REST API TO ACCESS AND MANAGE GEOSPATIAL PIPELINE INTEGRITY DATA

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Alexandra Francis

June 2015

COMMITTEE MEMBERSHIP

TITLE:	REST API to Access and Manage Geospatial Pipeline Integrity Data
AUTHOR:	Alexandra Francis
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Professor Alexander Dekhtyar, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Associate Professor Chris Lupo, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Gene Fisher, Ph.D. Department of Computer Science

ABSTRACT

REST API to Access and Manage Geospatial Pipeline Integrity Data

Alexandra Francis

Today's economy and infrastructure is dependent on raw natural resources, like crude oil and natural gases, that are optimally transported through a network of hundreds of thousands of miles of pipelines throughout America[28]. A damaged pipe can negatively affect thousands of homes and businesses so it is vital that they are monitored and quickly repaired[1]. Ideally, pipeline operators are able to detect damages before they occur, but ensuring the integrity of the vast amount of pipes is unrealistic and would take an impractical amount of time and manpower[1].

Natural disasters, like earthquakes, as well as construction are just two of the events that could potentially threaten the integrity of pipelines. Due to the diverse collection of data sources, the necessary geospatial data is scattered across different physical locations, stored in different formats, and owned by different organizations. Pipeline companies do not have the resources to manually gather all input factors to make a meaningful analysis of the land surrounding a pipe.

Our solution to this problem involves creating a single, centralized system that can be queried to get all necessary geospatial data and related information in a standardized and desirable format. The service simplifies client-side computation time by allowing our system to find, ingest, parse, and store the data from potentially hundreds of repositories in varying formats. An online web service fulfills all of the requirements and allows for easy remote access to do critical analysis of the data through computer based decision support systems (DSS).

Our system, REST API for Pipeline Integrity Data (RAPID), is a multi-tenant REST API that utilizes HTTP protocol to provide a online and intuitive set of functions for DSS. RAPID's API allows DSS to access and manage data stored in a geospatial database with a supported Django web framework. Full documentation of the design and implementation of RAPID's API are detailed

in this thesis document, supplemented with some background and validation of the completed system.

ACKNOWLEDGMENTS

Thanks to:

- Austin Wylie for making every meeting entertaining, keeping my TV show and movie queue full, and being more understanding than anyone I know. You are a saint for putting up with a partner that leads an extremely unpredictable and broken life.

- Chris Lupo for playing thesis matchmaker and always sacrificing yourself in the face of angry customers. Thanks for teaching me the basics in 101 and continuing to inspire me throughout my college career.

- Alex Dekhtyar for your time, guidance, and motivation throughout two years of thesis creation.

- Ignatios Vakalis for memories than I can list here. Dr. V, you were not only the best and funniest department chair anyone could ask for but also a great friend. Your vibrant and exotic shirts always make my day. Your jokes sometimes made me want to punch you but always gave me a good laugh. Our frequent meetings gave me guidance even though lasted five times longer than expected due to all of the interesting gossip and debates we had. I will continue to work with and be a part of the CSC Department as long as you have a Greek accent.

- Leslie Francis for telling me to stop complaining and deal with it. Mom is always right.

- Steve Francis for having the perfect words in any situation or dilemma. I am lucky to have the smartest man in the world as my father.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Natural resources, like crude oil and natural gases, are the raw material for energy that the world consumes. Transporting these resources from where they are found to where they are processed or refined, and then again to where they are eventually consumed should be done as safely, efficiently, and economically as possible. There are many methods used to transport resources, but pipelines remain the optimal way. America depends on a network of more than 185,000 miles of liquid petroleum pipelines, nearly 320,000 miles of gas transmission pipelines, and more than 2 million miles of gas distribution pipelines to move energy and raw materials to fuel our nation's economic engine[31].

All homes and businesses receive their power through liquid petroleum and natural gas pipelines that run above and below the ground. The pipelines sometimes get damaged and need to be repaired so that homes and businesses aren't left without power for long. In ideal situations, pipeline operators are able to detect potential damage before it occurs. Damaged pipes that go unnoticed have caused power losses and natural gas leaks, negatively affecting thousands of people and homes[2]; but, monitoring hundreds of thousands of miles of pipelines on a continent is difficult. Natural disasters like earthquakes, as well as climate change, can have effects on these pipelines. Detecting such events and then inspecting all potentially affected pipelines would take an impractical amount of time and manpower[2].

Aside from natural disasters, another significant challenge with pipeline integrity is pipeline encroachment. An encroachment is a building or structure that is not compatible with right-of-ways. A right-of-way is the strip of land that house a pipeline. Detecting encroachment is another task that is difficult to do at a large scale without satellite imagery and computing power[9].

When pipeline operation companies, like Pacific Gas and Electric Company and others, are looking to expand their pipeline network, they need to find the best pieces of land, or right-of-ways, to install them. The companies need to take into account various characteristics of the land including soil type, precipitation, seismic activity, buildings, structures, tree growth, temperature, and many more natural factors. In order to gather all of this information and analyze its importance, satellite imagery and ground movement sensors, as well as other weather and land data, have proven useful.

1.1 Problem Definition

Pipeline integrity depends on a very diverse collection of data sources. Each of these might be found in different physical locations, stored in different data formats, and owned by different organizations. In order to make a meaningful analysis of the quality for a plot of land to be used as a right-of-way, companies have to search through many databases and figure out what format they have to ingest. This is too time consuming and therefore restricts the number of input factors they can analyze.

1.2 Our Contribution

The solution to this problem involves creating a single, centralized interface that the pipeline operators can query to get all the information they need. Thus, a standardized service is needed to request specific data as well as input data from various sources. This service would simplify client-side computation time by allowing our system to find, ingest, parse, and store data from

potentially hundreds of repositories in varying formats.

The service needs to be efficient, consistent, maintainable, and reliable. With so many hundreds of thousands of miles of right-of-ways on top of all of the layers of geospatial data, the data model needs to be scalable. If the system fails, it could mean a loss of power for thousands of people or a potentially hazardous spillage of a non-renewable resource. For this reason, this system needs to be dependable and robust.

To fulfill all of these requirements and meet the needs of pipeline operators, the system is online. A web service for easy remote access is critical so that computer-based discision support systems (DSS) can then analyze the data and allocate resources to remedy any potential hazards.

1.2.1 REST API

REST, or REpresentational State Transfer, is a simple, stateless architecture that usually runs over HTTP. A RESTful API (Application Programming Interface, a digital library of functions) relies on a client-server, cacheable communications protocol[15]. Providing a well designed and intuitive REST API is one of the key pieces of addressing the problem of maintaining consistently functioning pipelines. Because REST is a web service, this API is the connection between the user-facing interface and the database where all of the data is stored.

By utilizing the REST API, pipeline operators can query for specific geospatial data layers for any given region. This system is called REST API for Pipeline Integrity Data, or RAPID. A sample query that an operator might request would be for "a 20 mile by 40 mile square of San Luis Obispo County, return all new seismic data since last Monday at midnight as a JSON[1] object." The original data may have been GeoJSON[2], Shapefiles[3], or another format

[1]JSON: JavaScript Object Notation

[2]GeoJSON is a format for encoding a variety of geographic data structures. A GeoJSON object may represent a geometry, a feature, or a collection of features[6].

[3]A Shapefile is a digital vector (non-topological) storage format for storing geometric location and associated attribute information[32].

entirely–but RAPID will have previously ingested and parsed the original file to store into the database. Finally, RAPID retrieves the data and rebuilds the standardized JSON that the user has requested.

This thesis aims to provide an optimal and complete REST API design and implementation documentation. A detailed description of the full implementation of RAPID's API can be found in Chapter 4.

1.2.2 Data Model

In order to create a system that aggregates varying types and formats of data, we created our own geospatial data model to store the core vital data that has been stripped from original files. Although there are a large number of formats in which geospatial data might be stored, the data content all boils down to the same core information: a location and some metadata.

The location is stored as a list of latitude and longitude pairs that represent the polygon for the region, or it could just be a single point. For example, bounding box is used for precipitation in a region, whereas a single point is used to represent the location of an earthquake's epicenter.

The metadata can be stored as a dictionary or key-value store of unique properties that contain relevant information to the feature. Using seismic data from an earthquake again as our example, the properties contain information about the magnitude of the earthquake and potentially aftershock times with their corresponding magnitudes.

To define the relevant information about a geospatial feature, we use the term "geometry" to define a region's location (the list of longitude and latitude coordinates from above) and "properties". Currently, the geometry and properties pertaining to a feature can be formatted in many different ways depending on the database or location that they are stored. RAPID is designed to accept and remain functional with two different existing popular geospatial data formats: GeoJSON and Shapefiles[4].

[4]Shapefiles will be defined in greater detail later. They are .shp files that are commonly

To do this data ingestion most efficiently, RAPID is built upon a PostgreSQL database with the PostGIS extension that provides built-in flexibility for any sized or shaped geometry (polygon representation of a region) with a list of any client system defined properties (the metadata). At a high level, PostGIS adds support for geographic objects and offers location SQL queries. It stores the locations as vectors and rasters connected to some metadata. Because the RAPID project is a joint work with Austin Wylie, a more in-depth description of the data model design and implementation choices can be found in Wylie's M.S. Thesis[42]. We have included some details regarding the necessary information regarding the database details in this thesis document in order to maintain a cohesive narrative.

This thesis concentrates on the online intermediate layer between the customer's software system and how the data is stored, as the REST API. This multi-tenant service layer is built in the form of a lightweight server that gives access to all of the secure request services. The document is organized as follows. Chapter 2 provides a background of APIs and a progression of how API design, creation, and implementation has developed over their short lifetime. Chapter 3 describes the design of RAPID's RESTful API, providing motivation and support for the choices that were implemented which is detailed in Chapter 4. Chapter 5 gives validation and verification of the quality of the completed API through a third party system that uses RAPID successfully.

used in ArcGIS for geospatial data[32].

Chapter 2

Background and Related Work

Most software applications and projects need to reuse components that have been previously implemented and they are able to do this through APIs that expose these components. Large and complex APIs are becoming an integral part of most current software development technologies[34]. Understanding where technologies come from and how they developed is critical in order to better understand where they are now and where the could potentially go in the future.

After discussing previous successful and unsuccessful API's in recent past, this chapter will cover proper API design and briefly cover a standard that RAPID is required to comply with before commercial use.

2.1 Early API Development

In order to understand how to best design an API for a large software system, it will be useful to discuss the history and origins of APIs. Martin Bartlett accurately states:

> "The principle of a well documented set of publicly addressable 'entry points' that allow an application to interact with another system has been an essential part of software development since the

> earliest days of utility data processing. However, the advent of distributed systems, and then the web itself, has seen the importance and utility of these same basic concepts increase dramatically"[23].

The first "dot-com" boom introduced a need for innovative ways to syndicate products across e-commerce web sites from various platforms. A new layer on top of existing HTTP infrastructure, such as a web API, proved to be the right tool. A few innovative tech engineers began defining and creating the first APIs for sales and commerce management. This early API development was just the beginning of a decade long evolution that we can now call the early history of web APIs[23].

There have been a large number of companies and other organizations that have created APIs and libraries for developers over the past 15 years. One of the first most popular APIs introduced was Salesforce, but numerous other companies like Twitter, Amazon, Google, FourSquare, Flickr, and Instagram have announced APIs for their organizations since then. The first few big ones are detailed in the following sections.

2.1.1 Salesforce

On February 7th, 2000, SalesForce officially launched its enterprise-class, web-based, sales force automation as an "Internet as a service" which used an XML API at its core. As a Software-as-a-Service (SaaS) company, they identified that customers needed to share data across their different business applications, and APIs were the way to do this. Salesforce was the first cloud provider to take an enterprise class web application and API. Today, they are still a leading power in creating, testing, and deploying real-time APIs[23].

2.1.2 eBay

On November 20, 2000, eBay launched the eBay Application Program Interface, along with the eBay Developers Program. Their API aimed to standardize the way in which various applications built and improved upon the

existing eBay application. Prior to the release of the eBay API, there were a number of applications that relied on eBay either legitimately or illegitimately; so, the goal of this API was to make it easier for partners and developers to build a business around the eBay ecosystem in a secure and efficient way[23].

2.1.3 Facebook

On August 15th, 2006, Facebook launched its long-awaited development platform and API. The API used REST with XML responses. Almost immediately, developers began to build social applications, games, and other tools with the new development library[23].

2.1.4 GoogleMaps API

On June 29th, 2006, Google launched Google Maps API allowing developers to put Google Maps on their own sites using JavaScript. When Google Maps was released six months earlier, it was immediately so popular that developers hacked the JavaScript interface and developed applications that hacked Google Maps and there were even some 'How-To' books written specifically for this application. For this reason, Google Maps was forced to release an API so that developers could utilize their local map services without the need to hack[23].

2.2 Modern APIs

Although the "history" discussed in the previous section only covers the span of about fifteen years, things are changing so quickly in the world of software that it makes sense to call this the "early history" of APIs. The motivation for modern APIs is largely the same and API implementation has also stayed pretty consistent. Developers were able to learn from a few poor APIs so as not to mimic their mistakes[23]. One example that is now infamous

is Flickr's first API that was released in 2004[23].

Today, external pressure to produce an API has been a major force behind many of today's API developments. Another large driving force for continuous API development is public demanding information from its government[7]. APIs provide an interface for previously implemented functionality that other developers have access to in order to perform various tasks. Robillard defines them to be a tool for "support code reuse, provide high-level abstractions that facilitate programming tasks, and help unify the programming experience"[34].

As mentioned earlier, one of the purposes of APIs is to allow for easier development by outside parties. However, some of the APIs for large organizations and code bases have gotten very complex and difficult to understand. They would be useless if the difficulty of using APIs nullified the productivity gains they offer[34]. Proper API design is key to allow for efficient use and implementation of integrated code.

2.3 API Design

An API that is difficult to understand and use leads to decreased programmer productivity which contradicts the purpose of APIs to begin with – to increase efficiency. In February and March of 2009, Martin P. Robillard did a survey of software developers to find out what obstacles were faced when learning to use APIs in order to better understand the best design practices when creating the API. Out of the thousands that the survey was distributed to, he received 83 responses that provided a representative sample of the population[34]. The study revealed three key points:

1. Information about the high-level design of the API is necessary to help developers:
    - choose among alternative ways to use the API,
    - structure their code accordingly, and
    - use the API as efficiently as possible.

2. Code examples can become more of a hindrance than a benefit when there's a mismatch between the tacit purpose of the example and the goal of the example user.

3. Some design decisions can influence the behavior of the API in subtle ways that confuse developers.

Robert DeLine performed a very similar study of 440 software engineering professionals involving a combination of surveys and in-person interviews. This study revealed that "the most severe obstacles faced by developers learning new APIs pertained to the documentation and other learning resources"[12]. Due to the fact that the majority of the results from the survey showed an issue with documentation, the analysis of this study provides five crucial factors to consider when documenting the API:

1. Documentation of intent

2. Code examples

3. Matching APIs with scenarios

4. Penetrability of the API

5. Format and presentation

In order to reach optimal API design, these factors can be interpreted to prioritize API documentation development efforts. In general, both studies came to the same basic conclusion: documentation can make or break an API. The actual design and implementation of the API is important but what is lacking in most modern APIs is explanations of how it should be used and when it should be used in specific ways. The perfect API design is nothing without complete documentation.

2.4 OGC Compliance Standard

The Open Geospatial Consortium (OGC) is an organization that aims to provide international standards for geospatial and mainstream location-based web services in a accessible and useful manner. It is an industry consortium currently made up of 511 companies, government agencies and universities that are working to develop publicly available interface standards[40]. The OGC Compliance Standard is a set of technical documents that provides an interface and set of encoding standards for geospatial data management and web services[40]. This interface specifies a set of possible operations and queries to access geospatial data independent of how the data is stored underneath. The operations that RAPID uses, discovery operations and query operations, are only a subset of the whole standard. The OGC standard states:

> "Discovery operations allow the service to be interrogated to determine its capabilities and to retrieve the application schema that defines the feature types that the service offers. Query operations allow features or values of feature properties to be retrieved from the underlying data store based upon constraints, defined by the client, on feature properties[40]."

There are other operations that are not in the scope of RAPID: locking operations, transaction operations and operations to manage stored parametrized query expressions. However, for a system to fit the official standard, it has to comply with only one of the "conformance classes." A Web Feature Service (WFS) is a modification in the way geographic information is created, modified and exchanged on the Internet. WFS offers direct fine-grained access to geographic information at the feature level and feature property level as opposed to File Transfer Protocol (FTP), which shares geographic information at the file level. A WFS allows clients to "only retrieve or modify the data they are seeking, rather than retrieving a file that contains the data they are seeking and possibly much more"[40].

2.4.1 Conformance Classes

The International Standard provides a list of the possible classes along with the behavior or operations that need to be satisfied in order to meet the requirements of that class. The possible classes are Simple WFS, Basic WFS, Transactional WFS, Locking WFS, HTTP GET, HTTP POST, SOAP[1], Inheritance, Remote resolve, Response paging, Standard joins, Spatial joins, Temporal joins, Feature versions, and Manage stored queries[40].

At a minimum, all implementations must provide Simple WFS class. All other classes are optional (and build off of Simple WFS). The operations required for Simple WFS are `GetCapabilities`, `DescribeFeatureType`, `ListStoredQueries`, `DescribeStoredQueries`, and `GetFeature` operation with only the `StoredQuery` action. The international standard also states:

> One stored query that fetches a feature using its ID shall be available but the server, may also offer additional stored queries. Additionally, the server shall conform to at least one of the HTTP GET, HTTP POST or SOAP conformance classes[40].

For RAPID, we have chosen HTTP[2] GET conformance class as the complementary conformance class. This class states, "the server shall implement the Key-value pair[3] encoding for the operations that the server offers"[40].

After much discussion with customers, we reevaluated the scope of RAPID as a M.S. Thesis. We decided to design RAPID with the standard in mind but without full standard compliance. By continuing with the existing RAPID model and API design, we were able to eliminate many of the tedious compliant modification tasks and continue with our optimal design for the requirements that we originally understood from the customers. The model and interactions were designed for fastest and most robust data ingestion and retrieval. Although the current implementation does not comply with the OGC Standard, our work has designed and set up RAPID so that it will be seamless for future work to tweak the existing system into an OGC Compliant WFS.

[1]SOAP: Simple Object Access Protocol
[2]HTTP: Hypertext Transfer Protocol
[3]KVP: Keyword-value pairs

Chapter 3

Design

Before beginning design of the REST API for RAPID, we had to understand all of the requirements that it needed to fulfill. Once back-end model and customer-facing API requirements were established, we could begin concurrent design of both pieces as well as the intermediate service layer that would connect them. The design went through dozens of iterations throughout the whole process as we continued to better understand the customer's desires, the capabilities of the tools, and the time constraint we were facing.

3.1 Requirements

There are three main components in RAPID: the front-facing client visualization[1], the back-end database, and the REST API that allows the two to communicate. While the database itself is not be directly accessible by customers outside this system, its data is exposed to developers through a well-defined REST API. The requirements for the whole system fall into two main categories:

- REQ-1: RAPID shall be able to ingest and integrate meaningful geospa-

[1]The RAPID UI is implemented by a third-party for validation, detailed in Chapter 5 and is not a main component of RAPID.

tial data from various data sources in a useful and user-friendly way.

- REQ-2: RAPID shall be able to retrieve desired data for a given region provided that the specified data has been previously ingested.

The pieces of the system that Cal Poly is responsible for designing and implementing are twofold: the data model and the REST API. Each piece has specific requirements that need to be measured to confirm validity and completeness through a third-party application that uses the completed system.

3.1.1 Model Requirements

We discuss some of the pieces of the data model that directly relate to the REST API and drove its design. All geospatial data has a location which we need to store in the database. This can be labeled as a region of geographical land like California. Full documentation of the model requirements, design, and implementation can be found in Wylie's thesis[42].

Scalability

The solution shall be scalable so that, in theory, it can one day be applied to the entire natural gas and petroleum liquids energy pipeline network in North America. The database developed shall be sufficiently robust. The data model and system architecture shall allow for scaling the system without significant changes to the core data model and the core system architecture. The system should remain efficient even with large data sets and large regions. Scalibility refers to maintaining storage space and time efficiency. The validation of these requirements are documented and discussed in Wylie's thesis[42].

- REQ-3: RAPID shall scale well with large amounts of data.
- REQ-4: RAPID shall remain robust no matter the region size.
- REQ-5: RAPID shall remain robust no matter how many regions need to be stored.

- REQ-6: RAPID shall be multi-tenant[2].

Single Database

The goal of the integrated database is to gather and sanitize the wide range of data about the conditions in the immediate vicinity of operating pipelines that could affect the integrity of the pipeline and store it in one place. Currently, the desired pipeline integrity data resides in various databases or physical file storage in different locations. Ideally, customers would not need to search through multiple databases to retrieve the data they desire. RAPID shall contain all ingested data in a single database. The data originates in different database models but is copied and converted into this data model to be stored indefinitely[3].

If more storage is required by the database, all of the data is moved to the new database, and remain together and in the single modeled format. The data should be well modeled—it should be able to gracefully handle all the content that it needs: client owners, regions, layers, categories, and data entries.

- REQ-7: RAPID shall have a single database storage model for all data.
- REQ-8: RAPID shall gracefully handle data that originates in different forms and store it properly in the modeled database.
- REQ-9: The data model shall be complete in containing and maintaining all necessary content for pipeline integrity geospatial data.

[2]Multi-tenancy is an architecture in which a single software application serves multiple customers or other software applications with their own secure virtual environment[10].

[3]Indefinitely, for all intents and purposes, means until RAPID is no longer used.

3.1.2 API Requirements

Speed and Cost

It is intended that accessing this data is reasonably fast and has a low cost. Provided syntactically correct data, RAPID shall ingest, parse, and store the data in a timely manner. Data retrieval should also happen quickly. To ingest or retrieve a single region or other data entry, we define the system to be "fast" if it takes no longer than .001 seconds for the transaction. This ratio shall remain the same as number of data entries increases.

- REQ-10: Data insertion through the API shall be fast.
- REQ-11: Data insertion through the API shall be low cost.
- REQ-12: Data retrieval through the API shall be fast.
- REQ-13: Data retrieval through the API shall be low cost.

Correctness

The data that is returned must be reliable in its correctness/integrity. Given a specific request, the data returned shall be complete, meaning all of the data matching the request specifications is retrieved. If the data exists in the database that matches the details of the request, it should be retrieved and no more. The data stored and returned should be an accurate reflection of the data that is inserted.

- REQ-14: The data that RAPID returns in response to requests shall be reliable, accurate, and correct.
- REQ-15: The data that RAPID returns in response to requests shall be complete.

Ease of Use

The documentation of the system shall be complete and user-friendly. Given an understanding of the documentation, hooking up a client system to the database should be intuitive. The API should provide ease of implementation in connecting an existing system to begin inserting and retrieving data. The calls shall be clear in their responsibilities, what information or parameters they need, and the content and format that is returned.

- REQ-16: The RAPID API documentation shall be complete and understandable by a pipeline operator.
- REQ-17: The RAPID API shall be intuitive and user-friendly, given an understanding of the documentation.

Standard Format

Ingesting the data shall be compatible in form for integration into existing tools used by the pipeline industry for managing data. No matter what the format or file type of data prior to ingestion, the data shall be in the single desired file type or data format desired by the client. The data can be pulled from various data sources and this should not affect the performance or reliability of the system.

- REQ-18: The RAPID API shall ingest data that originates in the formats agreed upon (GeoJSON and Shapefile).
- REQ-19: The RAPID API shall return data in a standard format that the client requests.

3.2 Data Model

The Data Model is designed to meet the requirements described in Chapter 3.1.1 and provide a way for the API requirements in Chapter 3.2.2 to be met.

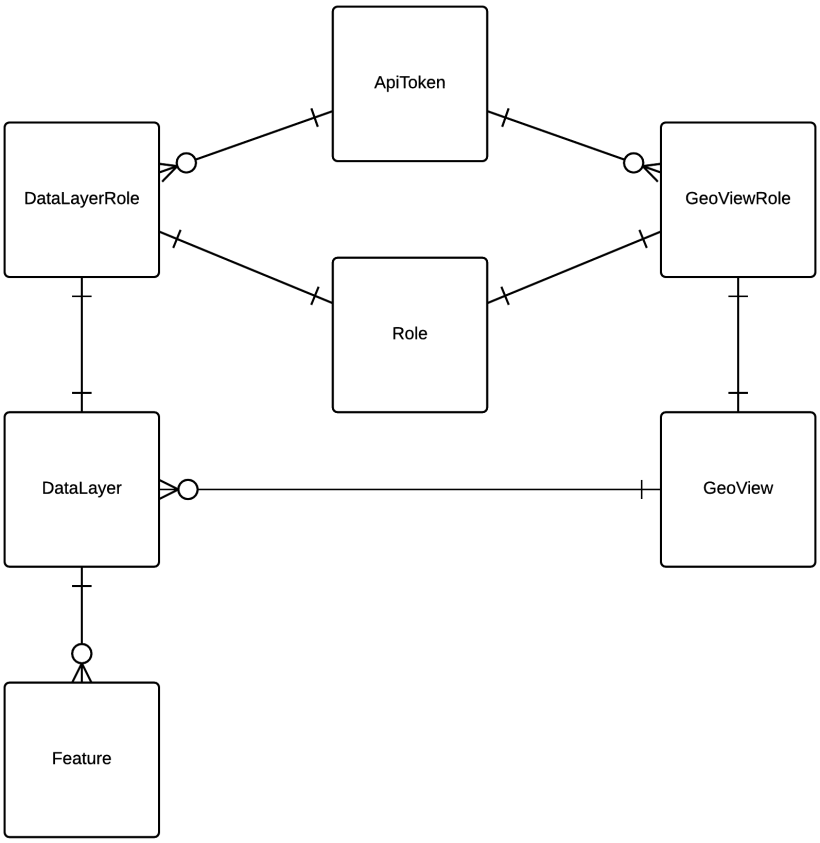

Figure 3.1: Entity Relationship Diagram modeling the database design at a high level. More detail can be found in Wylie's Thesis[42].

This thesis documents the minimum amount of the Data Model design to understand the API design. Figure 3.1 provides a high level understanding of the model as an entity-relationship diagram. Further reading can be found in Wylie's thesis[42]. The database is modeled to store all of the necessary geospatial and administration objects. It is modeled with tables to support each different object. All of the tables have different attributes. Each attribute represents a column in the table, with an additional column for id, auto-generated by the table. The columns each have a type, documented in the format '*name: Type*', for the tables listed in Chapter 3.2.1 - 3.2.6. RAPID data model supports the following column types.

- *TextField: A large text field. The default form widget for this field is a Textarea.*
- *ForeignKey: A many-to-one relationship. Requires a positional argument: the class to which the model is related.*
- *TimeKey: A time, represented in Python by a datetime.time instance. Accepts the same auto-population options as DateField.*
- *GeometryField: The default spatial reference system for geometry fields where the field coordinates are defined in longitude, latitude pairs in units of degrees.*
- *PolygonField: The spacial reference system for polygon fields where the coordinates of the field are defined in longitude, latitude pairs. For a rectangle, there are 5 pairs of coordinates, the first and last are the same. E.G. 'POLYGON((0 0, 0 5, 3 5, 3 0, 0 0))'*

[18]

3.2.1 ApiUser

An ApiUser represents a client owner that inserts and retrieves data. It is used to keep track of who owns what data and maintain permissions when retrieving data. The table to store API Users has the following attributes.

- *uid: TextField* - unique user id
- *email: TextField*
- *password: TextField*

3.2.2 GeoView

GeoViews, which represent a geospatial area of land that can be described by a polygon, are stored in the GeoView table with the following attributes.

- *uid: TextField* - unique GeoView id
- *descriptor: TextField* - user defined name to identify this region
- *geom: GeometryField* - null means there are no boundaries
- *bbox: PolygonField* - a rectangle that encompasses and completely contains the geometry, used for preliminary computations
- *properties: TextField* - optional details
- *layers: ManyToManyField(DataLayer)* - list of layers that belong to this region

3.2.3 DataLayer

The Data Layer table, which represents the collection of similar data entries for a region, has the following attributes.

- *uid: TextField* - unique layer id
- *descriptor: TextField* - user defined name to identify this region
- *bbox: PolygonField* - a rectangle that encompasses and completely contains the geometry, used for preliminary computations
- *is_public: BooleanField*
- *properties: TextField* - optional details

3.2.4 DataSource

The Web Data Source table, which represents the URL endpoint for a type of geospatial data, has the following attributes.

- *url: TextField* - the URL endpoint where this data can be found
- *layer: ForeignKey(DataLayer)* - the layer that this data will belong to
- *update_interval: TimeField*
- *properties: TextField* - optional details

3.2.5 Archive

The Data Source File table, which represents the data source files for a type of geospatial data, has the following attributes.

- *uid: TextField* - unique layer id
- *content: TextField* - the full file content
- *layer: ForeignKey(DataLayer)* - the layer that this data will belong to
- *internet_media_type: TextField* - tye type of the file, eg. '`application/json`', '`application/vnd.geo+json`', '`application/zip`'
- *create_timestamp: TimeField*
- *properties: TextField* - optional details

3.2.6 Feature

Feature is a single data point that pertains to a layer, and therefore category and owner. The data has the following columns.

- *uid: TextField* - unique feature id

- *archive: ForeignKey(Archive)* - the full contents of the file that was originally ingested
- *geom: GeometryField* - null means there are no boundaries
- *bbox: PolygonField* - a rectangle that encompasses and completely contains the geometry, used for preliminary computations
- *properties: TextField* - optional details
- *layer: ForeignKey(DataLayer)* - the layer that this data will belong to
- *create_timestamp: TimeField*
- *modified_timestamp: TimeField*

3.2.7 Other

The data model has other tables for model management. They are described in more detail in Wylie's thesis document. They are listed below.

- ApiToken
  - *token: TextField*
  - *token_owner: ForeignKey(ApiUser)*
  - *token_user: ForeignKey(ApiUser)*
  - *issued: TimeField*
  - *expires: TimeField*
- Role
  - *name: TextField*
- GeoViewRole
  - *token: ForeignKey(ApiToken)*
  - *role: ForeignKey(Role)*

  - *geo_view: ForeignKey(GeoView)*

- DataLayerRole
  - *token: ForeignKey(ApiToken)*
  - *role: ForeignKey(Role)*
  - *layer: ForeignKey(DataLayer)*

- ApiCall
  - *call: TextField*
  - *timestamp: TimeField*
  - *ip: TextField*
  - *token: ForeignKey(ApiToken)*

3.3 REST API

The RAPID REST API was designed around the Data Model to allow pipeline operators to easily insert, update, remove, and retrieve data. A more detailed description of what it means to be RESTful and what a REST API is can be found in Chapter 1.2.1. Our API calls are RESTful HTTP calls made to a specific endpoint. The general pattern is that collections are accessed via a URL ending in a noun such as `/geoview`, and to target an individual item, append its uid (`/geoview/432dd1k2`). To tell the API whether you are retrieving, inserting, or deleting the item, users should make use of the HTTP verbs, GET, POST, and DELETE (respectively). To update an item, we have simulated a PUT request by retrieving the existing data item, creating a new one with the updated data content, posting the new item, and removing the old one. The API responds with JSON reflecting the changes.

The design of the API was driven by the RAPID database model. In order to fill the database, there needs to be a way to insert records into each of the customer-modifiable tables (described in Chapter 4.2): GeoView, DataLayer,

Feature. Each of these has an HTTP POST method at a specified URL endpoint that creates an entry in the database in the respective table. In order to update or remove an entry, there is also be a HTTP DELETE method to either permanently remove or replace with an updated entry. For data retrieval, each of the customer-modifiable tables have an HTTP GET method at a specified URL endpoint that returns the data. After many iterations of the data model and API design, the finalized API is documented in Sections 3.3.1 to 3.3.10.

3.3.1 GeoView

A fully documented list of operations and endpoints can be found in Chapter 4, Implementation.

Create GeoView

Initially, GeoView of interest needs to be defined for the system so that layers of data entries can be inserted and queried for. A GeoView is simply a two-dimensional area of land. For our system, because of RAPID's data model and the way the data needs to be stored, a GeoView can be described by a series of latitude and longitude (lat-long) pairs. When defining a GeoView, a minimum of three different lat-long pairs of points (forming a triangle) need to be specified. The more points, the more specifically outlined the GeoView. The points need to form a valid polygon[4].

For example, the geometry for Palo Alto County is a simple quadrilateral with four unique coordinates: (-94.913890, 42.909700), (-94.913723, 43.255054), (-94.443137, 43.255014), (-94.442954, 42.908073). A sample San Luis Obispo County geometry has 132 coordinates to define the GeoView, a polygon with 132 sides[8]. A GeoView can have any number of coordinates greater than three to define its geometry up to potentially hundreds. The other piece of information needed when creating a GeoView is the descriptor. The customer picks this identifier for each GeoView that they create. These two fields, ge-

[4]A polygon is a plane closed figure with at least three straight sides and angles[6].

ometry and descriptor, are passed in the URL parameter when the POST call is made. The rest of the attributes in the GeoView model are automatically generated by RAPID.

Retrieve GeoView

The customer can get a list of all GeoView or a specific GeoView that they own or have access to with a HTTP GET method. If a "uid" is not specified, all GeoViews that the customer has access to is returned. Otherwise, if a single "uid" is entered in the URL, that specific GeoView is returned.

3.3.2 DataLayer

Create DataLayer

Similarly to GeoView creation, in order to insert a new Layer entry in the database, an HTTP POST call with proper information and formatting of a URL parameter can be made. The fields to create a DataLayer follow the model attributes: descriptor, is_public, and properties. The descriptor, again, is a customer-defined identifier for this layer. The rest of the attributes in the DataLayer model are automatically generated by RAPID.

Retrieve DataLayer

To retrieve a list of all layers that a customer owns, an HTTP GET call can be made to the endpoint. This call returns a list of the layers.

3.3.3 Feature

Create Feature

Features are the bulk of REST requests that will be made. Entering data points in the Features table of the model can be made with an HTTP GET request to the correct endpoint and the proper URL parameters. The data that is sent must in the parameter must contain the layer for which this geospatial feature belongs, the full file contents that this feature came from, and the internet media type of that file. The full file of the original data that this feature came from must contain the a geometry embedded somewhere to extract and store in the model. On success, this will return the features that were inserted.

Retrieve Features

To retrieve a list of features, the GeoView and layer need to be specified. These two pieces of information can be passed in the URL. The GeoView can be specified by the "uid" of the GeoView and the layer can be specified by the descriptor of the desired layer. This request will return a collection of all features that fall under the request specifications.

3.4 Discussion

The initial API design was very different. It was originally structured with Region retrieval being a separate call from Region metadata retrieval. The geometry was not a polygon but only had two coordinates, the corners of a rectangle. We chose to support a more detailed region perimeter instead of simply a rectangle. The uid did not exist in early iterations. It was only the automatically generated unique database id.

We introduced another id, the uid, so that if the data is ever moved due to scaling or other reasons, the uid can remain the same, even if the id might change in the new database. Ownership was handled through an access mode

field in the region that was one of *ownerOnly, registeredUsers* or *all* and is now done through a owner foreign key to an API user. Each region had a field for time stamp of when this region was inserted into the database, but that became obviously unnecessary for a region as time went on.

In the original design, the complexity and detail of the data was moved from the data entry level to the layers. A layer had many more fields. A layer belonged to a region and had a region identifier field to describe this relationship. There was also a type field that was used to describe the structure of the shape of the layer which was one of *continuous, polygon, line* or *point*. Layers had a access mode field (similar to region access mode), a static mode field (either *static* or *dynamic*), an origin field, and a time stamp for when the layer was created.

Data entries were called features originally and existed in FeatureCollections. They were very simple with only the layer identifier that they belong to, the geometry, and a time stamp for when the data was inserted, potentially a list of start and stop times for when this features was active. Data entries seemed more useful at a higher level for customer use.

Many fields have been moved, removed, or added. The specific URL endpoints have changed many times, as well. During some design iterations, the input parameters were part of the actual URL and in other iterations, they were simply URL parameters, and in the final design, they are a combination of both. These changes developed as we became more familiar with uses for one versus the other and realizing what fits best for our purposes and design.

After completion of our thesis implementation, work will continue improving RAPID and purposing it for customer use. This thesis provides documentation of RAPID as it is at this stage. Future work may invalidate the correctness of implementation and design details discussed in Chapter 3 and 4.

3.4.1 Race Conditions

In software, a race condition is an undesirable circumstance when two or more operations occur at the same time that must only be performed sequentially[36]. These operations might be one of the "CRUD"[5] operations. Reading and writing large amounts of data to storage at almost the same instant may cause some or all of the old data to be overwritten while it is still being read. Depending on the system, these errors are handled differently, whether it is full system failure and a crash or an illegal operation notification and shutdown of the program.

In RAPID race conditions might occur if one client is updating features in a layer while another client is requesting the data. This situation is unlikely due to the minimal number of customers and rare use case of retrieving a feature at the same time as updating it. General use of RAPID is designed for sequential write, read, and update operations. RAPID does not handle race conditions gracefully and depends on the user to prevent them from occurring. Customers are responsible for ensuring no read operations are made to the data while it is being updated.

[5]C.R.U.D. operations are create, read, update, and delete operations of data in a system, RAPID in this case.

Chapter 4

Implementation

We chose to implement the API using a powerful Python-based REST framework called Django with a PostGIS database under the hood. Django makes it easy to build a Web API and has extensive support for geospatial data with a toolkit called GeoDjango that manipulates GeoJSON with Django[17]. All of these tools provided a basis to build RAPID upon with some existing GIS functionality without starting from scratch.

With our framework and design decided upon, implementation next required a set of endpoints that the REST calls could hit in order to get or send the data desired. Table 4.1 defines the list of endpoints that RAPID supports, all using the Pipelions server at `http://pipelions.com/`. The table has the extension of the URL that would follow the Pipelion server address. For example, the full REST URL endpoint for `/rapid/geoview` is `http://pipelions.com/rapid/geoview`.

4.1 Create GeoView

The endpoint to create a GeoView (`https://pipelions.com/rapid/geoview/`) as well as retrieve a full list of all accessible GeoViews is the same. A RESTful POST call to the endpoint creates a new GeoView on success. As described

in the design in Chapter 4.3.1, the REST call requires the user to specify the geometry of the region described by the GeoView, the descriptor, and an optional properties field. The descriptor is simply the customer/user defined GeoView identifier. This can be a text name like 'SLO', a numerical id like '21073', or a combination of the two 'CA4019'.

4.1.1 GeoJSON Geometry

The geometry can either be in valid GeoJSON or can be in GEOSGeometry database text format. A valid GeoJSON object requires a "type" field and a "coordinates" field. The type field for our geometries needs to be "Polygon". The coordinates field of a Polygon are an array of LinearRing coordinate arrays. The first element in the array represents the exterior ring. Any subsequent elements represent interior rings (or holes). The coordinate array is an array of two element arrays where the first element has to be the same as the last element to complete the ring. For RAPID, the geometry is generally just be a rectangle, but RAPID does support any form of a valid geometry, which means a valid polygon. An example geometry in valid GeoJSON of a GeoView with no holes is below.

```
{''type'': ''Polygon'',
    ''coordinates'': [
      [[100.0,0.0],[101.0,0.0],[101.0,1.0],[100.0,1.0],[100.0,0.0]]
    ]
}
```

An example geometry in valid geojson of a GeoView *with* holes is the following.

```
{''type'': ''Polygon'',
    ''coordinates'': [
      [[100.0 0.0],[101.0,0.0],[101.0,1.0],[100.0,1.0],[100.0,0.0]]
      [[100.2,0.2],[100.8,0.2],[100.8,0.8],[100.2,0.8],[100.2,0.2]]
    ]
}
```

Another option for specifying a GeoView's geometry is as a valid GEOSGeomery Polygon text format to be directly inserted into the database. Depending on where the data comes from, it might already be in this format, which would make it redundant and wasteful for the system to convert into GeoJSON and then back into GEOSGeomerty text format. This format is simply a text field with the lat-long pairs in parentheses, separated by spaces. An example geometry in valid GEOSGeometry Polygon format is below.

```
'POLYGON((10.0 0.0, 11.0 0.0, 11.0 1.0, 10.0 1.0, 10.0 0.0))'
```

4.1.2 POST to GeoView

A RAPID API call to create a GeoView creates a new GeoView under the current owner id with the provided metadata. It is an HTTP POST method and the end point URL is `https://pipelions.com/rapid/geoview/` with one URL parameter, the data object. The data needs to contain the geometry, descriptor, and property information in JSON form with three attributes, "geom", "des", and "props". Packaged in a JSON object, "geom" is a string containing either the valid GeoJSON or GEOSGeometry, "des" is a string containing the descriptor, and "props" is an object, either empty or with more metadata that the customer desires. An example of a valid data URL parameter is below.

```
{   ``geom'': ``{
        ``type'': ``Polygon'',
        ``coordinates'': [
            [[0.0,0.0],[1.0,0.0],[1.0,1.0],[0.0,1.0],[0.0,0.0]]
        ]
    }'',
    ``des'': ``exReg001'',
    ``props'': {},
}
```

On success of inserting into the RAPID database, the uid of the newly created GeoView is returned.

4.2 Retrieve GeoViews

A RESTful GET call to the endpoint `https://pipelions.com/rapid/geoview/` returns a list of all GeoViews that the customer has permission to access. The list is returned as an array of GeoJSON objects with all visible fields that are stored in the database as attributes. The fields that each GeoView will display are "uid", "descriptor", "geom", "bbox", and "properties". An example of what is returned from a GET call to the GeoView endpoint with a single GeoView object in it is below.

```
[
{   ''uid'': ''8JrbghjzSaDCiitvYbWTyT'',
    ''descriptor'': ''exReg001'',
    ''geom'': ''{
        ''type'': ''Polygon'',
        ''coordinates'': [
            [[0.0,0.0],[1.0,0.0],[1.0,1.0],[0.0,1.0],[0.0,0.0]]
        ]
    }'',
    ''layers'': [
    {
        ''features'': [
            ''zwt4H8oFqxAQZFf9enm2Hb'',
            ''koRhUHuwdZKoGAh8GZgdtA'',
            ''cbWQroiKjuajKW8wZh8JRH'',
            ''APWf2dEbLoFKudaeE3tR6T''
         ],
         "uid": ''QJ6oTbnK6iaxp4TbJewYNi''
     },
```

```
    {
        ''features'': [
            ''nPsDvLNrZoYDXjRiCwhN6n'',
            ''gxyjZtGiMdX7NtdvcQfa9Y'',
            ''pAaGhrSRaftajxrnRJVWWL'',
            ''HfEtjCVCxMdkjoiifKoc7b''
        ],
        ''uid'': ''agRDPtbkZNoGGw3ZSkYmnA''
    }],
    ''bbox'': [[0.0,0.0],[1.0,1.0]],
    ''properties'': {},
}
]
```

On failure, an error message is returned.

4.3 Retrieve Specific GeoView

If the customer knows the uid of the GeoView that they are interested in, they can retrieve the data for that specific GeoView instead of an array of all GeoViews. This can be done by sending a GET request to the endpoint in section 5.2 with the uid appended to the end. For example, if the uid for the GeoView desired is `8JrbghjzSaDCiitvYbWTyT`, then the full endpoint is with a GET call to the URL endpoint `https://pipelions.com/rapid/geoview/8JrbghjzSaDCiitvYbWTyT`. This request returns a single GeoView object on success. This object looks exactly like a single element of the array in section 5.2 above. On failure, an error message is returned.

4.4 Delete GeoView

To completely delete a GeoView, a HTTP DELETE request can be sent to the endpoint in section 5.2 with the uid appended to the end. For example, if the uid for the GeoView that should be deleted from the database is `8JrbghjzSaDCiitvYbWTyT`, then the full endpoint is with a GET call to the URL endpoint `https://pipelions.com/rapid/geoview/ 8JrbghjzSaDCiitvYbWTyT`. This DELETE request returns a the GeoView object that has just been deleted on success. This object looks exactly like a single element of the array in section 5.2 above. On failure, an error message is returned and nothing is removed.

4.5 Create DataLayer

Data layer creation is very similar to GeoView creation. Insertion and retrieval are done through different REST requests to the same endpoint. A POST with proper data as URL parameters to `https://pipelions.com/rapid/layer/` will create a new layer and insert it into the database. The data that needs to be passed with the POST request is a JSON object with the following attributes: descriptor, is public, and properties. The descriptor, again, is the customer defined layer identifier. This can be a text name like 'Earthquakes', a numerical id like '4019'. The public field is a boolean value for whether or not this field will be available to access from any customer. This attribute provides ease in not having to add individual users to layers that have public data and will be accessed by everyone. The `props` field is similar GeoView properties: optional object to add further information or metadata about the layer.

All of these fields will be passed as attributes in a JSON object as 'des', 'public', and 'props'. An example of a JSON object that could be passed as a URL parameter in a POST request to `https://pipelions.com/rapid/layer/` is below.

```
{   ''des'': ''Earthquakes'',
```

```
    ``public'': true,
    ``props'': {},
}
```

On failure, the layer is not created and an error message is be returned.

4.6 Retrieve DataLayers

Retrieval of a list of all data layers that a customer has access to can be done through a GET request to the same endpoint as the endpoint in section 5.4, `https://pipelions.com/rapid/layer/`. The list is in the form of an array of GeoJSON objects with all visible fields that are stored in the database. The attributes that each layer will contain are `uid`, `descriptor`, `bbox`, `is_public`, and `properties`. Below is an example of the response of a GET call to the layer endpoint that is an array with only a single data layer in it.

```
[
{   ``uid'': ``mUy6ckBpUQzWDmiVqQg73Y'',
    ``descriptor'': ``US Counties'',
    ``bbox'': [[0.0,0.0],[1.0,1.0]],
    ``is_public'': false,
    ``properties'': {},
}
]
```

On failure, an error message will be returned.

4.7 Retrieve Features of a Specific Layer

To receive all of the features along with other pieces of data that belong to a single, specific layer, the user must know the uid of that layer. The response contains all of this information if a valid uid is appended to the end of the layer

endpoint. For example, if the uid of the layer for desired features is `mUy6ck`, then the full URL endpoint to query is `https://pipelions.com/rapid/layer/mUy6ck`. The response will come in GeoJSON form with attributes `features`, `descriptor`, `bbox`, `is_public`, `properties`, and `uid`. An example response of a successful GET request to the layer/uid endpoint is below.

```
[
{   ``features'': [``7vRqkL'', ``UoWYZEW'', ``3dv9CDC'']},
    ``descriptor'': ``US Counties'',
    ``bbox'': [[0.0,0.0],[1.0,1.0]],
    ``is_public'': false,
    ``properties'': {},
    ``uid'': ``mUy6ck''
}
]
```

The `features` attribute's value is an array of strings which are existing feature uids of that layer. The descriptor is the user-defined identifier. The `bbox` is a bounding box of the layer. The boolean value of `is_public` is true if this layer can be viewed by any user and false otherwise. The `properties` contain the optional metadata, and the `uid` is the system-generated unique identifier of the layer that was requested (which matches the URL uid). A failed request returns an error message.

4.8 Delete DataLayer

To completely delete a DataLayer, a HTTP DELETE request can be sent to the endpoint in section 5.2 with the uid appended to the end. For example, if the uid for the DataLayer that should be deleted from the database is `mUy6ck`, then the full endpoint is with a GET call to the URL endpoint `https://pipelions.com/rapid/layer/mUy6ck`. This DELETE request returns a the DataLayer object that has just been deleted on success. On failure, an error message is returned and nothing is removed.

4.9 Add Layer to GeoView

All GeoViews contain a collection of data layers and multiple GeoViews can have the same layer in it's list, making it a many-to-many relationship. To add an existing layer to an existing GeoView, both the GeoView's uid and the layer's uid. The endpoint is `https://pipelions.com/rapid/geoview/addlayer/` with a single GeoView uid and layer uid appended to the end, respectively. For example, to add the layer whose uid is `mUy6ck` to a GeoView whose uid is `8JrbghjzS`, the URL endpoint would be `https://pipelions.com/rapid/geoview/addlayer/8JrbghjzS/mUy6ck`. On success, a success message with the two uid's is returned. On failure, a failed message is returned as response.

4.10 Remove Layer from GeoView

To remove a layer from the GeoView, a very similar request as adding a layer to a GeoView is made. To remove the layer whose uid is `mUy6ck` from a GeoView whose uid is `8JrbghjzS`, the URL endpoint would be `https://pipelions.com/rapid/geoview/removelayer/8JrbghjzS/mUy6ck`. On success, a success message with the two uid's is returned. On failure, a failed message is returned as response.

4.11 Insert Feature

Inserting a feature or set of features into a layer is done with a RESTful POST request with a URL parameter with valid information and formatting. The full features URL endpoint is `https://pipelions.com/rapid/feature/`. The parameter is a JSON object with three mandatory fields: `layer`, `content`, and `props`. The layer is the uid of the layer that this feature pertains to. Content needs to have a `geom` attribute that will contain the geometry of the feature(s) to be inserted. The geometry is wrapped in a content field so that

multiple features can be added at once. The properties field is any other metadata to describe the feature. Below is a valid JSON object that can be passed as a parameter of a POST request to the feature endpoint.

```
{   ``layer'': ``mUy6ck'',
    ``content'': {
       ``geom'': {
           ``type'': ``Polygon'',
           ``coordinates'': [
              [[0.0,0.0],[1.0,0.0],[1.0,1.0],[0.0,1.0],[0.0,0.0]]
           ]
       }
    },
    ``props'': {},
}
```

On success, an array of the uid(s) of the inserted feature(s) is returned in a response. On failure, a failure message is returned.

4.12 Retrieve Specific Feature

Retrieving a specific feature's metadata is very similar to all other individual REST retrievals in the above sections. The uid of the desired feature is required and appended to the end of the feature endpoint. For a feature with uid `7vR9quzqcM`, the URL endpoint is `https://pipelions.com/rapid/feature/7vR9quzqcM/`. On success, all of the metadata is returned as a GeoJSON object. The relevant attributes of the response object are `uid`, `geometry`, `modified_timestamp`, `create_timestamp`, `layer_id`, `type`, and `properties`. An example of the response of this GET request is below.

```
{   ``uid'': ``7vR9quzqcM'',
    ``modified_timestamp'': ``15:05:34.500522''
    ``create_timestamp'': ``15:05:34.500494''
    ``geometry'': {
```

```
        ''type'': ''Polygon'',
        ''coordinates'': [
            [
                [
                    -77.855148,
                    37.418363
                ],
                [
                    -77.875486,
                    37.416015
                ],
                [
                    -77.867779,
                    37.394498
                ],
                [
                    -77.855148,
                    37.418363
                ]
            ]
        ]
    },
    ''type'': ''Feature'',
    ''layer'': ''mUy6ck'',
    ''properties'': ''{
      'NAME': 'Amelia',
      'LSAD': 'County',
      'GEO_ID': '0500000US51007',
      'COUNTY': '007',
      'STATE': '51'
    }'',
}
```

On failure, an error message is returned.

4.13 Update Feature

To update an existing feature, a POST request to the same endpoint as section 5.10, Feature Retrieval, is made and the uid of the specific feature to be modified is needed at the end of the url. The difference the REST request to Update Feature and Retrieve Specific Feature is that the former is a POST request with a URL parameter, and the latter is a GET request. The URL parameter required for updating is the same as the request parameter as initial feature insertion which can be found in section 5.8, Insert Feature, except there can only be a single feature's geometry in the content attribute. It is a JSON object with three attributes. This request will look up the feature with the feature uid provided in at the end of the URL endpoint and replace all data with the newly provided data in the URL endpoint as well as updating the `modified_timestamp` field of the Feature object.

On success, the uid of the updated feature is returned in the response. On failure, an error message is returned.

4.14 Delete Feature

Deleting feature is very similar to all other RESTful object deletions in the above sections. The uid of the feature to delete is required and appended to the end of the feature endpoint. For a feature with uid `7vR9quzqcM`, a HTTP DELETE request is made to the URL endpoint `https://pipelions.com/rapid/feature/7vR9quzqcM/`. On success, the same Feature object is returned as a GeoJSON object similar to retrieving a specific Feature. On failure, an error message is returned.

URL Endpoint	HTTP Request	Description
rapid/geoview/	POST	Create GeoView
	GET	Retrieve GeoViews
rapid/geoview/<geo_uid>/	GET	Retrieve Specific GeoView
	DELETE	Delete GeoViews
rapid/layer/	POST	Create DataLayer
	GET	Retrieve DataLayers
rapid/layer/<layer_uid>/	GET	Retrieve All Features of Specific Layer
	DELETE	Delete Layer
rapid/layer/<layer_uid>/?start=<start_time>&stop=<stop_time>/	GET	Retrieve Features of Specific Layer within a range of time
rapid/geoview/addlayer/<geo_uid>/<layer_uid>/	POST or GET	Add Layer to GeoView
rapid/geoview/removelayer/<geo_uid>/<layer_uid>/	POST or GET	Remove Layer from GeoView
rapid/feature/	POST	Insert Feature
rapid/import/<layer_uid>/	POST or GET	Import All Features from a Valid URL
rapid/feature/<feature_uid>/	POST	Update Feature
	GET	Retrieve Specific Feature
	DELETE	Delete Specific Feature

Table 4.1: RAPID REST API Endpoint Overview

Chapter 5

Validation

In order to test RAPID and ensure all of the requirements were met, we had a third party design and implement a system that uses RAPID's REST API to ingest, manipulate, retrieve, and display real data. This system, which we will call RAPID UI, serves as a proof of concept for the customers' software systems that will one day use RAPID. RAPID UI focuses on the quality and completeness of the API design which can be validated through fulfilling the requirements documented in Chapter 3.1. RAPID can be considered validated if it is able to perform all tasks required by the customers, returns thorough and valid results, and meets all standards in the specification.

5.1 Third Party System Specification

The validation of RAPID is being designed and implemented by another entity that was not present throughout the development of RAPID to ensure there was no prior knowledge to skew or bias the validation results. The developer of the validating web system is Kishan Patel, a Computer Engineering student at California Polytechnic State University. With no knowledge of REST APIs, only some knowledge of web development, and a focus of experience in graphics, he was a good candidate to test the ease of use of RAPID,

the quality of design, and correctness of implementation.

He began with a simple design using JavaScript with cross-origin resource sharing (CORS)[1] to make the foreign HTTP requests from client-side JavaScript to the RAPID pipeline server.

5.1.1 Independent Variables

The independent variables in this system test are the elements of the system that we can control. In general, they are the queries that we propose and submit to the system. The queries that the customer may be interested in is selecting Layers in regions and receiving the proper data. The specific parameters passed with the request in each query are the control, or independent, variables. The test is successful if the correct data is received given any legitimate and correct query.

5.1.2 Dependent Variables

The dependent variables are the elements that are affected by the independent variables listed above. Effectively, the data that is returned represents the dependent aspects of the validation test. We care about and are measuring the quality of the data and the speed at which it is returned.

We will define "quality" in our terms as the correctness of the content of the data, the correctness of its format, and if it is in accordance with the requirements. This measurement of quality is designed to validate the responses of the RESTful API as it connects with and communicates with the data model. Further validation of the database design and data modeling can be found in my colleague's thesis document's validation section[42].

[1]CORS is a mechanism that allows restricted resources (e.g. fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated[41].

5.2 Design

The third-party RAPID UI is designed to hit all end points that RAPID supports and ensure validity of each one. The list of queries that hit the REST API can be found below. Section 5.2.2 describes the business use cases that these endpoints fulfill and the order in which the REST UI completes them. The data that is sent back as the JSON[2] response is what is evaluated in order to validate RAPID's API.

5.2.1 List of Queries

- Create GeoView
  - End Point URL: http://pipelions.com/rapid/geoview
  - HTTP Method: POST
  - Request Parameters: JSON data object including geometry, descriptor, and properties
- Retrieve List of GeoViews
  - End Point URL: http://pipelions.com/rapid/geoview
  - HTTP Method: GET
  - Request Parameters: none
- Individual GeoView Metadata Retrieval
  - End Point URL: http://pipelions.com/rapid/geoview/8JrbghjzSaDCiitvYbWTyT
  - HTTP Method: GET
  - Request Parameters: none
- Create DataLayer
  - End Point URL: http://pipelions.com/rapid/layer

[2]JSON: JavaScript Object Notation

  - HTTP Method: POST
  - Request Parameters: JSON data object including descriptor, public, and properties

- Add Layer to GeoView
  - End Point URL: https://pipelions.com/rapid/geoview/addlayer/ 8JrbghjzSaDCiitvYbWTyT/mUy6ckBpUQzWDmiVqQg73Y
  - HTTP Method: POST
  - Request Parameters: none

- List of DataLayers Retrieval
  - End Point URL: http://pipelions.com/rapid/layer
  - HTTP Method: GET
  - Request Parameters: none

- List of Features within DataLayer Retrieval
  - End Point URL: http://pipelions.com/rapid/layer/mUy6ckBpUQzWDmiVqQg73Y
  - HTTP Method: GET
  - Request Parameters: none

- Retrieve Features within DataLayer in Time Range
  - End Point URL: http://pipelions.com/rapid/layer/mUy6ckBpUQzWDmiVqQg73Y/?sta &stop=1389177420
  - HTTP Method: GET
  - Request Parameters: none

- Insert Feature to DataLayer
  - End Point URL: http://pipelions.com/rapid/feature
  - HTTP Method: POST
  - Request Parameters: JSON data object including Layer, content (geometries), and properties

- Feature Retrieval
  - End Point URL: http://pipelions.com/rapid/feature/7vR9quzqcM
  - HTTP Method: GET
  - Request Parameters: none
- Update Feature
  - End Point URL: http://pipelions.com/rapid/feature/7vR9quzqcM
  - HTTP Method: POST
  - Request Parameters: JSON data object including Layer, content (geometries), and properties
- Delete Feature
  - End Point URL: http://pipelions.com/rapid/feature/7vR9quzqcM
  - HTTP Method: DELETE
  - Request Parameters: none
- Delete Layer
  - End Point URL: http://pipelions.com/rapid/layer/mUy6ckBpUQzWDmiVqQg73Y
  - HTTP Method: DELETE
  - Request Parameters: none
- Delete GeoView
  - End Point URL: http://pipelions.com/rapid/geoview/8JrbghjzSaDCiitvYbWTyT
  - HTTP Method: DELETE
  - Request Parameters: none

5.2.2 Use Cases

There are three main business use cases that RAPID UI executes successfully. These use cases each have to hit multiple endpoints in order to gather

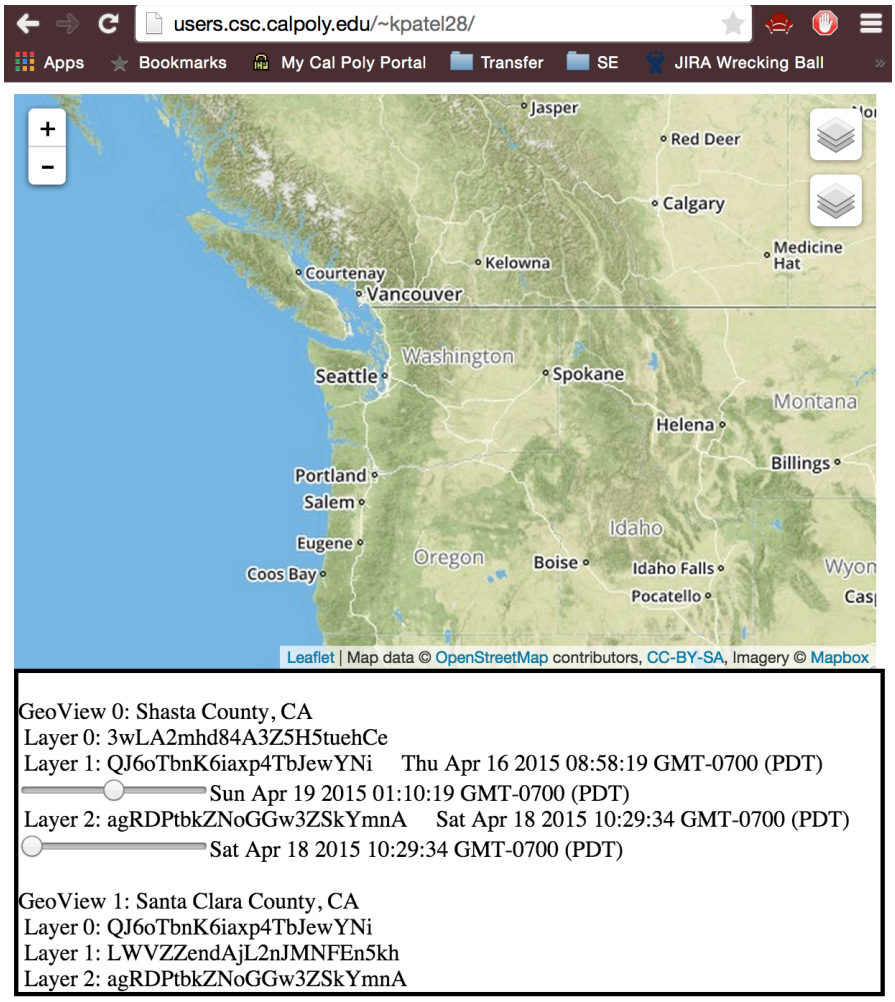

Figure 5.1: The browser window of RAPID UI webpage upon loading `http://users.csc.calpoly.edu/ kpatel28/`.

and display the necessary information. RAPID UI's third-party validation can be found at `http://users.csc.calpoly.edu/ kpatel28/`. When the webpage initially loads, it looks like Figure 5.1.

GeoView Selector

DSS may need to select a specific GeoView out of all of their accessible GeoViews in order to understand where the optimal right-of-way is in a larger region of land. When placing new pipelines, DSS are interested in a specific GeoView that has a set of layers; however, the customer with the DSS might have access to dozens of GeoViews. In order to select the GeoView that they

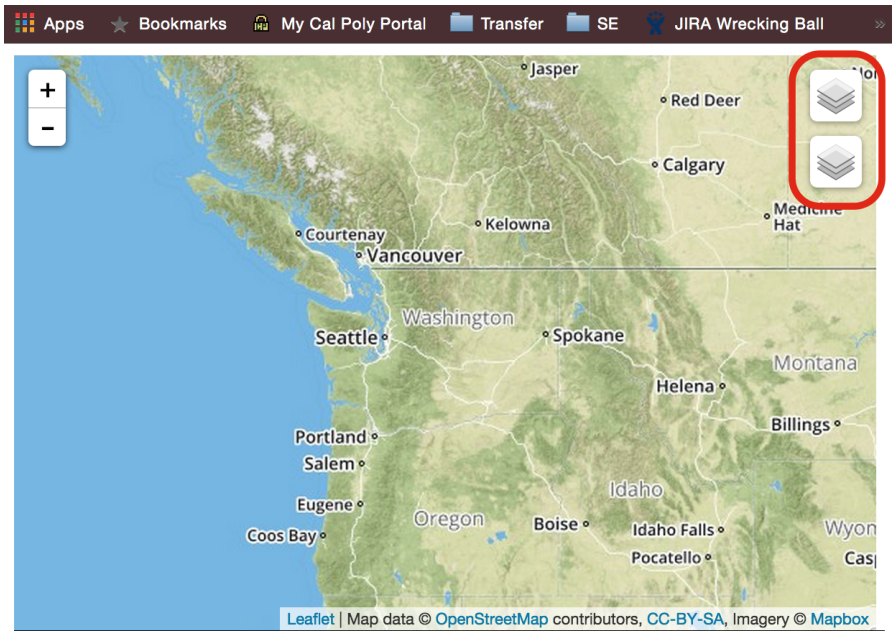

Figure 5.2: RAPID UI GeoView Selector.

are interested in, they need to first query RAPID for the list of all accessible layers. This can be done with the GET request to the `/geoview` endpoint. If the desired GeoView does not yet exist, they will then have to create a new GeoView. This is done through a POST request to the `/geoview` endpoint with the information desired and required.

The GeoView that is displayed in the RAPID UI gets all of the options with the general GET request first. It then uses the uid of the selected GeoView to send another GET request to the `/geoview` with the uid appended to retrieve all DataLayers that belong to the chosen GeoView. All GeoViews that are accessible are displayed in the bottom box of Figure 5.1 with some metatata. At the top right corner of the map, there is an icon for each GeoView that is present. For the RAPID UI validation, there are two goeviews so there are two icons that can be hovered over for more detail. These are highlighted in Figure 5.2.

Layer Selector

After either creating a new GeoView or retrieving a already created GeoView, the next step is adding DataLayers to the new GeoView. If the layers are already created, this can be done directly. Otherwise, first the layers need to be created. Creating a DataLayer can be done though a POST request to `/layer` with the desired and required information. After both the GeoView and DataLayer uid's are obtained, the layers can be added to the GeoView with a GET request to `/geoview/addlayer` with the GeoView uid and then the DataLayer uid appended with a slash between. To add multiple layers, that step can be repeated with different DataLayer uids.

The RAPID UI DataLayer selector is displayed by first gathering the information through a GET request to the `/layer` which retrieves a list of all DataLayers available. Selecting one or multiple of these layers reveals them on the map in that is displayed dynamically in the website. Figure 5.3 shows RAPID UI's interface of selecting a layer. When the user hovers over one of the GeoView icons in the top right corner of the map, it expands to show all layers that belong to that GeoView. The initial expanded GeoView for Shasta County, CA GeoView is shown in Figure 5.3.

Selecting one or multiple of these layers reveals them on the map in that is displayed dynamically in the website. Figure 5.4 shows what the map dynamically changes to if the user selects the earthquake layer (by selecting the checkbox next to the earthquake layer's uid).

The user can then dynamically unselect the earthquake layer and select the cities layer which will automatically modify the Features that appear on the map to display only the cities that fall within Shasta County, CA, shown in Figure 5.5

To display more than one layer within the Shasta County, CA GeoView, the user can select both check boxes, displaying on the web page what is shown in Figure 5.6

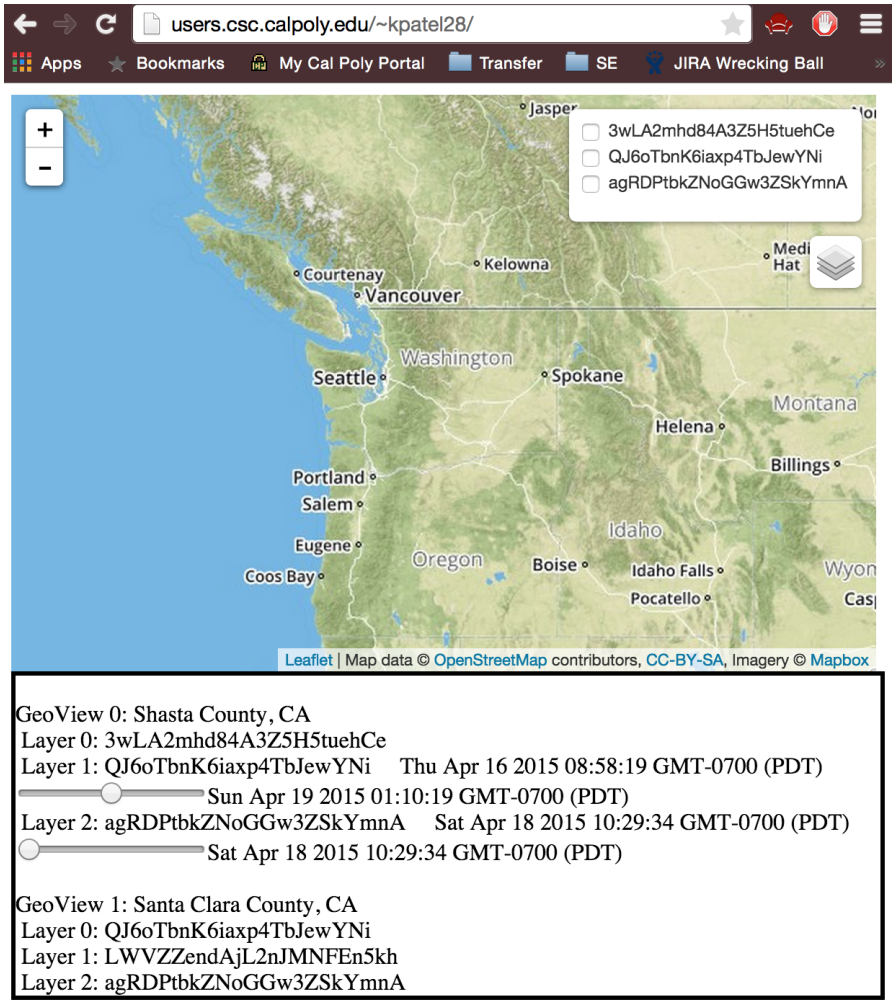

Figure 5.3: RAPID UI DataLayer Selector.

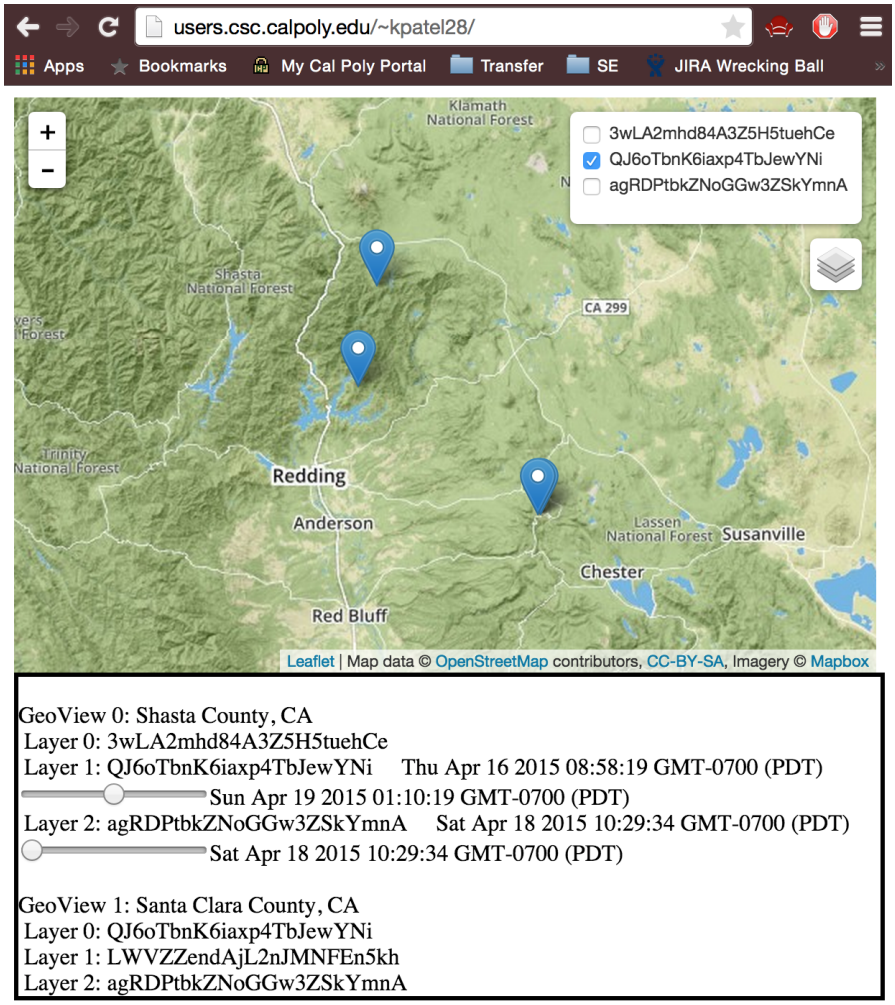

Figure 5.4: Earthquake Layer only of Shasta County, CA GeoView.

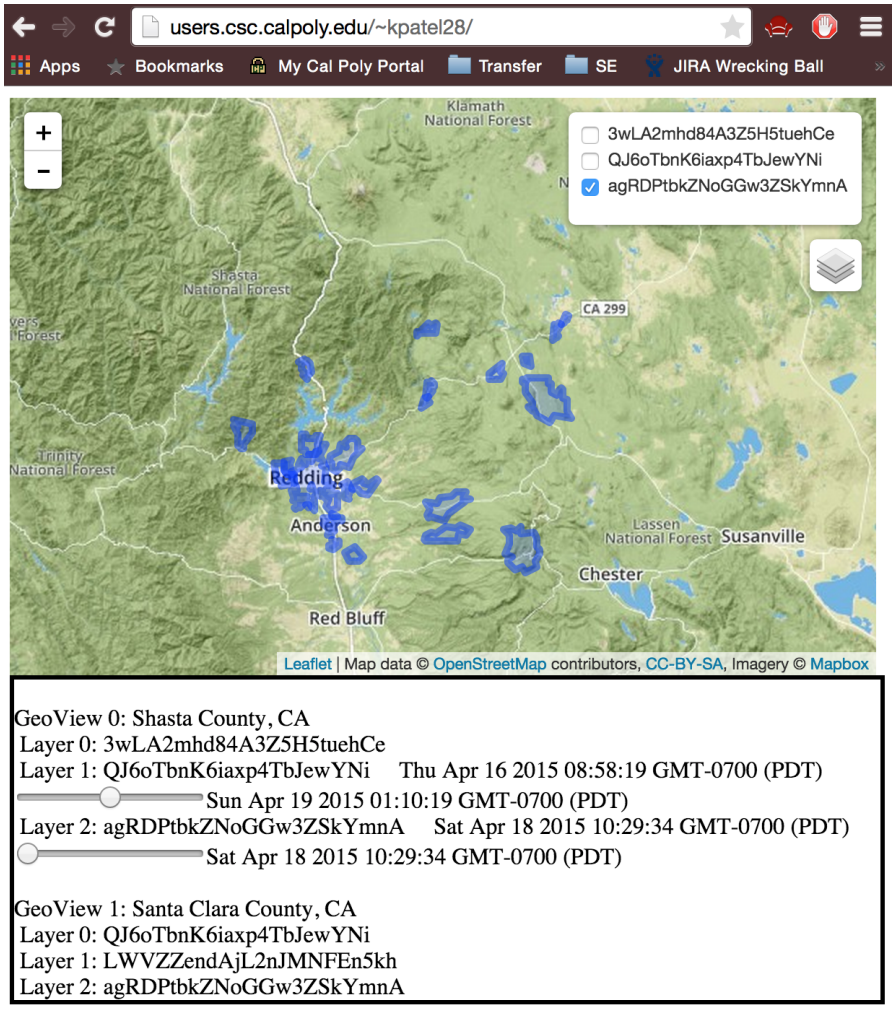

Figure 5.5: California Cities Layer only of Shasta County, CA GeoView.

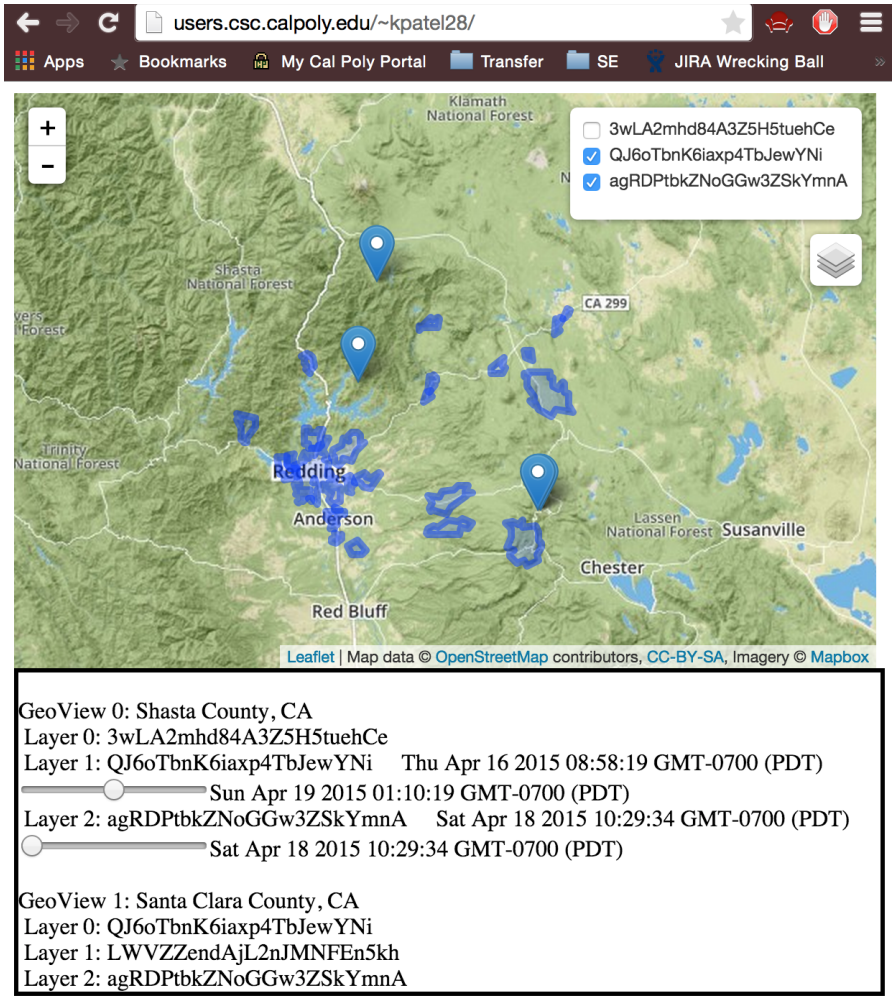

Figure 5.6: Earthquake and California Cities Layer of Shasta County, CA GeoView.

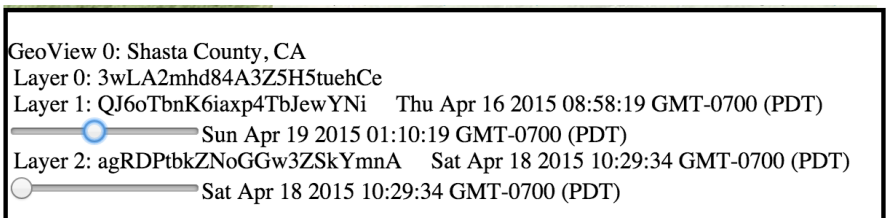

Figure 5.7: Time range slider for Features of Shasta County, CA Geoview.

State of Features at Specific Time

The date time slider in the RAPID UI simulates a freezing of RAPID's state at the given time. To select feature information that occurred between Wednesday of last week and Thursday of last week, the time selector can be moved to that date. Behind the scenes, RAPID UI sends a GET request to `/layer` with the desired layer uid, start, and stop times appended with a slash between each. This will modify the response from containing all features within that layer to only the features that occurred within the date times selected and display those on the RAPID UI map. The slider corresponding to Shasta County, CA GeoView can be changed to move the date-time range of Features that appear on the map. Figure 5.7 shows the slider for one GeoView.

5.3 Third Party Validation Purpose

The purpose of this third party website, RAPID UI, is to determine the API's quality and efficiency of use. A successful validity check of RAPID will return correct information for the given queries. For a specific GeoView, Layer, and Feature, provided by the GeoView uid, Layer uid, and Feature uid respectively, the response needs to behave as designed.

- Creating a GeoView inserts a new GeoView into the database with the provided geometry, descriptor, and properties and return a response containing the uid of the newly created GeoView;
- Geoview retrieval returns a JSON response containing a list of the ac-

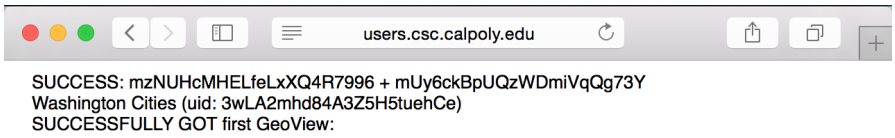

Figure 5.8: Response of Adding a GeoView to a Layer

cessible GeoViews from the provided valid API token;

- Individual GeoView metadata retrieval returns a JSON response containing the metadata of the GeoView with a uid 8JrbghjzSaDCiitvYbWTyT;
- Creating a Layer inserts a new Layer into the database with the provided descriptor, public boolean, and properties and return a response containing the uid of the newly created Layer;
- The request to add the existing Layer to the existing GeoView that was just created returns a response with both uids and a success message;
- Listing the existing Layers request returns a JSON response containing a list of the available Layers from the provided valid API token;
- The request to view a list of Features for a specific Layer metadata retrieval should get a JSON response containing the metadata provided valid Layer uid of mUy6ckBpUQzWDmiVqQg73Y;
- The retrieval of Features within a time range request for a specific Layer metadata retrieval should get a JSON response containing the metadata provided valid Layer uid of mUy6ckBpUQzWDmiVqQg73Y of only the Features that have timestamps within that time range;
- Creating a Feature inserts a new Feature into the database with the provided Layer, content (geometry), and properties, and return a response containing the uid of the newly created Feature;
- Feature retrieval returns a JSON response containing a single Feature for valid Feature uid of 7vR9quzqcM;

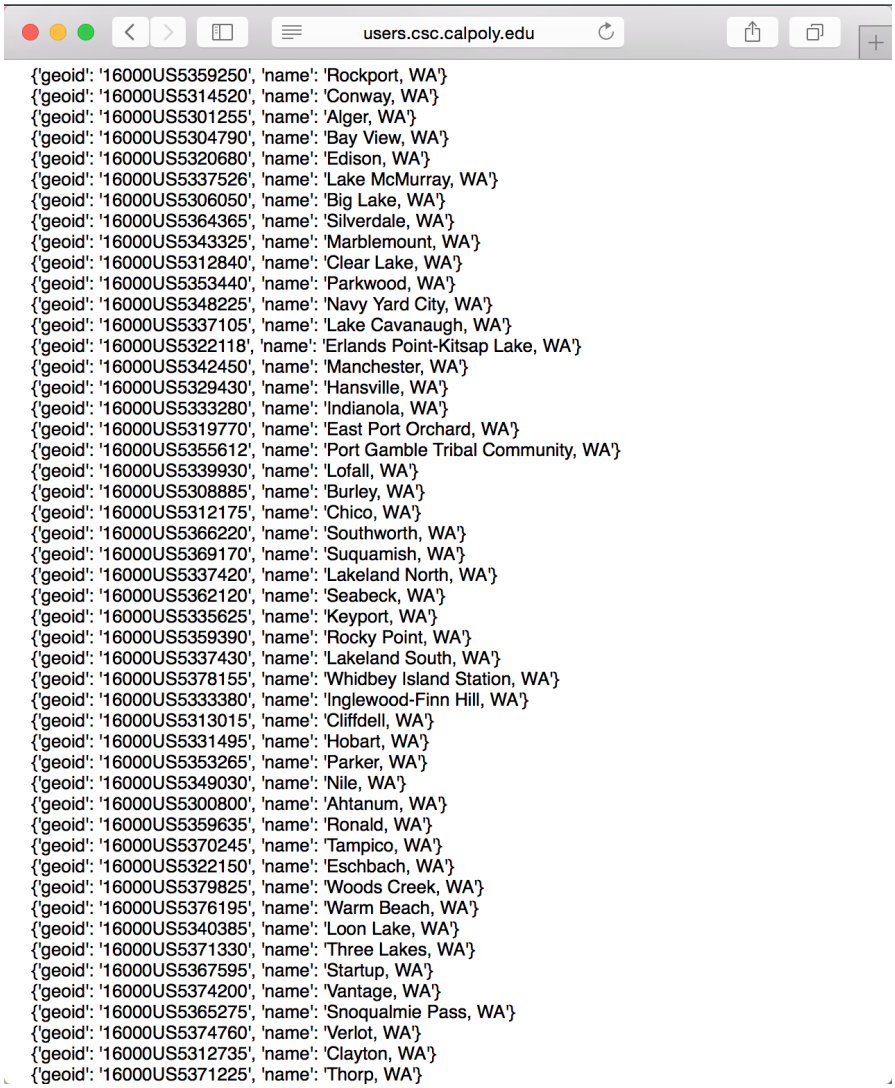

Figure 5.9: All Features in the US Cities Layer

- Updating a Feature updates the specific Feature with Feature uid 7vR9quzqcM with new Layer, geometry, properties, and returns a JSON response containing the uid of the updated Feature;

- Deleting the Feature with uid 7vR9quzqcM removes the Feature from the database and returns a response with a successfull delete message and the uid of the deleted Feature.

- Deleting the Layer with uid mUy6ckBpUQzWDmiVqQg73Y removes the Layer from the database and returns a response with a successfull delete message and the uid of the deleted Layer.

- Deleting the GeoView with uid 8JrbghjzSaDCiitvYbWTyT removes the GeoView from the database and returns a response with a successfull delete message and the uid of the deleted GeoView.

Chapter 6

Conclusions and Future Work

After understanding and exploring all of the factors that might affect pipes that carry vital resources, we were able to begin construction of a system that assists in detecting problematic events. The system that we designed, RAPID, has proven successful in addressing the requirements of pipeline operators of decision support systems. The root of the problem comes from the fact that the data that pipeline operators are interested in is stored in many different formats in many different locations physical and electronic. Decision support systems desire a single entity in which they can query for their geospatial data in a standardized format.

RAPID stores geospatial data that is relevant to pipeline integrity and new pipeline construction. This data ranges from precipitation and rainfall data, climate and weather changes, encroachments on right of ways and other construction, seismic activity, and much more. After RAPID ingests the data from multiple data formats it then stores the data in a PostGIS database modeled by Austin Wylie[42]. The data ingestion happens through a REST API. RAPID supports a number of queries documented in Chapter 4 to send, store, update, retrieve, and delete information. All of these queries can be done through RESTful POST, GET, and DELETE requests sent through the RAPID API endpoints.

6.1 Future Work

Many of the features and requirements that the customers desire have been met through rapid; however, there is much potential future work that can be done on rapid to improve it further.

6.1.1 OGC Compliance

The first item that would lead to improvement of RAPID has been addressed in Chapter 2.4 OGC Compliance Standard. Making the system OGC Compliant would allow for commercial use for customers in industry. This task is tedious but not trivial and in the upcoming months will be completed by someone that has extensive knowledge of the OGC Standard.

6.1.2 Data Formats

Currently, RAPID ingests geospatial data that originates in GeoJSON. Ideally, it would be able to ingest and parse data in multiple formats including Shapefiles and other common geospatial formats. This could be achieved through further parsing development. The same goes for formats of retrieve data. RAPID returns data in GeoJSON but DSS may hypothetically want to receive their data in other formats in the future.

6.1.3 Notifications

In early conversations with customers, there was discussion of an automatic notification system to alert DSS when there might be a threat to a pipeline caused by some data that was just ingested into RAPID. This was pushed out of our scope due to time and prioritizing requirements however would be a great future implementation task.

6.1.4 Automatic Ingestion

In order to get information automatically ingested so that instantaneous notifications can be sent out, the system would need a way to hook up to other data sources that are constantly updated, like earthquakes or weather. This task needs a way to set up a trigger when the data source is updated to hit an API endpoint in our system to import the new geospatial features.

6.1.5 Security

Encryption and security are huge concerns in today's increasingly electronic and online society. People are consistently trying to find new ways to hack into systems with valuable information. RAPID's stored data may eventually become important enough to its owner to provide a need for increased security. Currently, the security is done through API tokens given to each user that are passed through as requests are sent and RAPID API endpoints are hit. This future work involves both encrypting the data on the back end and introducing a more secure API token distribution scheme.

6.1.6 Scaling

Importing, ingesting, and parsing huge (e.g. 5GB or more) geospatial data files may be a requirement of customers in the future. Massive amounts of data must be imported in smaller pieces in RAPID's current state. Introducing a batch importing system that automatically partitions or distributes the load to complete the import in a timely manner would help to fulfill this large-scale requirement.

Chapter 7

Glossary

- RAPID - REST API for Pipeline Integrity Data
- System Entity - one of the objects or collections that can be represented in RAPIDa GeoView, DataLayer, or Feature
- Create - instantiate a new system entity with properties
- Update/Modify - change the properties, geometry, or other associated objects of a region, layer, or feature
- Ingest - parse data and insert into database in proper format by creating new system entities
- Geometry - a list of latitude and longitude pairs as a bounding polygon for a region
- Descriptor - a name or identifier that is external to the operation of RAPID, used to describe a system entity to outside users
- Properties - user-specified data in any format that describes a system entity
- Timestamp - a date and time (down to the second) identifying when a certain event occurs

- Start Time - feature was first noticed in this layer at this timestamp
- Stop Time - feature was first noticed gone from location at this timestamp
- Insertion Time - the timestamp for which the feature was inserted into the database
- Metadata - no specific or intended meaning in RAPID; describes information about data. Properties instead of metadata, should be used when describing the specific properties set that is associated with a feature or geometry.
- Feature - a object that contains geometry associated with properties
- Layer - a collection of related features with properties in a GeoView
- GeoView - a grouping of layers associated with a known and specified geometry and properties
- Encroachment - an incompatible or restricted use within the right-of-way; can include buildings and structures such as uninhabitable storage sheds, habitable room additions, pools, converted garages and other similar structures
- Pipelines rights-of-ways - strips of land of various widths in which pipelines are installed, either above or beneath the ground

BIBLIOGRAPHY

[1] Cal Poly Github. `http://www.github.com/CalPoly`.

[2] U. E. I. Administration. About u.s. natural gas pipelines, 2008.

[3] E. Augustine. Spoons: Netflix outage detection using microtext classification, March 2013.

[4] K. Bell. Automated student code assessment with symbolic execution and java pathfinder, June 2012.

[5] Boundless. Introduction to postgis, October 2014.

[6] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and C. Schmidt. The geojson format specification, June 2008.

[7] M. Cater. A brief history of api-based web applications, June 2013.

[8] E. Celeste. Geojson and kml data for the united states, November 2013.

[9] S. Chastain. Pipeline right of way encroachment: Exploring emerging technologies that address the problem, June 2009.

[10] Computerworld. Multi-tenancy in the cloud: Why it matters, April 2010.

[11] X. Cui, Y. Sun, S. Xiao, and H. Mei. Architecture design for the large-scale software-intensive systems: A decision-oriented approach and the experience, June 2009.

[12] R. DeLine and M. P. Robillard. A field study of api learning obstacles, December 2010.

[13] R. R. Downs and R. C. Chen. Developing an online resource center about geospatial data preservation, 2014.

[14] A. Dukovich. Design patterns go to hollywood, June 2008.

[15] D. M. Elkstein. Learn rest: A tutorial, 2014.

[16] M. Fan, H. Fan, N. Chen, Z. Chen, and W. Du. Active on-demand service method based on event-driven architecture for geospatial data retrieval, 2013.

[17] D. S. Foundation. The django framework, 2014.

[18] D. S. Foundation. Documentation: Model field reference, 2015.

[19] Y. Gang. A research on semantic geospatial web service based rest, December 2009.

[20] S. Gao, D. Mioc, and X. Yi. The measurement of geospatial web service quality in sdis, August 2009.

[21] P. Gas and E. Company. Natural gas pipeline rights-of-way, 2014.

[22] C. Hu, Y. Zhao, J. Li, D. Ma, and X. Li. Geospatial web service for remote sensing data visualization, March 2011.

[23] K. Lane. History of apis, December 2012.

[24] H. Li. Restful web service frameworks in java, September 2011.

[25] J. Li and N. Chen. Geospatial sensor web resource management system for smart city: Design and implementation, 2014.

[26] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu. Reengineering legacy systems with restful web service, August 2008.

[27] M. Malensek, S. L. Pallickara, and S. Pallickara. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals, 2013.

[28] M. McBride. Software architecture and design and ieee standards bundle, 2013.

[29] H. Meth. Decafs: A modular distributed file system to facilitate distributed systems education, June 2014.

[30] M. Nagpur. The development of web processing service using the power of spatial database, December 2009.

[31] A. of Oil Pipe Lines and A. P. Institute. Why do we need pipelines?, 2013.

[32] U. D. of the Interior. Tusgs cmg "shapefile" definition, November 2013.

[33] PostGIS. Spatial and geographic objects for postgresql, October 2014.

[34] M. P. Robillard. What makes apis hard to learn? answers from developers, October 2009.

[35] A. Robinson. Interacting with geospatial technologies, March 2011.

[36] M. Rouse. Race condition definition, 2015.

[37] Z. Sha and Y. Xie. Building spatial information systems based on geospatial web services, May 2009.

[38] G. Shi and K. Barker. Extraction of geospatial information on the web for gis applications, 2011.

[39] K. Stock and C. Cialone. An approach to the management of multiple aligned multilingual ontologies for a geospatial earth observation system, 2011.

[40] P. P. A. Vretanos. Open geospatial consortium, July 2014.

[41] Wikipedia. Cross-origin resource sharing, April 2015.

[42] A. Wylie. Surveying gis data storage and analysis, June 2014.

[43] J. Yuan, P. Yue, and J. Gong. Integration of geospatial web services and web portal technologies for geospatial information sharing and processing, August 2009.

[44] J. Zhang, D. Pennington, and W. Michener. Performance evaluations of geospatial web services composition and invocation, July 2007.

[45] J. Zhang and S. You. Large-scale geospatial processing on multi-core and many-core processors: Evaluations on cpus, gpus and mics, March 2014.

[46] H. Zhao and P. Doshi. Towards automated restful web service composition, July 2009.

[47] G. J. Zrate, J. Lisboa-Filho, and C. F. Sperber. Using the model-driven architecture approach for geospatial databases design of ecological niches and potential distributions, 2014.