GeoCivics

An Open Data Project for Humans

Final Report, EECS 395: Senior Project

James Hochadel, Justin Plasmeier, and Zachary Jaffee

04/28/2017

1. Table Of Contents

2. Abstract

A wealth of civic data exists on the web across a variety of sources and formats. Location is common to nearly all of these data sets. Thus, it should be possible to leverage this common feature to gain access to broader insights across otherwise disparate data sets. The GeoCivics team is building data infrastructure to consume location-based civic data, normalize the data around location, and provide an intuitive user interface to access the data. This will allow users to contextualize data in terms of location as well as in terms of other data available for a given location.

3. Introduction

3.1. Background

Government agencies have been collecting various data on wealth, crime, health, education and the environment for over a century, and have begun making it available digitally as Open Data during the past decade. Since 2013, all newly-generated government data “must be made available in an open, machine-readable format” [1]. This data has the potential to inform policy and voting decisions, but most citizens are not aware of its availability and potential utility. Additionally, due to the large number of agencies reporting on such metrics, this data is represented in many different ways.

Citizens and legislators need to have a shared perception of reality in order to hold productive conversations around policy. Given social media’s tendency to damage that shared perception, availability of relevant and recent data is more important than ever. The team sees value in making access to existing data easy, being able to flexibly intake newly available data, and making sure that data is up-to-date.

3.2. Project Goals

The GeoCivics project has two goals. First, the team aims to make it easy to compare open data in a geographical context, so that users may draw conclusions from multiple types of information for one geographical region. Second, the team will make it easy to take advantage of the newest data quickly by making it easy to load new data sets and update old ones.

To accomplish these goals, the team will create a database-backed web application centered around a map. The application will consist of three parts:

1. A web application centered around a map, which allows users to load multiple data sets,
2. A geospatial database normalized around location and populated by an ETL pipeline, and
3. An application server that functions as the database access layer, retrieving data from the database to be presented on the frontend.

The GeoCivics database will implement a Star Schema to allow queries to join on location. To populate the database, the team will create an ETL pipeline to acquire data from sources such as data.gov and load data into the database. The database access layer will translate user filters to DBMS queries. The user interface for the data will consist of a map based UI where users can apply filters and combinations to data and see the results overlaid onto a map. Features could potentially be represented as points (clustering), boundaries (polygon), or gradients (heatmap).

3.3. Related Systems

Data.gov, the most well-known repository of open government data, has been online since 2009. Several projects on the web have used its services to attempt making open data more accessible. Among the most well known are:

- DataUSA.io, which is branded as “the most comprehensive visualization of U.S. Public Data.” However, the newest data is from 2015. GeoCivics will emphasize automatic updates which will ensure that the freshest data is being displayed.
- City-Data.com, which uses information from data.gov, and enables comparison of data in a given city over time.
- American FactFinder, which allows lookup of information provided by data.gov.
- USAFacts, a USA civic data project focused on government spending and political inquiry. This service was released during the development of GeoCivics.

3.4. Advantages

The project’s main focus is perfecting the ETL pipeline. New data will be able to be added and updated quickly and with minimal manual effort. Thus, the main advantage is that the ETL pipeline will allow the service to scale easily to more datasets than if the team focused on custom-building the site to accommodate specific sets. Additionally, the ETL pipeline will poll the data sources in order to maintain the latest available data.

3.5. Disadvantages

Other projects using open data have two main advantages: Several have been operating for multiple years, and second, several support rich visualizations for comparing data between cities. DataUSA in particular already supports comparing data between two cities using a range of graphics. Additionally, the team’s MVP specifies support for 2-3 data sets; this is far fewer than are available on competitive products.

4. Application

4.1. Constraints

The automatic updating process is reliant on data being posted and existing in a specific location. If the URL of the data file changes, then the pipeline is broken. The GeoCivics team does not control the data source, so if any government agency performs any kind of take down, there is no control over that point of failure.

Additionally, data may be inconsistent and difficult to work with. The team is focusing on selecting datasets conducive to the ETL process and visualization goals. Regardless, each individual dataset will require its own ETL implementation, with potentially limited abstraction across datasets.

Lastly, the various underlying systems we are using have different constraints of their own. Postgres for example has constraints such as not allowing more than 1600 columns in a table, where the data held within a single row cannot exceed 8 kilobytes. This in turn has required the team to have to find ways to cull the data sets we are using in order to continue using postgres within our system.

4.2. Challenges

4.2.1. Extract

As part of the extraction process, a cron job will poll the URL of the data set. Data sets which are hosted on data.gov specify the last modified date of the data (Figure 3.0)

Harvest Source Title	Education JSON
Data First Published	2015-09-12
Homepage URL	https://collegescorecard.ed.gov/
Language	en-US
License	https://creativecommons.org/publicdomain/zero/1.0/
Data Last Modified	2017-01-13
Program Code	018:000
Publisher Hierarchy	U.S. Government > U.S. Department of Education > Office of Planning, Evaluation, and Policy Development

Figure 3.0: The "Data Last Modified" field on a sample data set from Data.gov

Certain data sets may not provide this information. In these cases, alternative options will be pursued.

When updating data sets, the old data must be accessible while the new data is processed. If possible, only the newly added data should be processed and appended to the respective tables. Otherwise, the entire data set will need to be processed again.

The team will also need to write code to properly transform the raw CSV/JSON files into Python objects. The team is evaluating whether to use the standard CSV/JSON libraries, or something already built into an ORM to write directly to the database.

4.2.2. Transform

The team will design schemas to normalize data and allow for cross table joins. The project will include a library of data cleaning operations specified for each incoming data set. Additionally, Geocoding/Reverse Geocoding will be used to translate location between street/zip code/etc. and latitude/longitude points. This will provide the data of both point and polygonal representations of a given location.

4.2.3. Load

The load process will require inserting a large number of records into the database. The process will be batched and tolerant to failure. The team will build on the existing GeoAlchemy 2 ORM to perfect the loading process. However, GeoAlchemy 2 is open source software in relatively early stage development (version 0.4.0) and as such may be lacking in documentation or features.

4.2.4. API

The client will need to retrieve data from the database according to the users' filters and queries. The server will be able to accept requests with the specified filters and return the corresponding GeoJSON to the client. The UI must make valid requests to the server and thus will need to receive information about the available operations from the server upon initial page load. The UI itself will need to reflect these operations as well.

4.2.5. Frontend

All of the geospatial data processing will happen on the server. The frontend will use MapboxJS to display GeoJSON from the server on a map. MapboxJS supports many features. Upon hovering over a data point, the user will see more information about the data at that point. Points will cluster when the map is zoomed in and out. The team will write JavaScript to link the filters and queries to their corresponding URLs. As a stretch goal, users will also be able to download the processed data in a static file or access the data through an API.

4.2.6. Performance

- Biggest concern is within the extraction process
    - Network I/O for downloading data
    - Diff on incoming datasets to load only new records, or must reload them entirely each time?
    - Data serialization/deserialization -- metrics?
    - If the process takes longer than the update period, the team may be continuously updating the data or run into a race condition where the newest data is being pulled in with the newer data (this needs to be addressed).
- DBMS Performance
    - How long does it take to load n location records?
    - Data size
- User Interface/API
    - MapboxJS will bottleneck the UI performance.
    - Using MapboxGL.js should help with this

4.3. Requirements Specification

4.3.1. Database

- The DBMS used will be required to support relations, functions, and indexing of spatial data. The PostGIS extension provides the spatial types and queries required on top of a well-known RDBMS, PostgreSQL.

4.3.2. ETL

- General library for common tasks
    - This will likely involve a set of libraries or abstractions on top of existing libraries to parse CSV and JSON data to best represent the data being used as objects
- Specific implementation for each data set transform
    - The transform step will be specific to the individual data that is being extracted. Generally speaking, this will at most involve dropping irrelevant or redundant columns.
- Cron Job
    - Periodic Cron Jobs will be run to extract the data source on each update. data sets will be polled weekly for updates.
- Geocoding and Reverse Geocoding
    - Mapbox supports this but is limited to 600 requests/min for the free version.
    - Geocoding data will be stored in a separate table to avoid unnecessary API calls when possible.

4.3.3. UI/Database Interface (API)

MapboxJS supports loading GeoJSON asynchronously from a given URL. Upon the initial page load, the server will provide URLs corresponding to the filters and queries available in a common format based on the REST [2] specification and HATEOAS [3]. The server and front end will be bound to a common JSON schema such that the front end can translate a given response into a UI representation associated with the URL. When a filter is selected, the front end will provide the corresponding URL to MapboxJS, and that data will be rendered on the map along with any other UI state changes associated with the URL (e.g. point vs. polygon vs. gradient). To manage state transfer via HTTP requests and responses, the team will implement an API for the necessary operations. The API will use Flask or Django REST framework (both are Python implementations of a RESTful API framework). The API client on the front end will be written in JavaScript.

4.3.4. UI

Based on the project goals and the UI methodology described in section 4.4, the user interface will be implemented as a web application centered around a map. The UI will require the following functionality:

- R-UI-1: A map, which takes up the majority (~90%) of the browser window and will present the datasets the user selects.
- R-UI-2: Clicking a point or region on the map will open a detail view of the datum being visualized at that point or region.
- R-UI-3: A navigation bar must exist at the top of the page.
- R-UI-4: A floating panel must allow users to choose which data sets they want to visualize on the map. Clicking a dataset will render it on the map.
- R-UI-5: A legend indicating how the data sets are represented on the map
- R-UI-6: A settings panel, allowing users to modify such settings as how data is presented (where appropriate). For example, a data set could be presented as point representations, or as a “heatmap”.

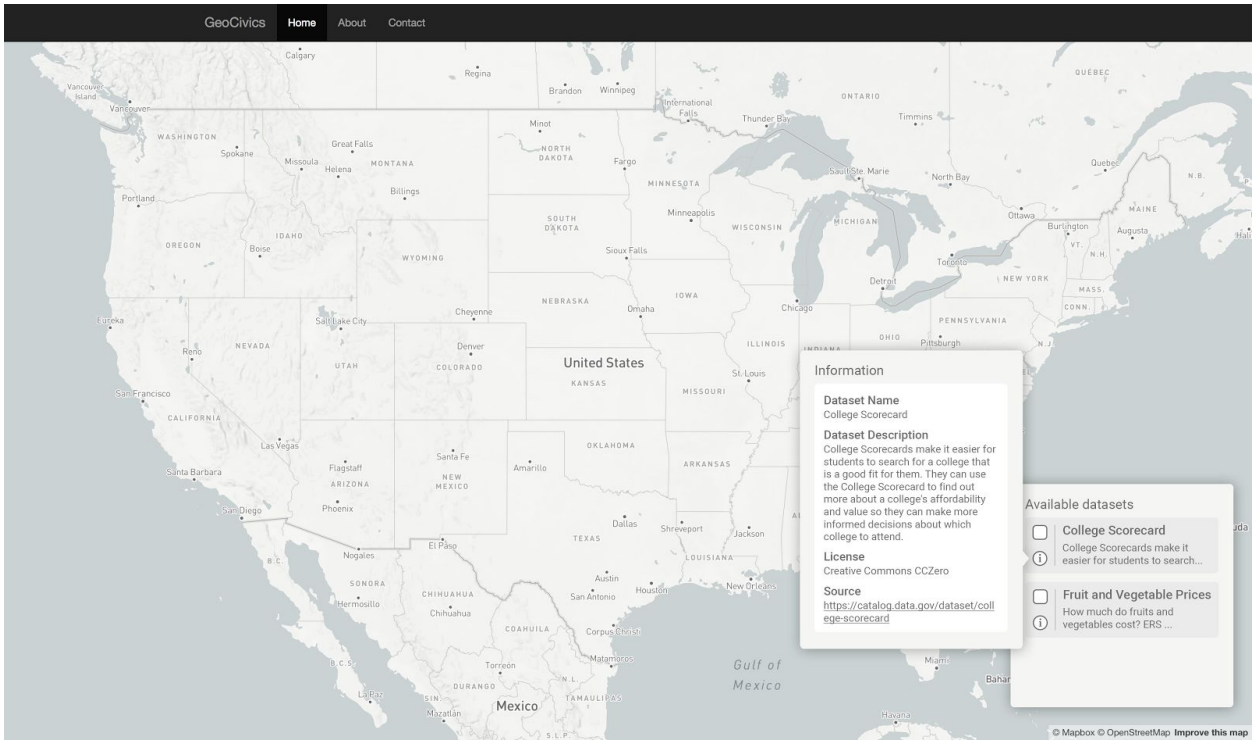


Figure 1: A mockup of the GeoCivics UI, showing R-UI-1 (map) and R-UI-4 (dataset selector). A summary of the "College Scorecard" dataset is shown in a panel.

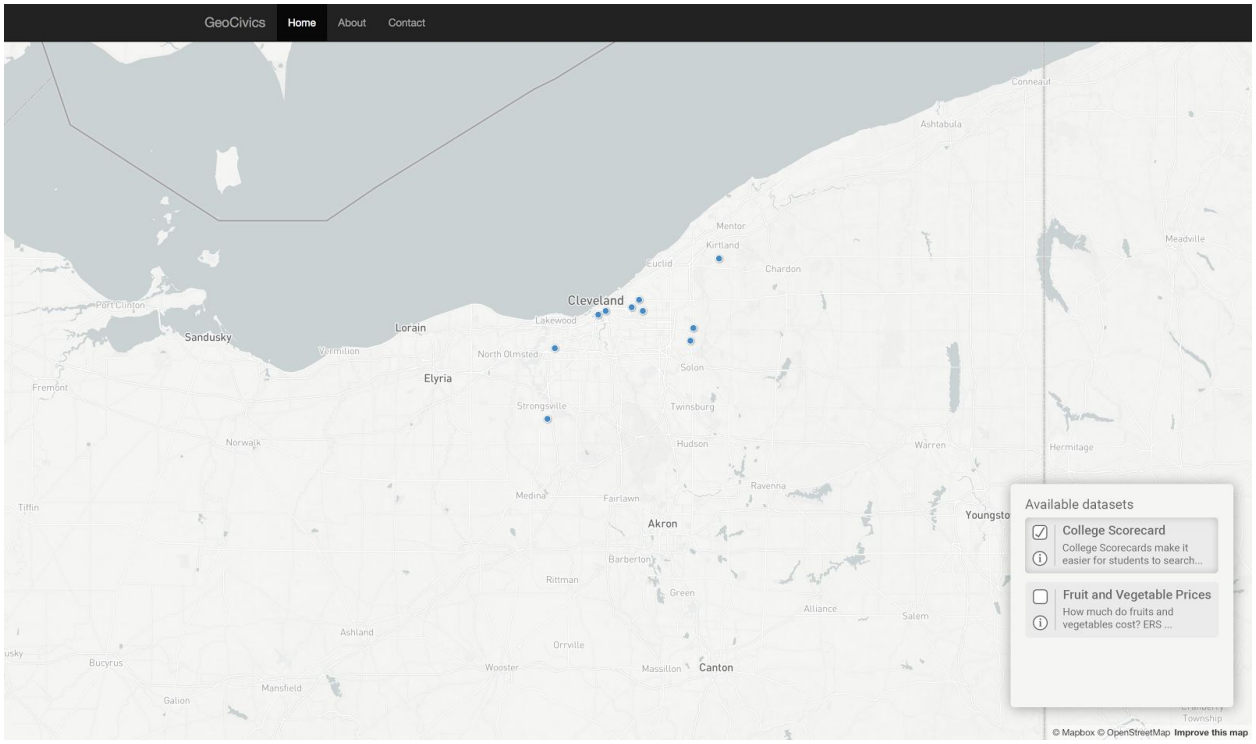


Figure 2: A mockup of the GeoCivics UI, showing R-UI-1 (map) and R-UI-4 (dataset selector). One dataset, "College Scorecard," is rendered on the map.

Since the available operations are provided by the server, the UI will need to be able to automatically represent queries and operations received from the server. Otherwise, the map will be front and center.

As a stretch goal, in addition to viewing the sets visually, users should be able to access the raw data. Users will also be able to combine multiple data features into a single representation.

4.4. Methodology

4.4.1. Database

Some of the special considerations for the database include:

- Making sure to update the duplicate tables every time a schema update is run on the database.
- Making sure to do a complete data dump of the oldest tables prior to re running the ETL process.
- In the event of needing to do a complete reinstall/setup for the first time follow the instructions on this link here, which installs postgresql 9.5, PostGIS 2.2, PGAdmin3, pgRouting 2.1 and additional supplied modules including the **adminpack** extension

4.4.2. ETL

- The team has decided to modify this step such that there are two tables and a configuration file to specify which table the API is reading from. Essentially the way to visualize this step is as a blue/green deployment, where the CF router is switched by modifying configuration file which the API would then point to.
    - A stretch goal would be to make this non blocking and do a blue green deployment of the API upon each configuration change, however, the team does not see a millisecond of downtime as a major problem at this time.
    - Upon a data update, and a verification check, a procedure will run to drop the data from the old table, so that it can be used for a future update. The team believe that this will be the best way to preserve data integrity across multiple updates.
- Batch loading data into the database.
    - The load process can be broken down into two parts, loading the schema and loading the data. To do this we have two separate psycopg2 sessions in which we assert and commit that the schema is loaded properly and then we load the file in as one large batch. If there are any disparities between any of the rows within the CSV file and the schema, psycopg2 will not commit any of the new data to the database.

4.4.3. API

The purpose of the API is to facilitate communication between the browser client and the server. The frontend uses MapboxGL.js to display geospatial data onto a map. This data is asynchronously loaded from a given URL. Thus, the state of the application is defined by the URL which the client is loading data from and any metadata associated with the data at that URL.

The API is designed to accommodate these requirements while following REST and HATEOAS principles. REST stands for *Representational State Transfer* and describes a style of building web applications around various HTTP verbs (GET/POST/PUT/etc.) as stateless operations on well-defined resources [1]. HATEOAS stands for Hypertext As The Engine Of Application State and describes the requirement that each response from the server contains the available URLs of the next application state [2].

The API has two resources- query objects and queries. In terms of their models, there is a one-to-one relation between query objects and their queries. Query objects contain the metadata of the associated query. Queries contain the actual query and the URL which Mapbox will find GeoJSON from running the query.

The resources are separate in accordance with REST principles because the response from a GET request to the URL of a query should return its result (GeoJSON) and the response from a GET request to the URL of a query object should return the metadata of that query. It would conflate REST to have different URLs for different representations of the same resource, so instead there is a URL for each representation of the resource, which are in turn different resources in order to avoid confusion between an entity and its representation.

4.4.4. UI

The first project goal is to “make it easy to compare open data in a geographical context, so that users may draw conclusions from multiple types of information for one geographical region.” From the user’s perspective, this will be primarily achieved through the UI. As such, it will be designed around the following principles:

1. Design so that the data is front and center, and displayed in the most useful way possible.
2. Follow the Pit of Success principle: The “right” way to perform a task is also the most obvious way.
3. Build ancillary features in service of principle #1, interfering with it as minimally as possible.

4.5. Software Design and Development

4.5.1. Software Architecture

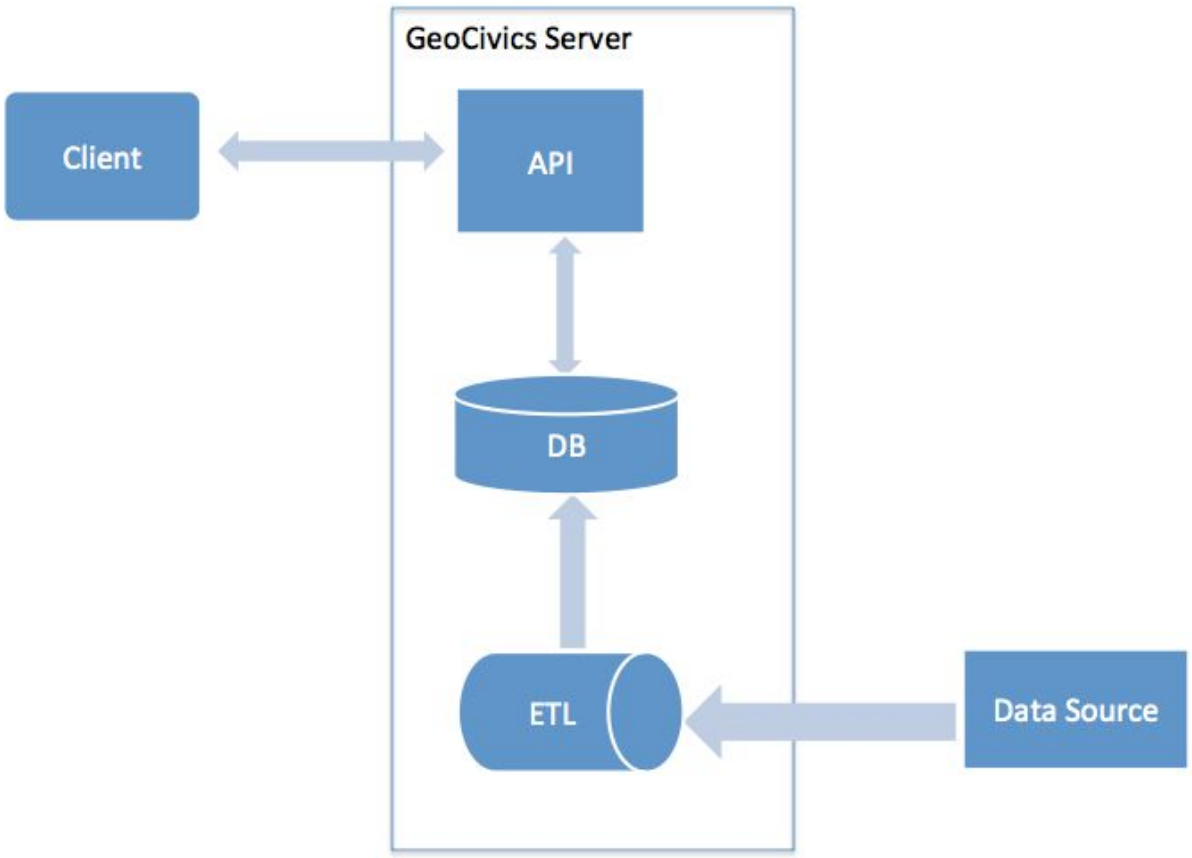


Figure 3: The high-level software architecture for GeoCivics.

The above diagram represents the architecture of the project. The client (a web browser) communicates with the server via an API. The API connects to the database via an ORM, which exposes a database session to the API. Thus the API can execute database transactions based on user input, as well as administrative commands. The database is populated via the ETL pipeline. The ETL pipeline pulls in data from a data source and infers a schema from the flat file. Then, various transformations on the data are applied, potentially creating new columns, before loading it into the database. The schema extraction code and the loading code are kept consistent across the Asset model.

4.5.2. ETL Pipeline

Asset Schema

GeoCivics uses an asset schema to provide a common metadata reference of which datasets should be downloaded by the ETL pipeline. While the data is stored in the csv file, these objects refer to that data and contain all of the necessary metadata. For example, the transformations to apply to the data are stored here, as well as the file paths of directories for each pipeline stage. This is implemented via a Python object `Asset` and has the following properties:

- url
- filetype
- filename
- headers (from the HTTP response)
- asset_path
- data_path
- xform_path
- load_path
- local_filename
- csvheader
- transformations (A list of Transformation objects)

Instances of Asset and Transformation instances are defined in the Python module `geocivics_assets.py` and determine how the database is constructed. Thus, one can easily implement their own instance of the ETL pipeline for their own data and transformations.

Extract

The extraction process, as it applies to downloading new data, comprises the following actions:

- Read from an asset schema (more details below)
- Download the asset file from the specified URL
- Compare the downloaded file with the existing file via cyclic redundancy check
- If the files are different, replace the old with the new
- Return a list of updated files

This is implemented in the Python module Rone, which uses urllib to download files and cksum for the comparison. The result is that the latest files are available for the next stage in the pipeline.

The next stage in the pipeline is an optional step to remove extraneous columns from a csv file. This was borne out of necessity; the College Scorecard data set is incredibly wide (1700+ columns) and many of the columns were not useful for the purpose of GeoCivics. All columns matching a given set of regular expressions are removed from the csv file before entering the next stage of the pipeline. This is implemented in the Python module cull, which uses pandas to manipulate large csv files.

Next, a database schema is inferred from the csv file. The Python module `extract_cscard.py` reads the csv file headers and infers a data type based on the values in that column. After the file

itself has been processed, the extra columns resulting from the transformation are appended to the original columns, including the data. The data is a result of a function that takes a row as input and returns a value. The column name, datatype, and function are encapsulated as a Transformation object along with an ordering, which are referenced from the Asset class (see more details below). So, for each data asset, one can specify any number of transformations, and the columns, type, and values will be constant across schema inference and the actual transformation.

Transform

The data in the downloaded files is not yet ready to be loaded into the database. In most cases, there are fields for location entities but naive data type inference cannot translate these values into valid location data types. More precisely, if a data set contains latitude and longitude points, the values will be inferred as double/floats. In order to store a value for a location coordinate using location types, a transformation must be a applied to the data. Transformations are defined per data set using the transform schema to enforce consistency between schema extraction and the data itself.

Transformations

When data is transformed, multiple steps in the pipeline must be aware and synchronized to all modifications to the data, especially in terms of the columns and their type. For example, when a column named “coordinate” with type geo point is added to the data set, both the schema extraction and transform steps need to account for this. The schema extraction component needs to know the name and type of the extra column. The transformation component needs to combine the latitude and longitude fields into valid geo point values. If these operations are not synchronized, the data will fail to load or worse- be inconsistent with the schema. Thus, the transform schema exists.

The Transformation object is implemented in the Python module `xform_schema` and has the following properties:

- column name
- column data type
- a function
- order

System Architecture

Each of these components is imported by `geocivics_assets.py` and called in sequence by a single function. The team decided to design the Python module dependencies in this order so that multiple databases could be easily supported in the future.

5. Project Management Plan

5.1. Milestones

The Senior Project due dates for GeoCivics were as follows:

- March 8: Progress Report due
- March 10: Progress Report presentation and discussion
- March 26: Research ShowCASE Poster Abstract due
- April 21: Research ShowCASE Poster Presentation
- April 28: Final project report and source code due

To make steady progress towards the GeoCivics final product, the following milestones were defined:

5.1.1. M1: Prototype, due March 8

The M1 proof of concept consists of two parts: A basic Map UI and a design for the ETL pipeline and data model.

First, the Map UI will display GeoJSON from a database independent of the ETL pipeline. The team will make certain design decisions while creating this prototype that will affect the final product. Second, the ETL pipeline and data model will be outlined in a well-specified design document.

5.1.2. M2: Preliminary ETL, due March 17

At M2, the team will be able to use one database with the ETL pipeline as specified in the design doc written for M1. The UI will interact with the database access layer to show the data on the map in some basic form.

5.1.3. M3: Expanded ETL + UI, due March 24

By M3, the UI will be able to display all relevant data from the first database, and the ETL pipeline will work at an M2 level with at least one additional database.

5.1.4. M4: MVP, due April 8

The M4 goals define the GeoCivics MVP; the project will be considered a success if these are achieved. The team will build:

- A map UI that presents 2-3 data sets and supports various queries.

- A database access layer that accepts queries from the frontend and requests the appropriate data from the database
- A database, loaded and normalized through the ETL process, containing 2-3 data sets

5.1.5. S1: Stretch goals, due April 21

The S1 goals will make the GeoCivics feature set more complete and make the product more useful. The goals include:

- Scale flexibly to more datasets.
- Better ETL tooling.
- Implement a preliminary automatic update feature that regularly polls data sources for updates and merges changes into the database
- Build a more sophisticated front end with support for more complicated queries.

5.2. Meeting Schedule

The GeoCivics team has been meeting on Sundays at 10:30am for general purposes. The team will continue to meet at this time, as well as schedule additional meetings as necessary.

5.3. Technical Management

The GeoCivics team used Git for version control, Trello for task management, and the Waterfall model for project management.

5.4. Division of Labor

James is responsible for front end development. Justin is responsible for API development. Zach is responsible for the database and has been working on the Extract step of the ETL process. Since the front end and API are nearly completed, James and Justin will assume responsibility of the Transform and Load steps as development continues. Justin will continue to organize and run meetings and James will work on finalizing the reports. Zach has graciously volunteered to handle all deployment, operations, and system administration tasks.

5.5. Reflection

The meeting schedule worked well and has been effective in making sure that responsibilities are assigned and completed in accordance with upcoming deadlines. For most meetings, the team creates an agenda prior to the meeting containing relevant discussion topics and upcoming deadlines. Through each meeting, notes are added to the agenda summarizing the discussion. This is usually somewhat helpful though meetings seem to work about as well with or without a formal agenda and note-taking.

The team has a group message thread on Facebook Messenger for informal communication and uses Google Hangouts when meeting in person is not necessary. Trello has not been as effective as the team had hoped. However, task synchronization is becoming increasingly important as the team starts to connect services together. This makes the process of creating and updating cards in Trello more valuable to the team, and doing so will be necessary in order to avoid confusion. Overall, development is progressing smoothly and the team is on target to meet the project milestones as planned.

6. Testing and Evaluation

The only testing done so far has been verification that components work as expected. Any tests for the API would be trivial (i.e. GET returns a 200 response). Though, more complicated cases will be considered for testing in the future.

7. Lessons Learned

The GeoCivics team has learned several lessons through its development efforts.

7.1. Justin

An interesting lesson that Justin learned was about a design paradox that seems to manifest within data projects. When writing code for a single data set, one tries to write code in a way that could be generalized to other data sets. However, this can be worse than simply writing less general code for each data set, before reviewing and refactoring. Justin laments that only one data set was used as the example to work with, and while the team had been designing with generality in mind, it's impossible to tell exactly where you can be general, where you have to be specific per data set, and how to organize that.

Additionally, Justin learned how broad GIS is. At the beginning, the team set out to be able to support geometry such as polygons and gradients, but points were enough trouble to keep the team busy. Supporting polygons would require a much more complicated transform process. As noted in section 9.4, further work in providing meaningful queries from the data would require GIS knowledge that is well beyond the scope of what the team knew at the beginning of the project.

7.2. Zach

Zach primarily focused on the database layer and various aspects of the ETL process. One of the biggest lessons learned was how to deal with the various inconsistencies within a dirty dataset and configure all of the various fields needed such that data can be properly loaded. His prior experience with data engineering had involved well structured logs and consistent database tables, rather than having to do large amounts of data cleaning.

7.3. James

Reflecting on the software development process, James identified the following core lessons:

1. Perform research to set realistic goals. Without an understanding of the complexities of client-side Javascript development, the UI aspects of project milestones were based on guesswork and past experience. Estimates should have been based on more concrete benchmarks. For example, to estimate the time required to develop each React component in GeoCivics, James should have implemented a simple React app, tracked the time required to complete it, and based his estimates on that data.
2. Pursue structured learning. To successfully achieve all UI-related goals for the GeoCivics project without knowing Javascript, the following steps needed to be taken:
    a. Develop an understanding of the JS language
    b. Research client-side Javascript frameworks and choose one that suits the purposes of GeoCivics
    c. Develop a conceptual understanding of the framework's approach to client-side development, as well as concepts for libraries in its ecosystem
    d. Implement GeoCivics, learning additional libraries along the way

    This was not achieved successfully. Insufficient work in early stages of this process (e.g., developing a conceptual understanding of a framework) was paid for in subsequent stages. Creating a comprehensive learning plan involving small projects in JS, React, and MapGL would have made missed deadlines less likely, and (critically) provided a better picture of exactly how far behind James was at any given point in the project.

The high rate of change in the Javascript library and framework landscape also complicated efforts to develop GeoCivics; documentation for Webpack, React MapGL, and other packages was often out of date or nonexistent.

8. Contributions

Justin focused on the the API, as well as the architecture, and several components of the ETL pipeline. Zach focused on the operational side of the project, miscellaneous database work, and the ETL process. James was in charge of designing and implementing the UI component of GeoCivics.

9. Final Status, Conclusions, and Future Work

The GeoCivics team was able to complete a majority of the requirements on the ETL pipeline. Assets can be defined by a URL and all of the necessary information such as to populate a table in a GIS database. Certain planned functionality was not implemented. In some cases, this was due to a shift in requirements. For example, geocoding and reverse geocoding turned out to not

be necessary. The data sets already contained records for lat/long points. Implementing geocoding would require a coherent way of integrating data across types which was farther beyond the scope of this project than originally considered.

9.1. Were the application specifications achieved? If not, why not?

Some of the GeoCivics application specifications were achieved. Generally, the teamed reached the M2 milestone of successfully loading a single dataset from a third-party source. The UI does not yet meet the standards set out in this milestone.

9.2. What testing/evaluation tools have been used?

No automated testing or evaluation tools are currently used by GeoCivics. The only input into the program is defined in the Asset schema and thus the scope of testing is vastly reduced. In every stage of the ETL pipeline, the data from the previous stage is still available. If one stage fails, simply throw an error and try again. In the future, this would need to be monitored closely to ensure that new data was always being loaded without failure.

9.3. Which frameworks/tools/libraries were used, dropped, added, and why?

9.3.1. Database

The primary GIS database for GeoCivics uses PostgreSQL and PostGIS as planned from the beginning.

9.3.2. ETL

The ETL Pipeline uses several standard Python libraries including:

- os (for operating on files)
- shutil (for copying files)
- urllib (for downloading files from the Internet)
- csv (for handling csv files)
- re (for regular expressions)
- pandas (for processing csv files)

9.3.3. API

The API uses the Django REST Framework to send and receive HTTP requests from the front end.

9.3.4. UI

The GeoCivics UI demonstrated during the team's Progress Report Presentation was originally written as a simple static web page with inline JavaScript. The site used MapboxGL for its map and layer support.

It was clear that certain tasks— making calls to the GeoCivics API and storing data sets, for example— would be repeated throughout the application. To accommodate the expected complexity of the data and operations, the UI was partially re-written in Backbone.js (a lightweight Javascript MVC framework) and Require.js (which provides support for modules). Unfortunately, these frameworks were later dropped due to mutual incompatibilities that significantly slowed progress.

Backbone and Require were replaced with React (for writing UI components), Babel (for compiling React's JSX), and Webpack (for bundling modules into a single production-ready .js file). Module support was provided by ES6, the latest release of Javascript.

Switching to the React ecosystem yielded several benefits. First, using React with Webpack and ES6 enabled more robust support of modularized code than Require. Second, several React components that encapsulate MapboxGL are publicly available and actively maintained, so Mapbox fit naturally into a React application. The primary tradeoff of using React was complexity; the React wrapper for MapboxGL, for example, provide difficult to use, and lacked documentation besides examples.

9.4. What should be done next?

9.4.1. ETL

Extract

Several of the components have clear next steps. Rone, the download component, should store the Etag of the resource in a log somewhere as to avoid running cksum on the file when possible. The cull component (to drop columns) could be rewritten without using Pandas for greater transparency and flexibility.

Transform

The transform code could be written to be performed more efficiently. The code makes several passes reading and writing the csv file. This was done for the sake of ease of development, and for writing modular functions. In principle, every transformation could be applied on a single pass. The team would consider refactoring this if performance becomes an issue.

Load

We would make it so that at the time of load, we are also modifying a join table such that we could easily visualize datasets that share similar location points easily.

9.4.2. API

Currently, the API contains code to structure responses from the database into GeoJSON. This relies on the convention of putting geometry values at the beginning of a query and returning the rest of the values as entries in “properties” as their column name and value. This approach works fine for simple use cases, but a more robust GeoJSON serialization would be useful as queries increase in complexity. It is an interesting thought to consider deriving API serialization rules from transformations, but this is unlikely to be productive.

9.4.3. UI

The GeoCivics UI lead will continue working on the user interface until the team demonstrates its project on Tuesday, May 2. The team intends to have a complete demo supporting at least one data set.

9.5. Production Readiness

GeoCivics is not currently ready for production. Pursuing the improvements in section 9.4 would move GeoCivics substantially towards production readiness.

9.6. Feasibility & Future

The project is still feasible. Further work would require either an in depth study of GIS techniques, or finding someone with expertise in GIS. However, the team is confident that there is value in software that can automatically and repeatedly convert a csv file into GeoJSON resources.

In the future, the team would consider pivoting GeoCivics to be a library for processing csv files into valid GeoJSON data.

10. References

1. U.S. General Services Administration. 2017. About Data.Gov. (2017). Retrieved March 7, 2017 from https://www.data.gov/about
2. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
3. Pivotal. 2017. Understanding : HATEOAS. (2017). Retrieved March 7, 2017 from https://spring.io/understanding/HATEOAS
4. Jeff Atwood. 2007. Falling Into The Pit of Success. (August 2007). Retrieved March 8, 2017 from https://blog.codinghorror.com/falling-into-the-pit-of-success/

11. Appendices

11.1. Repositories

The various components of GeoCivics are located in the following repositories:

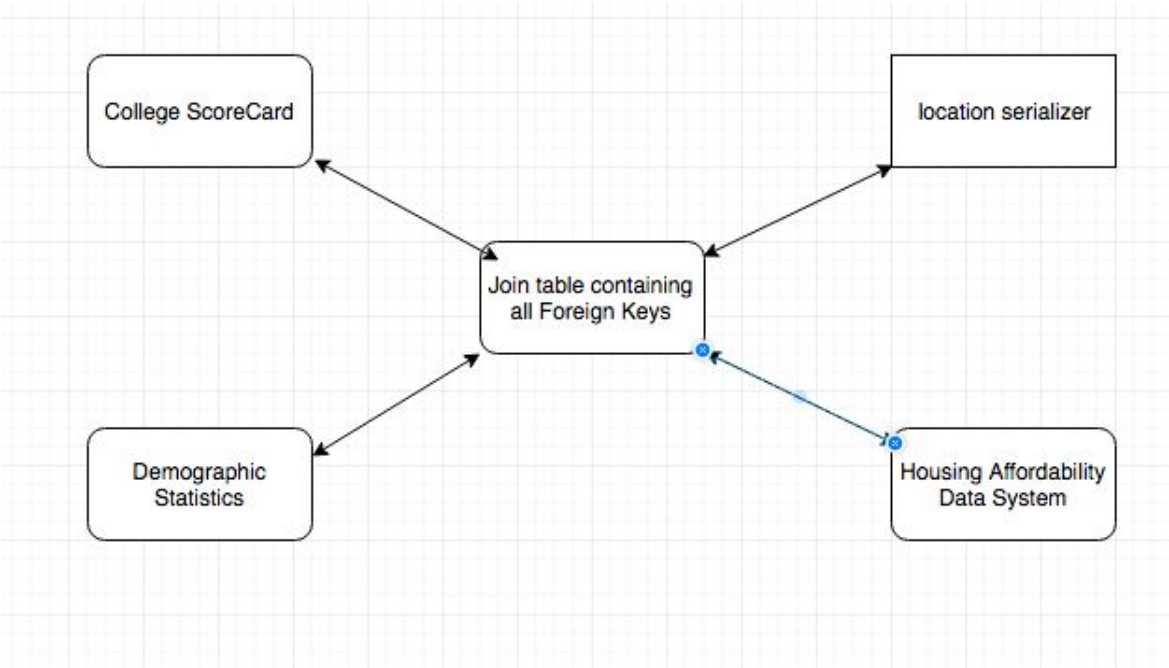
- GeoCivics (Asset Schema) : https://github.com/justinplasmeier/GeoCivics
- UI: nebel-react on github
- Rone (Extract - Download) : https://github.com/justinplasmeier/Rone
- cull (Extract - column removal) : https://github.com/justinplasmeier/cull
- Faraday (Transform) : https://github.com/justinplasmeier/faraday
- srproj (Extract Schema and Load) : https://github.com/justinplasmeier/srproj

11.2. Database Design

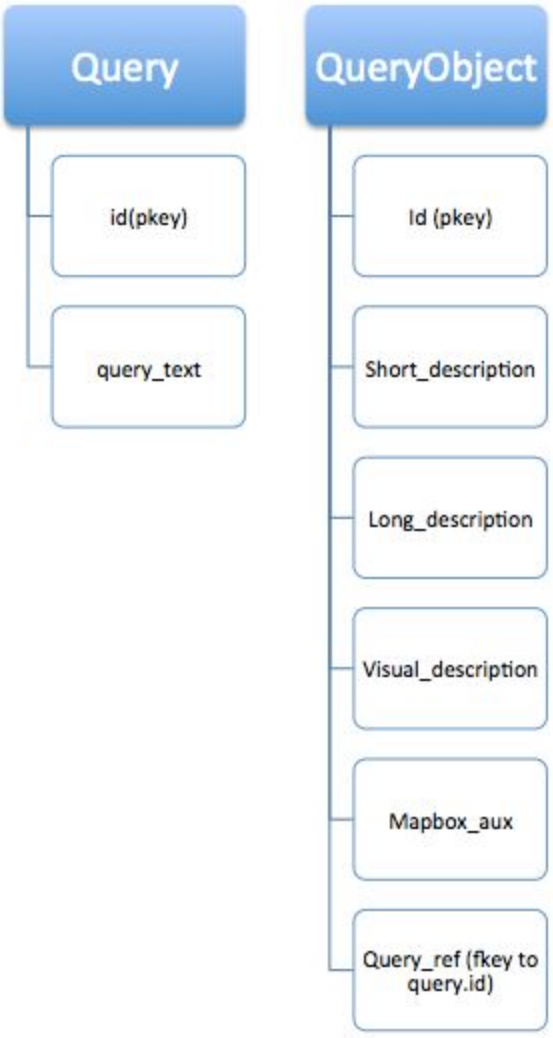
11.2.1. Primary Database

- Datasets included in initial implementation will include the following.
    - College Scorecard: https://catalog.data.gov/dataset/college-scorecard
    - Demographic Statistics by ZIP code: https://catalog.data.gov/dataset/demographic-statistics-by-zip-code-acfc9
    - Housing Affordability Data System (HADS): https://catalog.data.gov/dataset/housing-affordability-data-system-hads

- Star Schema with a focus around location.
    - This will involve having a main table which has all the primary keys for the above datasets, in addition to including information about location.
    - The purpose of using a star schema here is that joins across the datasets will be far more performant


College ScoreCard
location serializer
Join table containing all Foreign Keys
Demographic Statistics
Housing Affordability Data System

11.2.2. API Database



11.3. User's Manual

The user interface is simple and intuitive enough to be understood without any instructions.

11.4. Programmer's Manual

Each of the services GeoCivics comprises is available as a git repository on Github. Instructions for installation of each service can be found within the README file of each project.

Eventually, the team will use Docker containers to run the necessary services together on the same server. This will enable the team, and anyone else, to setup each of the application components with minimal friction.

11.4.1. Code Style Guide

GeoCivics follows most of the PEP8 conventions for Python. Most importantly:

- using 4 spaces for indentation - no tabs
- docstring comments under functions