
{ "_id" : "", "count" : 1349 }
{ "_id" : "0.00 afhankelijk van pas", "count" : 1019 }
{ "_id" : "free", "count" : 903 }
{ "_id" : "£0.108/kWh; for Polar Plus subscription members only", "count" : 735 }
{ "_id" : "£0.30/kWh; £0.15/kWh for Ecotricity energy customers", "count" : 572 }
{ "_id" : "Inclusive; for Polar Plus subscription members only", "count" : 528 }
{ "_id" : "0,40 €/kW", "count" : 500 }
{ "_id" : "DKK 5,50/kWh", "count" : 490 }
{ "_id" : "CAD 1.00/hour", "count" : 446 }
{ "_id" : "£0.059/minute; min £1.18; other tariffs available", "count" : 424 }
```

It is clear when we reach the values listed above, $0.20/kWh, we would not get the count of 5 we expect until some transformations have been performed (e.g. stripping whitespace). However, since there will be a case of *at least* this much variation for nearly every one cent increment or fractional currency figure (the unit $1.00 has 46 variations) the regular expression required to handle this is beyond my comprehension right now.

Conclusion & Future Work

There are obviously many paths we could go down to inspect, analyze, and manipulate this data from this point forward using these powerful tools. However, I think I have reached the limit of what I can put into this project for the time being. Let's review what has been achieved so far:

1) Understanding the structure and content of the data, including all foreign key relationships as diagrammed in Step 4.
2) Manipulation and faceting of subdocuments using OpenRefine such as AddressInfo and ConnectionTypes (Step 2).
3) Re-creation of the SQL-based schema from MS-SQL archive in a local MySQL instance as proof of concept (Step 4).

4) Restoration of MongoDB database backup into a local Docker container for the purposes of querying and manipulating using the mongo client.
5) Connecting that MongoDB to a Typescript application and NestJS framework for exposing these functions via a REST API

This last step is key because it makes these powerful results available to the public. The value in this dataset is its size and scope. It is curated by designated editors within the community, country by country, many of which may not have programming skills or database manipulation skills. I can make some of these routines available to just these admin users simply by typing a URL into their browser.

I am likely to continue working on this dataset in the future after this course, so I can help contribute to the Open Charge Map project and learn more about EV charging infrastructure. Perhaps I could help them to normalize their MongoDB data with some of these queries and reporting routines that were never replaced when they converted from MySQL (stored procedures would not translate to this database). I could also provide or suggest new API endpoints that provide answers to interesting questions for the general public. I may go so far as providing an alternative API as a proof of concept for the power this Typescript framework provides.