GEOSPATIAL DATA MODELING TO SUPPORT ENERGY PIPELINE INTEGRITY MANAGEMENT

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

Austin Wylie

June 2015

COMMITTEE MEMBERSHIP

TITLE:	Geospatial Data Modeling to Support Energy Pipeline Integrity Management
AUTHOR:	Austin Wylie
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Professor Alexander Dekhtyar, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Christopher Lupo, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Gene Fisher, Ph.D. Department of Computer Science

ABSTRACT

Geospatial Data Modeling to Support Energy Pipeline Integrity Management

Austin Wylie

Several hundred thousand miles of energy pipelines span the whole of North America—responsible for carrying the natural gas and liquid petroleum that power the continent's homes and economies [28]. These pipelines, so crucial to everyday goings-on, are closely monitored by various operating companies to ensure they perform safely and smoothly [1].

Happenings like earthquakes, erosion, and extreme weather, however—and human factors like vehicle traffic and construction—all pose threats to pipeline integrity. As such, there is a tremendous need to measure and indicate useful, actionable data for each region of interest, and operators often use computer-based decision support systems (DSS) to analyze and allocate resources for active and potential hazards [1, 5, 11, 20].

We designed and implemented a geospatial data service, *REST API for Pipeline Integrity Data* (RAPID) to improve the amount and quality of data available to DSS. More specifically, RAPID—built with a spatial database and the Django web framework—allows third-party software to manage and query an arbitrary number of geographic data sources through one centralized REST API.

Here, we focus on the process and peculiarities of creating RAPID's model and query interface for pipeline integrity management; this contribution describes the design, implementation, and validation of that model, which builds on existing geographic standards.

ACKNOWLEDGMENTS

My thesis would not be what it is today (read: finished) without invaluable support and instruction. Several people have my deepest thanks:

- Dr. Alex Dekhtyar for advice, encouragement, and skills to work with biologists.
- Dr. Chris Lupo for the Picture of the Day, and teaching me to throw hardware at the problem.
- Alexa Francis—a solid teammate and neighbor—for learning about Web Feature Services with me.
- My parents, Suzi and Patrick, for academic cheerleading and a bunch of love.

This research was supported by the Department of Transportation (DOT) Office of Assistant Secretary for Research and Technology (OST-R) under Cooperative Agreement to California Polytechnic State University (OASRTRS-14-H-CAL)

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Problem definition

Several hundred thousand miles of energy pipelines span the whole of North America—responsible for carrying the natural gas and liquid petroleum that power the continent's homes and economies [28]. These pipelines, so crucial to everyday goings-on, are closely monitored by various operating companies to ensure they perform safely and smoothly [1].

Happenings like earthquakes, erosion, and extreme weather—and human factors like vehicle traffic and construction—all pose threats to pipeline integrity [5, 11, 20]. Compromised lines put business activities at risk; individual safety, personal property, and the surrounding environment feel the impacts as well [11]. However, with appropriate knowledge and management processes, these issues can be avoided [5, 11].

As such, there is a tremendous need to measure and indicate useful, actionable data for each region of interest. It is difficult, though, to keep a close-enough eye on massive pipeline networks with manual, in-person inspections [5, 11]. Even recurring aerial surveillance demands too much information from too few (and distant) sources [11].

As an alternative to visiting pipeline sites, operators often use computer-based decision support systems (DSS) to learn about and allocate resources for active and potential hazards [1, 11].

In general, these DSS allow operators to coalesce and study geographic, demographic, and remote sensor data relevant to pipeline integrity management [11, 30]. A great deal of worthwhile DSS data is already recorded, stored, and made available for analysis, but it is scattered across physical and digital media and locations. As such, pipeline operators are left to pick and choose smaller subsets of attainable and manageable inputs for analysis, which can end up painting an incomplete picture [11].

1.2 Contributions

We have created specific system requirements alongside web developers and industry partners to address their primary data aggregation and analysis needs. The remainder of this document describes the implemented, deployed, and validated software for improving the amount and quality of data available to pipeline operator DSS, which we dubbed *REST API for Pipeline Integrity Data* (RAPID). More specifically, RAPID allows third-party applications to manage and query an arbitrary number of disparate geographic data sources through one REST API. Rather than leaving DSS and their users to identify, model, convert between, and store dozens of data formats and repositories, our system manages and hides that trivia—instead exposing simple REST calls for more consistency and centralization.

A lot of geospatial data is conceptually alike because it pairs a physical location or demarcation with information about that place. The crux of the problem from Section 1.1, though, is that there are numerous and varied formats and access methods for those geospatial datasets. However, with appropriate modeling and processing, they can each be stored, modified, and queried in approximately the same way.

As we built it, RAPID uses the Python-based Django web framework with a PostgreSQL database. To store and query first-class geospatial data, PostgreSQL's PostGIS extension has been added and enabled, and our schema stores geometries and metadata located on Earth's surface.

Our most significant contribution to this project was the design and implementation of RAPID's data model (along with supplementary tools). In this document, we describe the process and peculiarities of building the RAPID model and query interface for real-world pipeline integrity management.

While the database itself is not accessible to external users, developers and applications can interact with the aggregated content through a well-defined REST API.[1] The API provides means of looking up geospatial data for particular regions (in space and time), as well as necessary metadata and documentation for its use and interpretation.

[1]REST, or *Representational State Transfer* is an architectural style for web service APIs, often using HTTP [16].

As a crude example, this allows API users to make a request like "retrieve all rain data for California in JSON" as well as a request like "retrieve all *seismic* data for California in JSON *from last week*." The rain data may have originated as JSON files from portable weather stations; the seismic data could have come from a Shapefile on a government server.[2] The point is that customers can view and download different forms of data from wildly different sources in a mostly-standard and interchangeable fashion.

My research partner, Alexa Francis, focuses on the REST API in her master's thesis, covering its implementation and usage, with considerations for third-party developers and our internal system architecture [16].

Finally, serving to validate our system, an external web application has been built to use RAPID—both retrieving and visualizing geospatial data. This additional software, RAPID UI, aggregates and indexes several datasets relevant to our pipeline operating partners and displays *a*) how they are organized within our system and *b*) where and what the geospatial features are.

RAPID UI looks up important locations and associated properties using the RAPID API, letting the UI perform quick and easy location-based queries with only HTTP requests—normally requiring custom-built logic for spatial processing. We have also used RAPID UI's activities to guide performance measurements and optimization. While our own test data does not strain RAPID as much as a pipeline-covered continent would, we also show system efficiency and stability during real, on-demand work.

1.3 Document outline

Following this chapter, Chapter 2 discusses the most essential and helpful context for creating RAPID, including related software, standards, and theory. Chapter 3 overviews our guiding set of system requirements—coming from partners and commonplace industry recommendations. Chapter 4 shows how we built RAPID, designing the data model and algorithms in Python. Chapter 5 describes our procedure for

[2]We discuss particulars for RAPID's supported formats in the following chapters.

validating RAPID, showing a correct and efficient implemented solution. Chapter 6 concludes this document, summarizing our work and other contributions that may follow.

Chapter 2

BACKGROUND

In this chapter, we provide helpful background on geospatial data and software. We describe the standardized concepts and programming interfaces RAPID builds upon, as well as the inspiration and direction we have taken from related work.

2.1 Introduction

Our partners' DSS use spatial data;[1] their eventual goal is to lay out and recommend pipeline routes through areas with as few hazards as possible [11]. As such, this end-to-end system can be considered a spatial DSS (SDSS). SDSS are uniquely tasked with providing "easy access to spatial data and decision models through the integration of spatial databases, analytical models, and visualization tools" [30]. RAPID is tasked with managing the spatial database and has to account for mathematical foundations in geography.

This chapter describes *a*) basic concepts in geographic software, *b*) structured and unstructured spatial data formats, *c*) spatial database management system usage, and *d*) related standards. These have all played into RAPID's origins, design, and implementation.

2.1.1 Open Geospatial Consortium

The Open Geospatial Consortium (OGC) is an international consortium—with several hundred industry, government, and academic participants—whose goal is to establish standard formats and interfaces for geospatial data and applications [24]. Many organizations insist on using OGC-compliant products and services to promote

[1] A note on terminology: some groups make a subtle distinction between "spatial" data and "geospatial" data. In those cases, spatial data describes data "distributed in three-dimensional space," with "measurable" dimensions [2]. Given *geography*'s definition, geospatial data is "spatial data which is related to the Earth" [2]. In this project, spatial data is always located on Earth, so "spatial" and "geospatial" are used interchangeably. As one supporting source, the United States Geological Survey considers "the terms 'spatial' and 'geospatial' [to be] equivalent" [2].

robustness and interoperability, and they have created over thirty-five standards, which often include API specifications and markup languages [11, 24]. While not a top-to-bottom OGC-compliant system, RAPID utilizes and implements components of OGC Standards and still builds on the same foundational concepts.

2.2 Geographic information systems

A geographic information system (GIS) is a computer system (including hardware and software) for working with geographic data—that is, data associated with a location in space [13, 32]. Quite simply, GIS provide a means of viewing and managing this data. More importantly, GIS allow people to understand deep, complex information about geographic locations, their relative positions, and the objects and conditions that are located there [13, 32].

Because many everyday and large-scale tasks and concerns are related to specific locations, so too are the computer applications that people use: from flight scheduling to car navigation, weather prediction, and social networking [13]. RAPID and its related applications, with their geospatial data, enable similar possibilities.

2.2.1 SDSS

The terms "GIS" and "SDSS" sometimes overlap ambiguously [27]. GIS might only involve general-purpose geography and cartography, but SDSS have the specific goal of improving geospatial analysis and resource management [27]. As such, SDSS *use* a GIS or could be considered a *type* of GIS [27].[2]

2.2.2 Functionality

GIS functionality can usually be placed in one of five categories [13]:

Mapping object locations. Descriptors can be assigned to points in space (take addresses, for example). In a technical sense, this associates string-like data—

[2]Third-party analyses are outside our work's scope; however, partners are exploring novel improvements to their DSS [11].

ideas or concepts (like place names)—with floating-point latitude and longitude. These are commonly called *features* (central to any geospatial system) [17].

Mapping quantities. Quantities can be mapped to certain points—to relating locations with calculable data about them. This could be integers and floating point values, again, assigned to latitude and longitude. These are also considered *features*. For example, earthquake have a *numeric* magnitude.

Mapping densities. Not only can values be assigned to specific points on a map, densities can be generated and displayed, which show the distribution of objects or values over an area. Continuous atmospheric datasets visualize this nicely.

Determining relative positions of objects. GIS can determine if spatial entities are located within or near each other. Additionally, they measure distances between objects and locations and determine whether or not they intersect. This can show when specific hazards encroach upon pipelines (all represented as features).

Analyzing trends. Many GIS look at trends in data. There is often demand and opportunity for analysis in how locations and features change over time: in particular, pipeline operators look out for increases in hazards and want to avoid areas where more are expected.

In summary, GIS capture, manage, analyze, and display geographically-referenced information; viewing and understanding the relevant patterns can be done with maps, graphs, and written reports [13]. RAPID, at a low level, acts as a foundation for that functionality.

2.3 Data characteristics

This section outlines more technical considerations for geospatial data, which is not always easily represented or stored in common formats.

Although humans often notate points in space with numbers (and, occasionally, symbols) it is best to treat them as their own primitive data type in software for

friendliness and robustness [32]. Because of this, geospatial data structures and file formats must be able to specify points in two or three dimensions: latitude, longitude, and elevation (if desired).[3] GIS and spatial database management systems often need to account for any arbitrary coordinate system [32].

Considering the above, two common file and data types have emerged in GIS: vector and raster. They are briefly described in Sections 2.3.3 and 2.3.4. We preface this with a needed introduction to standard geographic concepts from the OGC, including coordinate and projection system usage.

2.3.1 Abstract Specification

OGC's Abstract Specification describes and standardizes the essential components of geospatial data, including the central information model and glossary for all OGC Standards [22].

Any sizable geospatial software system is naturally governed by certain geographic and geometric properties and functions; the Abstract Specification is the OGC's formalization of those concepts, all of which are encompassed in the data types and formats in the upcoming sections [22]. The idea of a geospatial *feature* (described in Section 2.2.2 above), for instance, is essential [17, 22].

2.3.2 Simple Features Access

One foundational OGC standard, Simple Features Access (ISO 19125) defines the official data storage and access model for geospatial features [6]. As with many other standards and GIS-focused designs, Simple Features Access builds directly on the Abstract Specification [6, 22]. The SQL statements and spatial operators described later in this chapter are directly shaped by the Simple Features Access standard.

[3] Conceptually, time can be viewed as an additional dimension—one of a feature's properties—so the data is geospatial and temporal, or *geospatiotemporal*. RAPID includes a timestamp on its data and allows filtering based on that timestamp. See related discussions in Chapters 3 and 4.

Projection

There are many different mathematical and geographic coordinate systems and projections—created and suited for different tasks and locations—but they are not necessarily compatible with each other [17]. OGC standardizes unique Spatial Reference System Identifiers (SRIDs) in Simple Features Access to differentiate in-use coordinate systems [6]. To correctly identify, parse, and measure geometries, they must have a specific, known SRID [17].

In common with other GIS defaults, we specify RAPID to use SRID 4326, the standard World Geodetic System (WGS 84) [6, 21]. WGS 84 utilizes the latitudes and longitudes most people know (from -90° to 90° and -180° to 180°, respectively), and it is used in GPS [21]. Data imported into RAPID using other SRIDs must (and can) be transformed to SRID 4326 before performing spatial queries. This process is briefly described in Chapter 4.

2.3.3 Vector

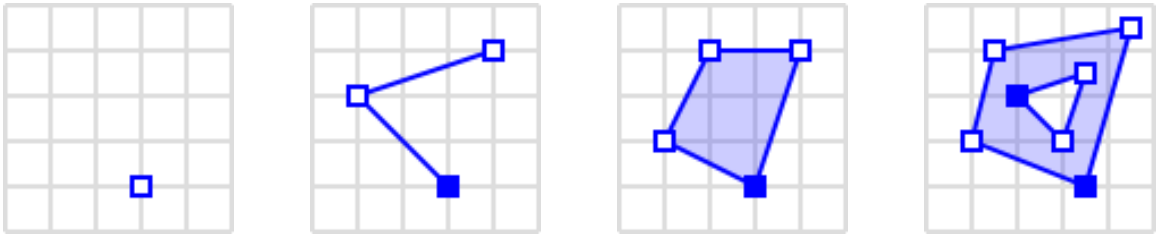
Vector files contain mathematical and geometric representations of spatial data, defining points and shapes in space. They are split into two components: geometries and attributes (or "properties").

Geometries

Coordinate pairs are two values that represent a point in space in a coordinate system [32]. Most commonly (and in RAPID), this is WGS 84, where one value is latitude and other is longitude; taken together, any point on Earth's surface can be represented.

In vector files, sequences (or "series") of these points can represent different geographic concepts, most often called "geometries." There are Points, Lines, Linestrings, and Polygons (shown in Figure 2.1) [32]:

Points. Points include the necessary coordinates to designate one position in space. See Figure 2.1a.



(a) Point (b) Linestring (c) Polygon (d) Polygon with hole

Figure 2.1: Standardized geometries from OGC's Abstract Specification.

Lines. Lines extend from one Point to another (and are represented by the ordered pair). See the Linestring in Figure 2.1b, which is composed of three Points and two Lines.

Linestrings. Linestrings are multiple connected Lines (and, thus, chained pairs of Points). These can represent objects or markers like road networks or pipelines. See Figure 2.1b.

Polygons. Polygons are Linestrings where the last Point is equal to the first—creating a closed shape. Political borders in the United States are a good example: the country is broken up into polygonal states, and states are broken up into polygonal counties, cities, and congressional districts. See Figure 2.1c.

Polygons with holes. Holes can be added to the interior of Polygons: the main, outer Polygon is represented by one Points series, and subsequent Point series represent internal gaps. Continuing the example of political geographic borders, one could imagine bodies of water occupying the holes of a Polygon. See Figure 2.1d.

Attributes

Attributes (often called "properties") are simply non-spatial data that is associated with the spatial data described above. An address (also mentioned above) is a great example: house numbers, streets, cities, states, and zip codes all associate an identifier with a point in space (or other geometric type) [32].

2.3.4 Raster

Raster data is simply grids of values. It is best to imagine a digital image: pixels in a matrix are each assigned red, green, and blue color values. When they are appropriately combined and visualized, a continuous dataset spreads over a region in two or three dimensions [32]. Images, in fact, are a type of raster data often found in GIS (like satellite images). Atmospheric sensors and elevation maps, for example, can also provide datasets that blanket an area [32].

Because raster data is continuous over a region, programs and users need to account for the *density* of data—that is, the size of each grid cell, which is referred to as "resolution" [32]. Resolution is either spatial or spectral:

Spatial. Spatial resolution is how large a cell of the grid is and how it corresponds to a real geographic area [32]. For example, a side of one cell could be four feet.

Spectral. Spectral resolution is the number of color "bands" in raster images [32]. In an ordinary image, red, green, and blue are these three bands—the spectral resolution [32]. In some cases, images may also capture infrared or ultraviolet light and store more data in other color bands [32].

Georeferencing makes raster data useful geographically: it associates the raster data with a location. For example, a GIS that displays a satellite image of California on top of an interactive state map can calculate where real geographic features are in the image [32].

With the following four pieces of information (and some accompanying calculations), a raster file can be properly georeferenced and positioned in a coordinate system [32]:

- Coordinates of the file's top-left pixel
- Width of a pixel
- Height of a pixel
- Rotation of the grid

RAPID does not query raster data directly, but its geographic footprint can be represented as a vector, with its additional information encoded as serialized properties.

2.4 Data storage

This section overviews several storage considerations for GIS data that is most relevant to RAPID.

2.4.1 Database management systems

Because of the unique data considerations described earlier, databases that manage geospatial data need to pay special attention to the methods of doing so. Spatial database management systems (spatial DBMS) provide ways of storing and querying geospatial data [3]. This includes syntax for adding vector and raster data types [3].

Example

Inserting geometries (vectors) into a spatial DBMS canbe done as in Figure 2.2, where we use a SQL statement to add geometries to a PostGIS table (RAPID's underlying DBMS). These SQL statements demonstrate the creation of four geometric types in a geospatial database: Point, Linestring, Polygon, and Polygon with holes. The results of the SELECT are shown in Figure 2.3. Note that the generic `geometry` data type, shown in the figure's `geometries` table, is also standardized in Simple Features Access, which can store any of the more specific geospatial types [3, 6].

In terms of notation, the space-separated numbers are a coordinate pair. Commas separate the points, creating ordered series. This particular syntax is Well-Known Text (WKT), an OGC Standard for defining Abstract Specification geometries [24]. Those PostGIS geometric data types—`POINT`, `LINE`, `LINESTRING`, and `POLYGON`—correspond to the vector types described above in Section 2.3.3.

```
CREATE TABLE geometries
 (name varchar, geom geometry);

INSERT INTO geometries VALUES
  ('Point',
   'POINT(0 0)'),
  ('Linestring',
   'LINESTRING(0 0, 1 1, 2 1, 2 2)'),
  ('Polygon',
   'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
  ('PolygonWithHole',
   'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),
   (1 1, 1 2, 2 2, 2 1, 1 1))'));

SELECT name, ST_AsText(geom)
FROM geometries;
```

Figure 2.2: Example spatial DBMS operations.

```
"name";"geom"
"Point";"POINT(0 0)"
"Linestring";"LINESTRING(0 0,1 1,2 1,2 2)"
"Polygon";"POLYGON((30 10,40 40,20 40,10 20,30 10))"
"Polygon with hole";"POLYGON((35 10,45 45,15 40,10 20,35 10),
                            (20 30,35 35,30 20,20 30))"
```

Figure 2.3: `SELECT *` results from Figure 2.2.

Querying

Simple Feature Access, introduced above, defines several required methods for comparing and relating Geometries, summarized below [6]. Figures 2.5 through 2.7 visualize several of these relational operators, as examples.

Equals. Tests for equality of two Geometries.

Intersects. Tests whether the interiors of Geometries intersect. See Figure 2.4.

Disjoint. The opposite of intersection.

Cross. Tests if the intersection of two Geometries is in one dimension fewer than the source Geometries.

Overlap. Determines if the intersection of two Geometries is different from both the source Geometries but of the same dimension. See Figure 2.6.

Touch. Tests whether Geometries have their boundaries touching (without intersected interiors). See Figure 2.5.

Within. Test if a Geometry is fully inside of another. See Figure 2.7.

Distance. Calculates the shortest distance between two Geometries. Note that this is the only operator without a boolean result.

DWithin. Tests whether Geometries are within a specified distance of each other. Note that this is the only boolean relational operator that takes an additional argument (the distance).

It is easy to see where and why these come in handy. As a simple example, imagine that a mobile device stores its geographic location—a Point—and is trying to determine the city it is currently in. If a DBMS stores borders (Polygons) for cities, a query can be performed that asks if the Point is within cities' borders [3].

The queries in Figure 2.8 show some similar ideas that are unique to geospatial data: the first selects the neighborhood(s) that the Broad Street subway station is located in, and the second selects points of interest within 1609 meters of any road.

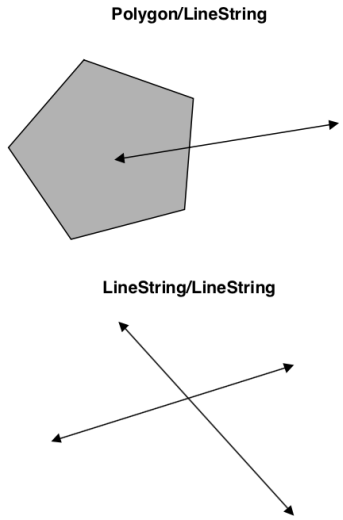


Figure 2.4: Geometries exemplifying the 'Intersects' relationship.

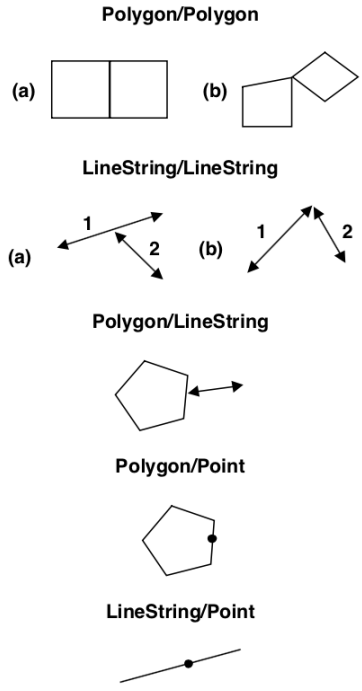


Figure 2.5: Geometries exemplifying the 'Touch' relationship.

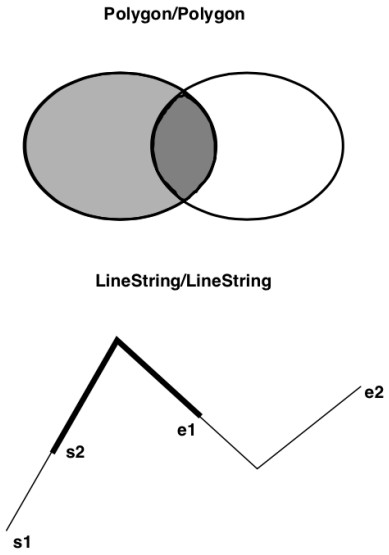


Figure 2.6: Geometries exemplifying the 'Overlap' relationship.

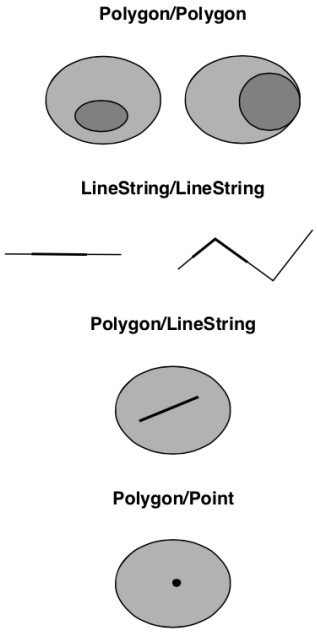


Figure 2.7: Geometries exemplifying the 'Within' relationship.

```
SELECT nyc_subways.name, nyc_neighborhoods.name
FROM nyc_neighborhoods
JOIN nyc_subways
ON ST_Contains(nyc_neighborhoods.geom, nyc_subways.geom)
WHERE subways.name = 'Broad St';
```

```
SELECT roads.roadname, pois.poiname
FROM roads
JOIN pois
ON ST_DWithin(roads.geom, pois.geom, 1609);
```

Figure 2.8: Example queries using geospatial operators.

This type of analysis can be done application-side (as opposed to in the database) but DBMS technologies have advanced enough that this type of data can be stored and queried very efficiently, with user-friendly queries [3].

2.4.2 Files

As can generally be the case with file formats and proprietary or fragmented software, geographic file formats are numerous and varied. Several come from cartographic- and surveillance-focused government agencies, like the United States Geological Survey and the National Geospatial-Intelligence Agency [33]. Other GIS vendors may have their own file considerations and requirements [4, 12, 29].

Esri Shapefiles

A go-to file format for vectors is Esri's Shapefile, which includes the two vector components described above: attributes and point data [12]. Shapefiles are a (mostly) open and standard format managed and developed by Esri for use within the GIS space (and particularly their popular software, ArcGIS) [12]. Although this is not an OGC Standard, it has become a ubiquitous vector format for GIS.[4]

Despite its singular name, a "Shapefile" is actually a set of several different file

[4]Interestingly, Shapefiles have begun to show their age (they originated in the early 1990s), and there were calls for a more modern, robust shapefile format from the OGC, which resulted in GeoPackage, a modern vector format based on SQLite. Shapefile's ubiquity and history keeps it in widespread use, however [25, 29].

types (including but not limited to) [12]:

`.shp` These files contain actual shape geometries.

`.dbf` These files store the non-spatial attributes that are associated with the geometries in the shapes file.

`.shx` These are index files that store metadata about entry locations in a file and allow for seeking forward and backward between features.

GeoJSON

GeoJSON is another common vector file format, although it is not standardized by OGC. It stores data in a nicely-readable format that is a subset of JSON (JavaScript Object Notation). Take this example of a geospatial feature in GeoJSON:

```
1 { "type": "Feature",
2     "bbox": [-180.0, -90.0, 180.0, 90.0],
3     "geometry": {
4       "type": "Polygon",
5       "coordinates": [[[-180.0, 10.0], [20.0, 90.0],
6                        [180.0, -5.0], [-30.0, -90.0]]]
7     }
8 }
```

It should be apparent that this syntax corresponds closely to the Abstract Specification we have been discussing, with standardized geometry types notated by lists of Points (latitudes and longitudes in this case). Arbitrary, situation-specific properties can be appended in the same fashion as the key-value pairs above.

Bounding box. The *bbox* attribute is a bounding box, which serves as an *approximate* area—bounding, rectangular dimensions—for a geospatial feature [3]. Although actual feature coordinates may describe a more intricate border, a rougher bounding box is less complex and more efficient to compute spatial queries for—sometimes a good-enough replacement for the slower (but precise) queries. In some cases, it is reasonable to check spatial queries on bounding boxes and, depending on the results,

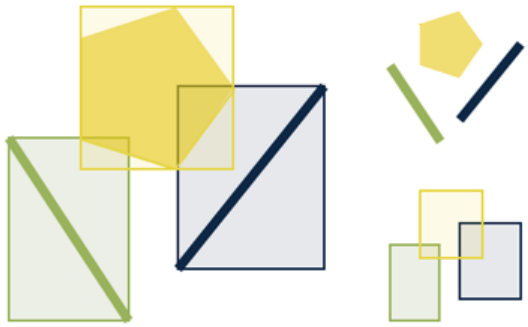


Figure 2.9: Bounding boxes shown for three example geometries.

later check the true geometry for additional accuracy [3]. A bounding box is not unique to GeoJSON and is commonly found in geospatial data specifications and APIs [3].

See Figure 2.9 for examples, which nicely shows the difference between querying bounding boxes and querying true geometries. Studying the bounding boxes, we would quickly learn that *a*) each of the lines *might* intersect with the Polygon but *b*) the lines surely *do not* intersect with each other. Examining the true shapes takes longer but reveals that *none* of the geometries intersect.

Related formats

There are dozens of other geospatial file formats with certain specializations, degrees of standardization, and application-specific features, but Shapefiles and GeoJSON are nicely representative of their general capabilities and stylistic differences (and are RAPID's initially-support formats).

Chapter 3

REQUIREMENTS

The previous discussions overview motivations and practices for pipeline integrity management. We have also covered the geographic and technological framework that RAPID works within—primarily adhering to OGC's Abstract Specification. Alongside external pipeline operating partners, our team established additional requirements for RAPID's domain-specific functionality.

At a high level, our goal is to combine the earlier-mentioned spatial operations and formats with updatable and searchable geographic data. Partner organizations also require permissioning capabilities (so that private and proprietary information is not shared with unprivileged requestors).

These broad characteristics are spelled out below, and we describe the subsequent system design for meeting these requirements in Chapter 4.[1,2]

3.1 Related work and standards

We discovered several geospatial data stores and aggregators, but RAPID's design does not directly draw from these related systems. They sometimes perform similar tasks, so we pause to clarify differences.

3.1.1 Web Feature Service

OGC's Web Feature Service (WFS) is a lengthy and highly-specific standard for geospatial data storage, formatting, and transmission [34]. Several existing refer-

[1]Of additional note, these requirements are particularly suited to RAPID's *data model*. Other tertiary requirements exist for the REST API—pertaining to its style and usage—but many of those are derived from the central points of this chapter. Francis, in her thesis, describes the API's requirements and design in depth [16].

[2]Also be aware that our contribution is primarily visible at the code level, but some of the terminology and capabilities carry through to external developers and end users. These requirements end up spanning different stakeholders and levels of abstraction, but they are still the most important asks and agreements that formed our design.

ence implementations also exist [23]. While this complex, in-use standard outdoes RAPID's capabilities in some scenarios, there is currently no standard specification for spatial REST APIs [34].

This gives our team the opportunity to create the ecosystem we see fit—even allowing non-GIS developers to incorporate RAPID data relatively pain-free. Our validation software, RAPID UI,[3] for instance, would not otherwise have the resources to interface with a WFS directly, but can still make use of RAPID because of its idiomatic REST API.

3.1.2 FeatureServer

One related system we are aware of, a 2012 open-source project called FeatureServer, implements many OGC standards, allowing for automated and full-featured data querying—even adding a REST API for some use cases [15].

To clarify the difference between FeatureServer and RAPID, though: FeatureServer expects users (developers) to have a thorough understanding of these specific OGC standards, and it acts mostly as a set of helper tools for customers already working amongst those standards [15]. While FeatureServer is a sort of stepping stone between GIS and WFSs, RAPID is lighter-weight and focuses on gathering and serving real data, even when it may not end up in a commercial GIS.

3.2 Supported formats

As outlined in the Background, numerous geospatial data formats exist for use in various situations and software. We describe the two formats we chose to support: GeoJSON and Shapefiles.

3.2.1 Input

Throughout our own experimentation and research, GeoJSON and Shapefiles (both summarized in Chapter 2) emerged as reasonable starting points for file importing in

[3] Introduced in Section 1.2 and detailed in Chapter 5.

RAPID. There are several reasons:

Availability. Many (or even most) high-profile, public geospatial datasets are archived in one or both of these formats, including ones of interest for our external partners.

Standardization. While not wholly formulated by the OGC, both GeoJSON and Shapefiles incorporate the Abstract Specification, and other components of the formats are open standards [4, 12]. This, in part, drives their ubiquity and availability.

Shapefiles, primarily Esri's creation for their ArcGIS software, include many standard *and* non-standard amendments and appendages. While some of these are used to enable extended functionality in proprietary systems, they are often used for (optional) performance tuning [12]. That is, the most essential vector data is openly specified, and the other components are not particularly important to us.

Toolsets. Thanks to their standardization and popularity, these formats are well-supported by many open-source spatial libraries and frameworks. This means a lot of robust, reusable functionality for parsing and validation already exists (which we use and discuss later).

Abstraction and conversion. Again, with the consideration that these vector formats are standardized and built on top of the Abstract Specification, their data types can store and represent the *same information* [22].

Further, these representations can be converted from one to another: the vectors of a valid GeoJSON file can be entirely converted to a Shapefile with the same vectors. This also means that end users—even third party developers—do not often need to know the underlying detail of which file format was used at one point or another. In the case that one particular format is required, the conversion is mostly trivial.

GeoJSON readability and ubiquity. GeoJSON's relatively user-friendly and readable syntax (extended from JSON) lets nontechnical stakeholders and contrac-

tors understand (and even generate) structured geospatial data with minimal training.

Additionally, the JSON component of GeoJSON particularly suits it for other programming tasks (and software that is not necessarily in the GIS space). Many other formats are only known and initially usable by geospatial professionals, but JSON is accessible for a large portion of developers.

3.2.2 Output

For the same reasons that we settled on GeoJSON and Shapefiles as input formats, they both made sense as output formats, and data in RAPID can be exported as either (no matter the input format).[4] While a commercial, enterprise-ready application might implement several extensive OGC standards (like WFS) for data exchange and parsing—enabling instant plug-and-play functionality for some GIS—GeoJSON, supplemented by Shapefiles, was chosen as a reasonable starting point. Additionally, it is not difficult to find open-source libraries that convert these formats to other OGC-compliant ones.[5]

3.3 Data storage

As discussed, the core geographic data model, with geometries and arbitrary properties, is mostly standardized. RAPID begins to differentiate itself with how data is grouped and queried: it is a requirement to have "layers" of data, collecting features within user-defined categories—(think categorization of weather stats versus census numbers).[6]

[4]This decision was helpful as a practical matter, too: with the tools and libraries we use, it is less difficult to export formats we have already researched and configured for decoding.

[5]This functionality could be added to RAPID, it just does not exist yet.

[6]In practice, we do not dictate how to separate layers logically or assign their metadata, but the concept of disparate (but grouped) data is central to RAPID's functionality.

3.3.1 Example

Every five minutes, the United States Geological Survey (USGS) publishes a GeoJSON file on their website of earthquakes from around the world (locations, magnitudes, depths, and such). If RAPID is asked to import that data every five minutes also, after some hours, it can process tens of GeoJSON input files from the USGS containing hundreds of spatial seismic measurements, all slotted into the same "earthquake" category. Our design later names these collections of like features *DataLayers*.

While visible under the hood, RAPID will not reveal which features came from which files. Even data fetched from several different sources is, effectively, merged (that is, a pipeline operator could have more granular seismic data alongside the USGS feed); from then on, all seismic data is accessible through one uniform, queryable DataLayer.

3.3.2 Geospatiotemporality

Trend-watching provides an important perspective on geospatial data: pipeline operators and DSS find it helpful to narrow datasets by a time range for in-depth study. Therefore, RAPID must include a timestamp with its stored geospatial features, letting requestors retrieve only the most relevant datapoints, which are *geospatiotemporal*. By analyzing other metadata along with this timestamp, DSS can find desirable or worrying trends.

3.3.3 Updatability

In the previous example, we alluded to RAPID's data modification capabilities: existing DataLayers need to seamlessly integrate new measurements as they become available.

Similarly, stored data must be modifiable at the feature level through the API; users and applications should not have to edit and resubmit whole Shapefiles or GeoJSON files to slightly tweak stored objects. In this way, the entire procedure is programmable and allows for fine-grained manipulation when necessary.

3.4 Querying

Introduced in 3.3, RAPID must be programmatically queryable (for the sake of external DSSs), fetching subsets of spatial data filtered by layer, location, and time, and DataLayers can be further characterized by this process. In most cases, data from a single DataLayer will be analyzed similarly or aggregated. If a pipeline operator DSS seeks rainfall trends (for example), every feature in RAPID's rainfall DataLayer can be returned, and relevant information is available for study.

For the sake of efficiency and user-friendliness, RAPID can offload, from external software, "cropping" of data by time and region. That is, users have the option to specify a sub-area and timeframe for data export. Refer back to Chapter 2 for specific examples of spatial table selections and joins, but consider RAPID's common use case: the system will gather, export, and transfer recent geographic features for a defined space (represented as a Polygon). Features within[7] the Polygon and in the correct time period are processed and returned to requestors.

3.4.1 Monitored regions

We solidify the concept of queryable view subsets by making them nameable and saveable—stored in RAPID for indexing and retrieval. Our design later names these stored, on-demand filters *GeoViews*.

We let users create GeoViews by associating a specific Polygon with a set of desired DataLayers. Any time a requesting user or DSS looks to analyze features local to the region, they simply query RAPID for the chosen GeoView and, optionally, include a start and end time for data. Any features *in the specified region* (and in the optional timespan) are compiled into GeoJSON or Shapefiles and returned to the API caller.

[7] `Within` is a function in Simple Features Access, but we are not using the term as such here. We simply mean to designate features contained by or encroaching on the specified region.

3.5 Multi-tenancy

Although RAPID's marquee feature is gathering and sharing data from multiple (and discrete) sources, open and total access discourages organizations from storing and querying sensitive or trade-secret data. Because aggregating and efficiently filtering this data *is* a requirement for our pipeline operating partners, RAPID needs permission management capabilities for DataLayers and GeoViews, making this a multi-tenant system.

Chapter 4 explains our developed permissioning system in further detail, which allows decision makers to hide information when appropriate. The permission model is not overly complicated, but it does have to be precise and secure. In broad terms, DataLayers require Owners that decide their visibility for other users. Non-Owners can be made Viewers or Editors for accessing or modifying data, respectively. This lets users specify which other users can work with DataLayers they create, and RAPID provides the authentication capabilities to ensure requestors and curators operate within assigned privileges. DataLayers not sensitive in nature and otherwise widely-viewable may be specified as *public* layers.

3.6 Modularity and abstraction

While this is not an explicit technical requirement, the system architecture should be modular—easily-maintainable with concerns separated. This is for a few reasons:

RAPID will likely see further development in the next several years with different software engineers and stakeholders. While the current RAPID system is a helpful tool for many geospatial operations, we recognize the demand for a broader platform of data aggregation, monitoring, and retrieval, allowing more extensive functionality down the road.

Ideally, more advanced functionality will fit directly into the models and workflows we have already created; when it does not, the RAPID codebase still must be flexible enough to incorporate new work without entirely dismantling existing features and recreating large components from scratch.

From discussions with external partners, we have heard that potential add-ons could be support for additional file formats, near-real-time notifications, or even encryption. New developers should be able to join the project, understand the system design quickly, and then add features like these wherever they most belong. In Chapter 6, we discuss potential future work.

Chapter 4

SYSTEM DESIGN

We designed RAPID to fulfill the discussed system functionality and stylistic expectations discussed in Chapter 3. This chapter focuses on our conceptual framework for moving toward those requirements; going from abstract to concrete, we also detail the process for converting these specifications to deployed source code—mapping broader logical entities and actions to specific modules, methods, and tables in Python and SQL.

Of further note, RAPID's design is mostly presented at the same level of abstraction as OGC's Abstract Specification—utilizing its standardized types and operators, just in an application-specific context. That is also to say, our model, with minimal modifications, could be implemented in any number of procedural programming languages and spatial DBMS. Our design decisions specifically aim for flexibility, clarity, and robustness, as there might be further development in the works.

One of our primary goals, here, is to spell out RAPID's data model and data flows. We describe basic storage and querying (as well as constraints)—continuing to build on the discussed standards—along with expected business rules and ancillary features. Among the high-level model descriptions, we share portions of source code and describe how we use the Python-based Django web framework and PostGIS DBMS to implement RAPID.

4.1 Introduction

We preface our contribution with a quick, helpful introduction to Django and related API work by Francis. We, additionally, describe our chosen pattern for cross-module method calls and finish by outlining a larger example we will continue for the rest of the chapter.

4.1.1 API design and capabilities

In concert with Francis, and regarding the requirements from Chapter 3, we defined several high-level API calls and activities for customer software (some of which are broken into sub-steps or combined where possible):[1]

1. Creating DataLayer instances, including metadata.
2. Importing GeoJSON and Shapefiles, adding their geospatial features to a designated DataLayer.
3. Creating GeoView instances, including geometric boundaries and metadata.
4. Choosing DataLayers for a GeoView.
5. Browsing and filtering DataLayers, Features, and GeoViews.
6. Setting permissions on DataLayers and GeoViews.

Additionally, to manage permissions, each external system is assigned a unique and private API token that identifies them to RAPID. In the above API activities, the caller must include their secure token to be identified and provided properly-privileged data.

4.1.2 Django usage with PostGIS

To execute on RAPID's chosen functionality, Django (using a PostGIS database[2]) quickly became a clear choice for our implementation framework, and we show how ordinary Python classes persist and are queried. These factors mainly influenced our decision:

1. Django's object-relational mapper lets us form database models quickly and use them wherever necessary in our Python codebase. Rather than crafting custom

[1] The specific API syntax is not essential to our discussion here. We list external DSS' primary tasks when using RAPID, but Francis provides more precise, developer-focused documentation [16].

[2] Although other spatial DBMS exist, PostGIS is open-source and easily connects with Django. MySQL, another free and open-source option really just *stores* geometric data and only supports cursory spatial calculations [7].

SQL queries, Django easily exposes tables and columns as classes and properties in Python.

2. Django is, sometimes, partially branded *GeoDjango* and makes available a lot of GIS functionality—particularly in parsing and converting geometries.

Referring to the technical advantages above, note further details and examples below. After this section, we will not dwell as much on Django implementation details: assume that future model creation and querying works in approximately the same form as these small tutorials.

Django models

Django creates and manages tables in our PostGIS database, mirroring them with classes defined in a `models.py` file. See how Django turns a Feature class into a SQL `CREATE TABLE` statement from Figure 4.1 to 4.2.

```
class Feature(models.Model):
    uid = models.TextField(primary_key=True)
    geom = models.GeometryField(null=True)
    bbox = models.PolygonField(null=True)
    properties = models.TextField(null=True)
    create_timestamp = models.TimeField(auto_now_add=True,
      null=True, db_index=True)
    layer = models.ForeignKey(DataLayer, null=True)
    hash = models.TextField(null=True, unique=True, db_index=True)
```

Figure 4.1: Feature class in models.py.

Many of the arguments in a Feature's fields (Figure 4.1) are common database constraints:

- Setting `primary_key` to `True` makes a column the schema's primary key (rather than adding an auto-generated one).
- Setting `null` to `True` makes a column nullable.
- Setting `auto_now_add` to `True` for a timestamp automatically populates it when inserting tuples.

- Setting `db_index` to `True` adds an index for that column to speed up matches. Note that, by default, all geometric fields are spatially indexed.
- On foreign keys and other relations, the class type argument indicates the class or table that the model references.

Other field options exist for other purposes, which are described fully in Django's modeling documentation [10]. Note that, to ensure all database entries are fully and properly accounted for, Django adds an auto-incrementing `id` field to each model automatically, which acts as the primary key (except where we specify UID primary and foreign keys). The complete resulting GeoDjango SQL is available in Figure 4.2.

```
CREATE TABLE "rapid_feature" (
    "uid" text NOT NULL PRIMARY KEY,
    "geom" geometry(GEOMETRY,4326),
    "bbox" geometry(POLYGON,4326),
    "properties" text,
    "create_timestamp" time,
    "layer_id" integer REFERENCES "rapid_datalayer" ("uid")
      DEFERRABLE INITIALLY DEFERRED,
    "hash" text,
    "modified_timestamp" time
);

CREATE INDEX "rapid_feature_uid_like"
  ON "rapid_feature" ("uid" text_pattern_ops);
CREATE INDEX "rapid_feature_geom_id"
  ON "rapid_feature" USING GIST ("geom");
CREATE INDEX "rapid_feature_bbox_id"
  ON "rapid_feature" USING GIST ("bbox");
CREATE INDEX "rapid_feature_create_timestamp"
  ON "rapid_feature" ("create_timestamp");
CREATE INDEX "rapid_feature_layer_id"
  ON "rapid_feature" ("layer_id");
CREATE INDEX "rapid_feature_hash_like"
  ON "rapid_feature" ("hash" text_pattern_ops);
```

Figure 4.2: `CREATE TABLE` statements generated by Django using the Python `Feature` class.

Our other classes defined in models.py—our whole data model—count on that same functionality to generate tables.

Database interaction

To create and store a class instance—a row in the table—one can write the code in Figure 4.3 (we use a mock Feature as an example). Creating an in-memory object is equivalent to using any Python constructor (note the optional and named parameters compared to Feature's definition). The `save()` method, to save the instance to our database, creates and executes a corresponding `INSERT` SQL statement.

```
new_feature = Feature(uid=12345,
  geom='POINT(0 0)',
  properties='{ "Description": "Test Feature" }')

new_feature.save()
```

Figure 4.3: Django syntax for instantiating and saving a Feature.

Database queries mimick powerful `WHERE` clauses and joins in SQL. The static `filter` method on Django models returns a collection of selected objects, subject to the chosen parameters. See Figure 4.4.

```
features = Feature.filter(uid=12345)
for feature in features:
    foo(feature)
```

Figure 4.4: Django syntax for querying Features using the `filter` method.

Multiple `filter` calls can be chained to continue modifying the result set.[3] The lookup in Figure 4.5 finds Features with UIDs greater than 10000 (note the `gt` keyword) and `properties` strings containing "Test".[4]

Inter-module communication

Having described Django's unique mechanisms and advantages, we now show our approach to passing messages between RAPID's persistence layer and the API mod-

[3]The `filter` method returns a QuerySet object that, generally, does not access the database until its results are evaluated. This means that, over time, filters can be changed on a result set. Then, entries are queried as efficiently as possible *only when needed*.

[4]All the database operations in Django are much more complex than we can cover here. Refer to the official Django documentation for in-depth developer information [7].

```
features = Feature.filter(uid__gt=10000)
features = features.filter(properties__contains='Test')
for feature in features:
    foo(feature)
```

Figure 4.5: Django syntax for instantiating and saving a Feature.

ules. Francis organizes a `urls.py` file and a `views.py` file to parse incoming customer requests and generate suitable responses. Together, these define—precisely—how customers interact with our outside-facing REST API. Those responsibilities largely include extracting and transforming values from JSON application requests (as well as creating sensible outputs).

For that code to retrieve and use database entries, Francis and I created an additional file—the intermediary `select.py`—to largely handle data management, utilizing the technical concepts previously outlined in this subsection. We map high-level API business logic to this file (like adding DataLayers to a GeoView) and also add four sets of functions for us to easily add, retrieve, update, and delete customer-facing data: `create`, `get`, `update` and `delete`. We name these `create_feature`, `get_feature`, `update_feature`, `delete_feature`, and so on with DataLayers and GeoViews.

- The `create` functions take in end users' parameters, perform any additional back-end processing and store newly-created objects, returning their assigned UID to a caller.
- The `update` functions perform the same logic as creating an object, except specified parameters are modified for a chosen UID, not assigned to a new one.
- The `get` and `delete` functions take in a UID and fetch or delete (respectively) any corresponding database entry.

4.1.3 Chapter example

To aid the remainder of this discussion, we introduce a small (but reasonable) example of RAPID in use. Consider a scenario with two pipeline operating companies that

seek to *a*) import and modify public *and* private data and *b*) create, share, and query GeoViews.

We imagine ABC Pipeline Co. and XYZ Operations both operate within San Luis Obispo (SLO) County, California, overseeing two pipelines: one in Paso Robles and one in Pismo Beach (note the county map in Figure 4.6 and the city maps in Figures 4.7 and 4.8). The two companies cooperatively monitor and manage the section of pipeline in Pismo Beach, and XYZ Operations exclusively operates the section of pipeline in Paso Robles. For pipeline integrity management, each organization gathers and processes relevant datasets, intending to add them to RAPID (see Table 4.1). To easily monitor the data, ABC and XYZ create GeoViews for the two regions of interest.

Organization	Data	Covered region	Format
ABC Pipeline Co.	Traffic densities	SLO County	Shapefiles
	Construction equipment	Pismo Beach pipeline	GeoJSON
XYZ Operations	Rainfall	SLO County	GeoJSON
	Ground movement	Paso Robles pipeline	Shapefiles

Table 4.1: Example pipeline operator datasets.

The data from both organizations uses our permissioning system to allow shared, privileged access, as required by the business policies described below. Figure 4.9 diagrams *a*) simple relationships between DataLayers and GeoViews and *b*) the corresponding permissions setup for ABC Pipeline Co., XYZ Operations, and other RAPID users.

- As XYZ operates the Paso Robles pipeline on their own, they do not share the GeoView. The ground movement DataLayer is local and proprietary; they are sole Owner and Viewer. However, XYZ releases their rainfall DataLayer publicly, so that any RAPID user can view it.
- Because ABC Pipeline Co. and XYZ Operations cooperate in the Pismo Beach pipeline region, ABC shares their relevant resources with XYZ: *a*) the county

Figure 4.6: San Luis Obispo County lines. Note Paso Robles in the north. Pismo Beach (unlabeled) is between Avila Beach and Arroyo Grande.

traffic DataLayer, *b*) the Pismo Beach construction equipment DataLayer, and *c*) the Pismo Beach GeoView.

To slightly increase the complexity and realism of this example, we specify that the input formats begin as both GeoJSON and Shapefiles, and although ABC and XYZ review the same data for the same pipeline region, the groups use different DSS: ABC's only reads GeoJSON, and XYZ's only reads Shapefiles.

At a high level, with the above considerations, this scenario entails *a*) setting up API tokens, *b*) creating DataLayers, *c*) importing GeoJSON and Shapefile features into DataLayers, *d*) creating and querying GeoViews, *e*) modifying permissions, *f*) exporting data in GeoJSON and Shapefiles. Again note these are not always specific tasks for users—Francis describes those—but these end up being the most critical activities for our data model, described in the remainder of this chaper [16]. In other words, all of these can be accomplished by utilizing our models and writing queries with the techniques described in Section 4.1.2, and following sections discuss specific class designs and source code that handle geospatial organization and querying.

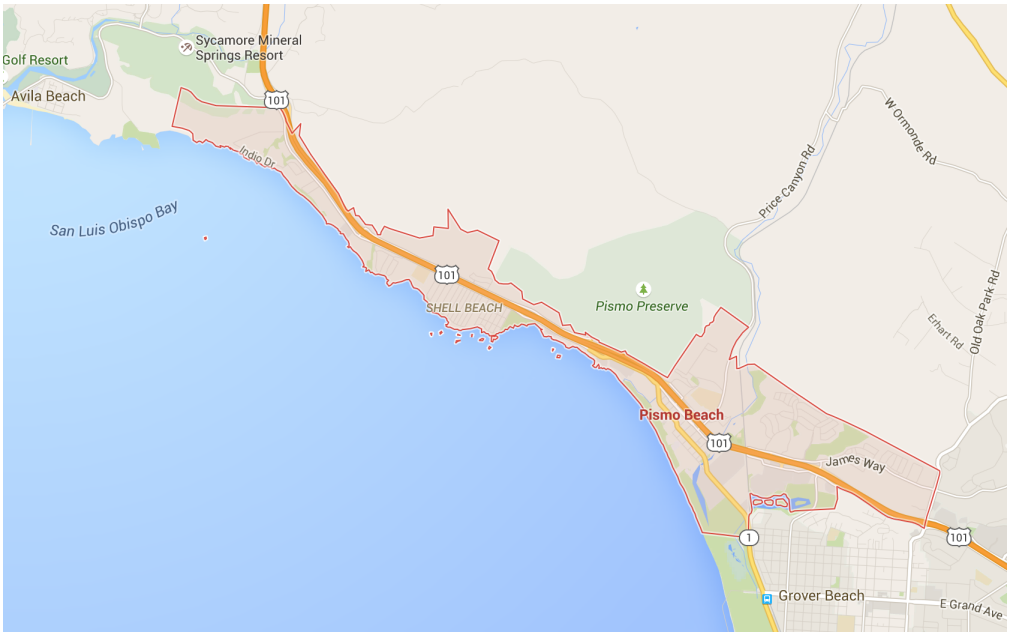


Figure 4.7: Pismo Beach city limits and pipeline region.

4.2 Geospatial data modeling and organization

Taken together, a small subset of RAPID classes enables structured data storage and retrieval for geographic features. In other words, these components could stand on their own, letting users browse and analyze geospatial objects (and groups of geospatial objects)—which fulfills our central database requirements. For the moment, we hold off discussing secondary functionality like permissioning, importing, and exporting data.

Figures 4.1, 4.11, and 4.12 show relevant portions of our `models.py` file for the three models we discuss throughout this paper. Figure 4.10, shows a simple entity-relationship diagram to clarify the setup.

4.2.1 Feature

We first describe the makeup of *features* in RAPID—that is, real geographic entities combined with other noteworthy indicators—and how they are presented to third parties. We saw in Chapter 2 that Geometry is the most generic and encompassing

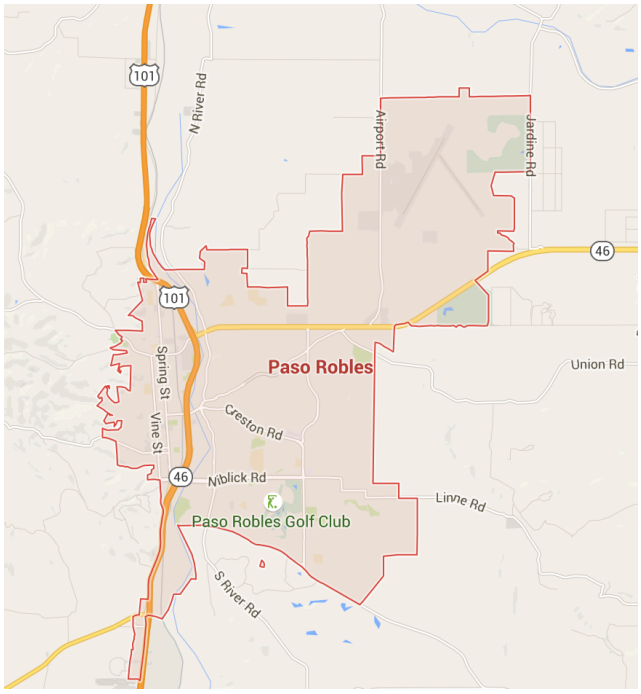


Figure 4.8: Paso Robles city limits and pipeline region.

geospatial data type.[5] We also discussed that geospatial features correlate those mathematically-defined geometries with other descriptive attributes.

A RAPID Feature object is mostly a recreation of the geospatial features described in the Abstract Specification. There are some minor system-specific additions, but the purposes are similar enough that we borrow the terminology. The Feature model comprises the following (with actual source code in Figure 4.1):

Geometry. A Geometry object represents the real-world footprint of the Feature (whether it is a Polygon, Line, Point, etc.). Functions from the Simple Features Access can also be used to highlight various spatial characteristics like length, area, or circumference—calculations and descriptions people use everyday [6]. Comparing two or more geometries (with the earlier-discussed relational operators) can reveal useful patterns, too.

For simplicity, RAPID does not support elevations in a geometry. When importing Features, we only store the first two dimensions. This is usually fine for our

[5]Remember that the Abstract Specification models these types hierarchically: a Polygon is a Polygon with holes, and a Polygon with holes is a Geometry, and so forth.

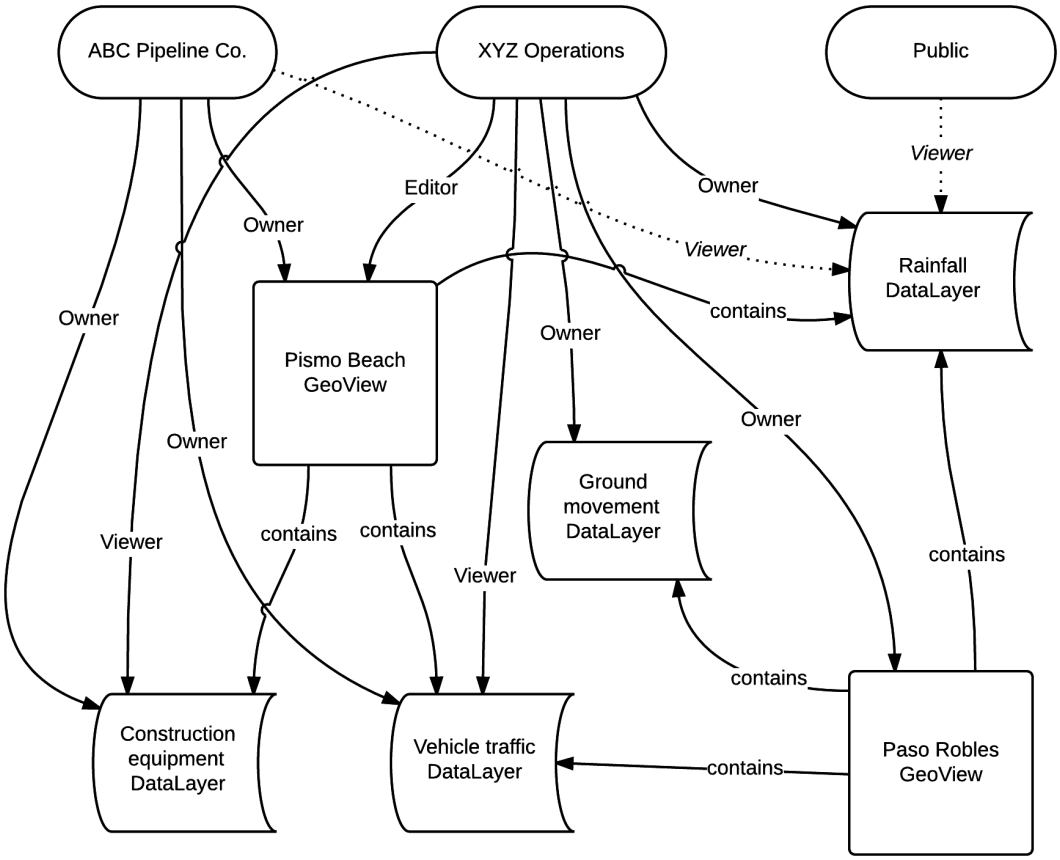


Figure 4.9: Example permissions setup between two organizations.

pipeline operating partners: most relevant datasets are two-dimensional—the features are located on the Earth's surface (or at least within tens of feet).

This missing third dimension is not related to the querying process (we never planned to allow Feature filtering based on height); it would only matter in data visualization or other analyses. In fact, three-dimensional geometry support is still rare in spatial DBMS (PostGIS is one of the only ones to actually support it). As an easy workaround on the user's end, the elevation could be ported rather easily to the Feature's properties, instead of residing in a Geometry.

Properties. Structured key-value stores are the norm for handling arbitrary GIS data, so an unlimited-length properties string bundles other relevant fields in a Feature. When shaping RAPID's usage style and conventions, we specifically

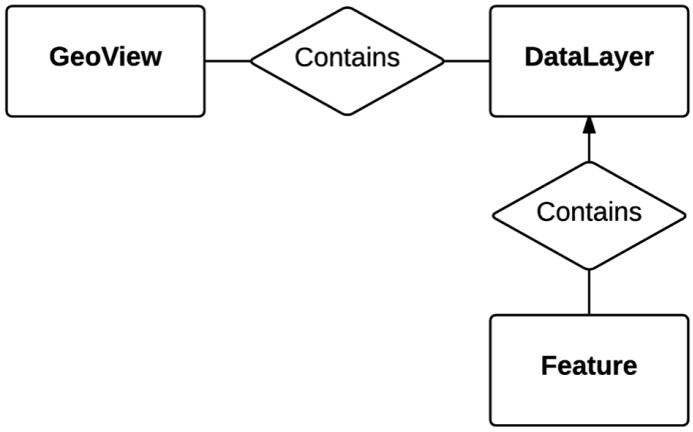


Figure 4.10: Entity-relationship diagram for RAPID's three primary geospatial classes.

```
class DataLayer(models.Model):
    uid = models.TextField(primary\_key=True)
    descriptor = models.TextField()
    properties = models.TextField(null=True)
    is_public = models.BooleanField(default=False)
```

Figure 4.11: DataLayer class in models.py

wanted to support JSON as an intuitive data interchange format because our implementation leans on GeoJSON (and the REST API already parses JSON requests).

The `properties` field introduces the ability to store and retrieve virtually any data for the Earth's surface (assuming it is serializable to JSON). Users that need extra functionality client-side can perform more advanced filtering and analysis with information besides just shape and location.

Unique identifier (UID). We use a public, user-facing unique identifier (UID) for direct Feature lookups. While these could be any kind of unique value type, RAPID automatically generates textual UIDs upon Feature creation that are URL-safe and user-friendly (in that they are relatively short, using everyday

```
class GeoView(models.Model):
    uid = models.TextField(primary\_key=True)
    descriptor = models.TextField()
    geom = models.GeometryField(null=True)
    bbox = models.PolygonField(null=True)
    properties = models.TextField(null=True)
    layers = models.ManyToManyField('DataLayer', null=True)
```

Figure 4.12: GeoView class in models.py

numbers and letters).[6,7] Because Features will often be retrieved using their UID, we add a database index for this field.

It is worth noting that our UIDs are generated randomly (with a secure random number generator) and end up being 22 bytes. An integer identifier would be reasonable too (especially with some slight indexing efficiencies over a longer string), but random strings won out:

- Features are the most numerous object type in RAPID. While we would not expect to easily max out a 32- or 64-bit number, 22 bytes erases any worry of integer overflow when deployed over a wide area and many years.[8]
- We do not want identifiers to indicate ordering, magnitude, or chronology for Features: the UID should look more like a hash, as proximate integers could otherwise imply a relevant or analyzable relationship.
- Following that logic, a random string is helpful from a security standpoint. One organization could theoretically share UIDs for all their classified Features in the open, but no one else would be able determine how new, old, or similar they are through the UID. An incrementing counter could otherwise hint at those characteristics.

Bounding box. It is somewhat common for GIS features to include a bounding box attribute—defining the minimally-bounding rectangle for the geometry. Recall

[6]We add UIDs to the three most prominent queryable models in RAPID—Feature, DataLayer, and GeoView—expecting that these are the objects under study and discussion. In that respect, the portability's very helpful.

[7]We coined this particular UID terminology for RAPID. An "ID" already gets used internally for back-end database work, and "UUID" (universally-unique identifier) is an existing standard for generating object identifiers [19]. Our UID makes use of the UUID standard behind the scenes, but they are not one and the same.

[8]The same argument could be made with fewer bytes: the library we use happens to use 22.

our discussion from Chapter 2: the small number of coordinates in a bounding box can estimate and rule out results in spatial functions more quickly than the "true" geometry.

RAPID does not currently use bounding box checks (but they could be useful in the future, pending performance tests). However, bounding boxes are an important GIS topic in Simple Features Access, so we store them anyway [6].

Timestamp. RAPID stores timestamps with Features—a filterable attribute when querying GeoViews—to indicate their relevance in time. This makes Features *geospatiotemporal*.

In a technical regard, `timestamp` is unremarkable, as it is simply another attribute to filter (as a range) when querying Features. However, as discussed in Chapter 2, trend-watching is particularly useful and needed in GIS work. This timestamp lets pipeline operators and DSS find data that *a*) is most recent and applicable or *b*) relevant for historical analyses.

DataLayer foreign key. A foreign key references the Feature's DataLayer—the DataLayers's UID. Note that this is a many-to-one relationship (and the primary reason the relationship is directed from Feature to DataLayer). In other words, a Feature always has access to its DataLayer.

Implementation note. This one-way relationship, however, does not preclude us from traversing the relationship in the other direction—seeing all the Features for a DataLayer. That alternative action is common enough, and Django makes the querying easy:

```
features = layer.feature_set
```

Even though a `feature_set` is not explicitly defined for a DataLayer, Django adds the property implicitly and handles the reverse lookup of the primary keys behind the scenes.

4.2.2 DataLayer

DataLayers exist to usefully store multiple similarly-structured Features. Although there is a notion of layering in OGC standards, RAPID diverges from any of their

detailed considerations and only keeps the spirit of categorization—hence the rename [6, 22, 34].

While RAPID does not enforce a consistent structure on properties, we would expect them to be similar within a DataLayer so that they can be analyzed consistently. Although a misnomer, a DataLayer's Features might be imagined as *instances* of the layer descriptor (see below), if they have similar schemas.

DataLayers include these fields:

UID. DataLayers use the same UID construction as Features. While the larger string size is not as necessary for the number of DataLayers as it is with Features—their count will never be the same order of magnitude—we keep the same setup for consistency. We could partially truncate the UID for DataLayers (and GeoViews) if we had to be extra concerned about it.

Descriptor. As a simple user-friendly and -facing title for DataLayers, we include a short Descriptor text field. The Descriptor labels the set of contained Features; "Earthquake" and "Construction equipment" are reasonable examples.

Properties. To mirror the properties capabilities in Features, we store JSON metadata in DataLayers (and leave it up to third parties to define the fields). Use of the metadata is optional, but it sometimes includes important documentation for Features' properties.

Public flag. When creating a DataLayer, we ask users to specify if it is a *public* DataLayer, which anyone is allowed to view. For DataLayers that should be widely-accessible, this removes the inconvenience of adding a lot of Viewers manually. The original creator retains ownership and can continue to add other Owners and Editors as necessary (described later in this chapter).

We have not carried out performance testing to determine the tradeoffs, but we specifically chose *not* to include bounding boxes on DataLayers, even though there *could* be significant efficiency improvements to explore: one check with a DataLayer bounding box could avoid costly comparisons against thousands (or millions!) of

Features. It would be somewhat more complex, however, to keep the bounding dimensions up to date: as soon as Features change within a DataLayer, the bounding box must be recalculated.[9]

4.2.3 GeoView

It is possible (and reasonable) to retrieve all Features for a given DataLayer, but a more common and interesting use case lets users narrow the scope to a more precise and helpful dataset. This is shown explicitly with GeoViews—a way to retrieve data from combined DataLayers for a particular area. We, in fact, mostly expect applications to query GeoViews during daily operation—not DataLayers or Features—because they encompass a more complete study-able concept.

To elaborate, a GeoView includes *a*) one or more DataLayers and *b*) a Geometry for a region of interest. RAPID saves which DataLayers are added to the view; when the GeoView is queried, Features from those DataLayers are returned to the requestor if they encroach on the chosen Geometry. GeoViews are rather analogous to ordinary SQL views (bringing about the name). They could even be implemented as them (aside from metadata), if so chosen.

To accommodate this functionality in the API, we define the GeoView model with these attributes:

DataLayer collection. Each GeoView has a collection of the DataLayers to retrieve Features from. Users can add DataLayers to GeoViews (and remove them), so at run-time, their chosen DataLayers are searched.

Geometry. To indicate a specific physical location to query within or around, GeoViews include an assignable Geometry field. Early in requirements gathering, it was a forgone conclusion that GeoView Geometries narrow DataLayers to their specified region. That implied using the `Intersects` boolean relational operator so that all Features within or encroaching on a region are available. On the other hand, as RAPID's implementation progressed and API calls were tested, we

[9]The system design could have been set up to incorporate this, and there is an obvious place in the implementation to include the code in `select.py`, but it is outside our scope of work.

noticed the value in supporting several of the Abstract Specification's spatial operators.

As such, `Intersects` is left as the default, but we also let callers to this interface choose `Contains` or `DWithin`. `Contains` produces of subset of the results of `Intersects`; the difference is that "contained" Features are *fully* contained within the Geometry and cannot merely be overlapping the border. The `DWithin` operator produces a superset of Features from the `Intersects` results; the difference is Features within a certain distance of the Geometry are also selected. `DWithin` could highlight *nearby* Features that are still relevant (or that may be relevant in the future).

Bounding box. Again, pending further performance testing, GeoView bounding boxes could prove to better our speeds, but they are not currently set or used in our queries. We do not have the same concerns with bounding box consistency as we do with DataLayers, however, because there is only one geometry associated with the object.

UID. This identifier for GeoView rounds out our use of UIDs, included for the same reasons discussed in Features and DataLayers.

Descriptor. It makes sense for GeoView instances to have a user-friendly name to encompass their purpose—possibly shuffled among other GeoViews. An example we saw would include "Pismo Beach Pipeline."

Properties. Included for the same reason in DataLayers, a properties field stores any desired metadata. In the case of GeoViews, users may want to provide detailed explanations on why the DataLayers are relevant or what different values indicate.

Implementation note. Django simply presents many-to-many relationships as collections in each model. Take the following example, where an instantiated DataLayer is added to a GeoView:

```
geoview.layers.add(layer)
```

Behind the scenes, Django creates and populates a junction table, mapping DataLayers to GeoViews (using their UID primary keys).

4.2.4 Example

Here, with a few example objects, we show how these models may be populated and stored for the large SLO County scenario. In Figure 4.13, let us create the Construction equipment DataLayer, import a Feature, and create the Pismo Beach GeoView, associating the GeoView and DataLayer.

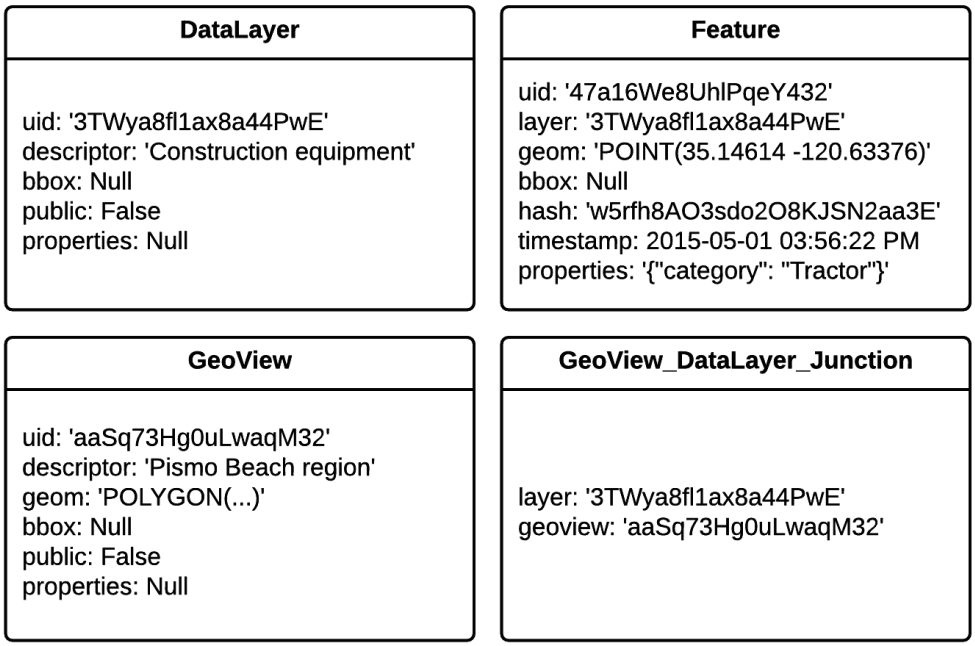


Figure 4.13: Several example instances of Features, DataLayers, and GeoViews (including the junction table entry for the many-to-many DataLayer-to-GeoView relationship).

4.3 Importing and exporting data

We can now describe how data is added to RAPID—how Features are "imported" from geospatial file formats—which might occur on-demand from API requests or at regular intervals automatically. We also take the opportunity to describe the export

process.[10] As we discussed in Chapter 4, RAPID must read from and write to a variety of geospatial file formats (GeoJSON and Shapefiles at first). Here, we note how a user specifies a file and file type, and the generic process RAPID uses to parse and save Features. Although our requirements do specify GeoJSON and Shapefile support, we know other parsing capabilities may be required soon. As such, we structured the objects and interfaces so that additional file formats will be easily importable at another time.[11]

4.3.1 Importing files

The multiplicity of spatial and geographic file specifications is mirrored by the number of options in parsing, writing, validating, and transforming them. Some tools are easier to use than others; some support files that others do not. Some spatial DBMS will even handle encoding and decoding of popular formats natively. Therefore, we prioritize flexibility in the architecture for importing files and set up a pattern and process for consistent conversion.

In an `import.py` file, we add high-level file retrieval and parsing functionality, letting the API designate a file type and file location to import. As currently implemented, RAPID can parse GeoJSON and Shapefiles from a URL or on-disk file path. Once the file content is available, it is fed into a parser.[12]

At a minimum, the parsing utilities and methods must extract geometries and key-value properties (as a dictionary) from the specified file content. With those available, they can be passed (along with the chosen DataLayer's UID) to our `create_feature` function to handle final processing and storage. Properties are converted to JSON; the geometry must end up in GeoJSON or WKT.[13]

[10] Note that this only goes as far as to say how Features become and are converted from geospatial formats. The REST API itself includes other metadata and formatting in its requests and responses; the serialized geospatial file may be embedded in a larger JSON object [16].

[11] In general, we looked to future-proof these high-level interfaces so that they are more dependent on the Abstract Specification than any one format or library.

[12] GeoJSON is an ordinary text-based file format, but as we discussed in Chapter 2, Shapefiles are made of up multiple files, so they are transferred and stored as `.zip` files. Before parsing the extracted files, RAPID unzips this file.

[13] Third-party documentation and tutorials provide the most accurate and proper guidelines for file parsing and validation. For example, GEOS provides spatial data structures and operations; python-geojson and PyShp parse and write GeoJSON and Shapefiles [8, 14, 18].

Because multiple files can be added to one DataLayer, duplicate Features could be encountered; however, we need to eliminate redundancies so that these are not interpreted as discrete entities. To address this, we add a hash value—a digest of the Feature—with the constraint that it is unique per DataLayer. In other words, new Features with the same geometry and properties as an existing Feature are disregarded.

4.3.2 Exporting Features

Exporting retrieved Features (maybe from queried GeoViews) works similarly to file importing in reverse, but it is primarily in the realm of Francis's API work [16]. To summarize, these are some of the critical steps and requirements:

When reading a GeometryField, Django boxes the value into a GEOSGeometry object, exposing helpful functions and properties code-side. Most importantly, these objects have a `geom_type` field and a `coords` list, specifying the Abstract Specification data type and its list of coordinate pairs. Using these components (and the fact that we prescribed SRID 4326), any utility can render serialized Abstract Specification geometries, whether they turn into GeoJSON, Shapefiles, or WKT (or any other format).

The only remaining consideration is how to incorporate Feature `properties`. When the JSON `properties` string is pulled from the database, it can be 1) used and transmitted as-is (the case for GeoJSON) or 2) turned into a Python dictionary for use in other serializers (like for Shapefiles).

4.4 Multi-tenancy and permission management

Chapter 3 laid out permissioning capabilities for RAPID DataLayers and GeoViews. That functionality is enabled by adding several components: the entity-relationship diagram in Figure 4.14 builds on the diagram in Figure 4.10, showing the additional classes that make this an operational multi-tenant system (note that the `Has permissions` relationship is encompassed in the GeoViewRole and DataLayerRole models described later).

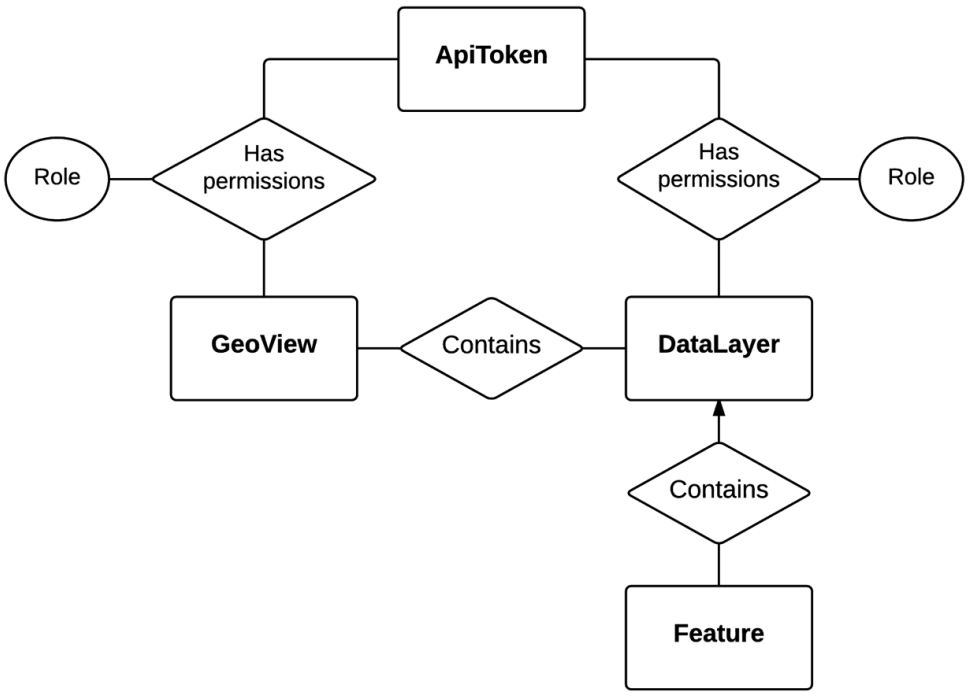


Figure 4.14: Entity-relationship diagram for RAPID's main conceptual data model, with permissioning capabilities.

4.4.1 Role enum

We add an enum, Role, to define the three available permission roles for RAPID data: Owner, Editor, and Viewer.

Again recall that, on a daily basis, in-operation software is RAPID's end user: *people* are not necessarily Owners, Editors, or Viewers of data. We instead distribute API tokens that have assignable roles on each DataLayer and GeoView, and any system using that token has a uniform view of RAPID's resources.

4.4.2 ApiToken

The simple ApiToken model associates a friendly name (used as an identifier for administrative purposes) with a cryptographically-secure, randomly-generated key that grants resource access. When API requests come *with* a token, RAPID knows who or what is requesting data and can return an appropriate set of results.

```
class ApiToken(models.Model):
    key = models.TextField(unique=True, db_index=True)
    descriptor = models.TextField(unique=True)
    issued = models.TimeField(null=True, auto_now_add=True)
```

Figure 4.15: ApiToken class in models.py.

Name. We require a friendly public name to identify this token. This can be used when choosing and assigning permissions for RAPID's other managed tokens.

Key. Upon creating an ApiToken and giving it a name, the system creates a long, random, URL-safe hash value to use as a password for sharing and accessing privileged data. When performing access-dependent API activities, the caller uses the (unguessable) token to show that they have the correct permissions, and RAPID checks the key against its internal records (the DataLayerRole and GeoViewRoles below).[14]

Timestamp. Although our work is not focused on the security aspects of RAPID, we define and assign a creation timestamp for each token so that expiration or renewal policies can be implemented by future contributors.

To track newly-assigned permissions on DataLayers and GeoViews, we create DataLayerRole and GeoViewRole models—associating an ApiToken with a DataLayer or GeoView and the particular role.

```
class GeoViewRole(models.Model):
    token = models.ForeignKey(ApiToken)
    role = enum.EnumField(Role)
    geo_view = models.ForeignKey(GeoView)
```

Figure 4.16: GeoViewRole class in models.py (analagous to DataLayerRoles for DataLayers).

[14]While this setup is an accepted basis for web-based API security, RAPID should see an audit before storing truly classified data. For example, any other security measures are for naught if HTTPS is not enabled on the production server [26, 31].

4.4.3 DataLayerRole and GeoViewRole

Both DataLayerRoles and GeoViewRoles contain the same fields (except for referencing DataLayers versus GeoViews). See Figure 4.16.

ApiToken foreign key. Specifies the ID of the ApiToken that has resource access.

DataLayer or GeoView foreign key. Specifies the ID of the DataLayer (if a DataLayerRole) or GeoView (if a GeoViewRole) to assign permissions to.

Role enum. The Role enum specifies the level of access this setup grants for the token on the chosen object.

4.4.4 Example

Again extending our earlier example, Figure 4.17 shows the specific object instances that support multi-tenancy and data sharing. Specifically, we create two API tokens and make ABC Pipeline Co. an Owner of the Pismo Beach GeoView and Construction equipment DataLayer; XYZ Operations becomes an Editor for the GeoView and a Viewer of the DataLayer (the GeoView and DataLayer are seen in Figure 4.13).

4.4.5 Race conditions discussion

The RAPID model and API enable and account for multi-tenancy; however, behavior can become inconsistent during and across database changes, particularly when fetching and modifying individual Features. For example, new data could be imported into RAPID while already iterating through a list of a GeoView's Features; or a token's permissions could change after it reads some DataLayers of a GeoView, but not all the ones it originally had access to. In other words, operations and sessions are not currently atomic or transactional.

This inconsistent behavior could be improved with one or both of the following strategies:

- Including a "last modified" timestamp on GeoViews, DataLayers, and Features would provide additional information for requestors to evaluate data recentness

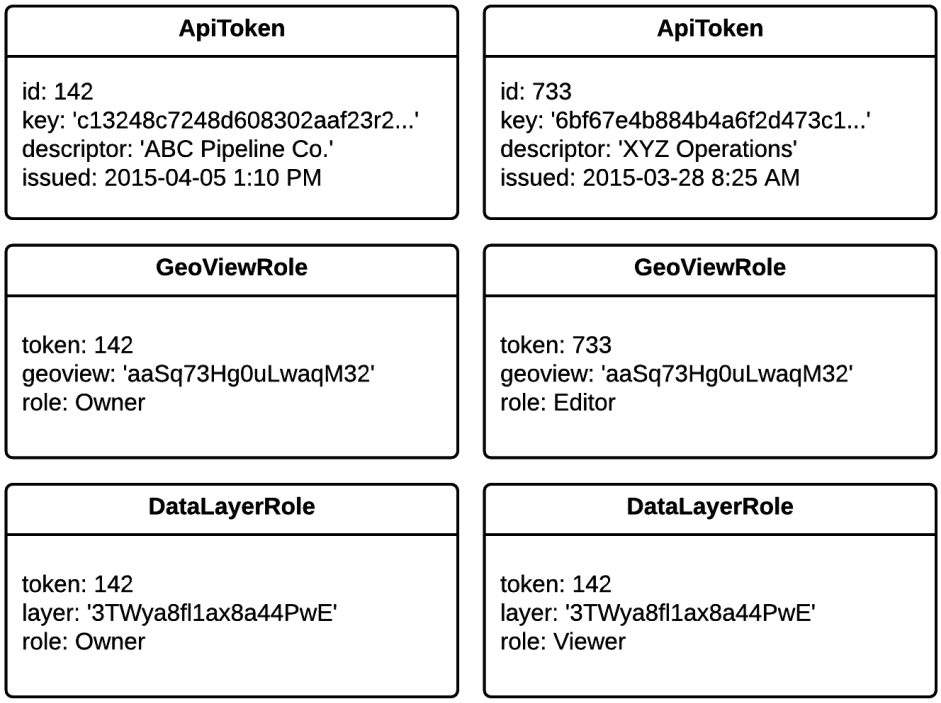


Figure 4.17: Several example object instances supporting multi-tenancy.

and relevance. A "last imported" timestamp on DataLayers would be worthwhile here, too. For operations where atomicity is important, the external application can track and compare system times and object timestamps. This approximates a transaction or import ID added to the most-recently-changed objects.

This process might be simplified for third-party developers (but made more complicated in RAPID) by implementing a ticketing or batch-update system with reviewable queues and changelogs (this could involve performance concerns or business policy changes). In this way, RAPID could manage multiple high-level operations at once and without conflict, and third parties could monitor progress and specific changes.

- Using a combination of database transactions and custom locking mechanisms, the API could be restructured such that third party conflicts are avoided. That is, a design decision should be made and documented for reading uncommitted or in-progress changes, and policies must be developed to request (and release) locks on the different objects and collections.

Chapter 5

VALIDATION

In this last chapter, we summarize our validation of RAPID, showing that the system meets Chapter 3's critical requirements, correctly uses the concepts and technologies outlined in our Background, and is efficient enough for real use in the future.

We made two primary contributions: 1) guidelines and findings for API developers—at this point, Francis—validating our data model and overall system design and 2) performance testing results for common operations.

5.1 Functionality

RAPID includes a geospatial data model as well as file parsing and querying capabilities. We must, therefore, verify that RAPID's design meets its specification and that our codebase parses, stores, queries, and outputs geospatial data correctly. These are particularly prudent, as RAPID's entire purpose is to enable portability and interchangeability for *external* partners and *future* developers.

Our example setup between ABC Pipeline Co. and XYZ Operations is a relatively complete case study, showing real class instances and relationships. By showing the models and implementation patterns in Chapter 4 that support that needed functionality, it should be apparent that RAPID completely and successfully meets our system requirements from Chapter 3.

Even though our model is, hopefully, convincing on its own, we worked with another student developer, Kishan Patel, to integrate a web application with RAPID for further validation. We call this software *RAPID UI*, which visualizes spatial entities and our organization of data (Features in DataLayers, DataLayers grouped into GeoViews).

Our peer developer, Patel, had to become acquainted with basic geospatial concepts and client-side libraries, but our contributions to RAPID were significant and successful enough that he was able to create a sensible UI using our most important

API calls. Second, being able to visualize our data—both geographic geometries and accompanying properties—we literally see that data makes it through RAPID, from beginning to end, as expected. As an additional nicety, we share and visualize several datasets with RAPID UI that were recently recommended by pipeline operating partners.

Francis describes our work with Patel in-depth—she worked with him to design and fully test REST calls—but please see the short following summary, which outlines our expectations and findings for RAPID UI (focusing on the data model's role) [16]. A screenshot of the interface is shown in Figure 5.1.

- A primary goal in researching GIS standards and technologies was to ensure RAPID operates as expected when handling (rather finicky) geospatial data. Therefore, we set up several sample DataLayers (from GeoJSON and Shapefiles) for RAPID UI to attempt retrieving and visualizing. In particular, we had the goal of requiring as little custom UI logic as possible—relying instead upon pre-built GIS tools.

  Patel and Francis ensure RAPID UI could properly request data and handle GeoJSON responses through the API (using common HTTP libraries in Python and JavaScript). After RAPID UI fetches standardized files, using particular geospatial libraries *should and does* let the application parse and visualize our data in one step.

- Aside from visualizing correctly-shaped and -placed geospatial Features, we had to count on RAPID UI retrieving and displaying relationships between GeoViews, DataLayers, and Features. Patel, using RAPID's JSON responses (designed by Francis), displays available GeoViews in RAPID UI, letting users switch between them and their DataLayers. That is, users can observe and analyze whichever datasets they need at that time.

- Last, RAPID UI incorporates RAPID's geospatiotemporality by filtering data by time. For example, users can switch between days of a single month to see which Features are generated or applicable on which days.

  This characteristic is demonstrably useful: out-of-date Features can be imported alongside recent data that is of much more interest. Operators fre-

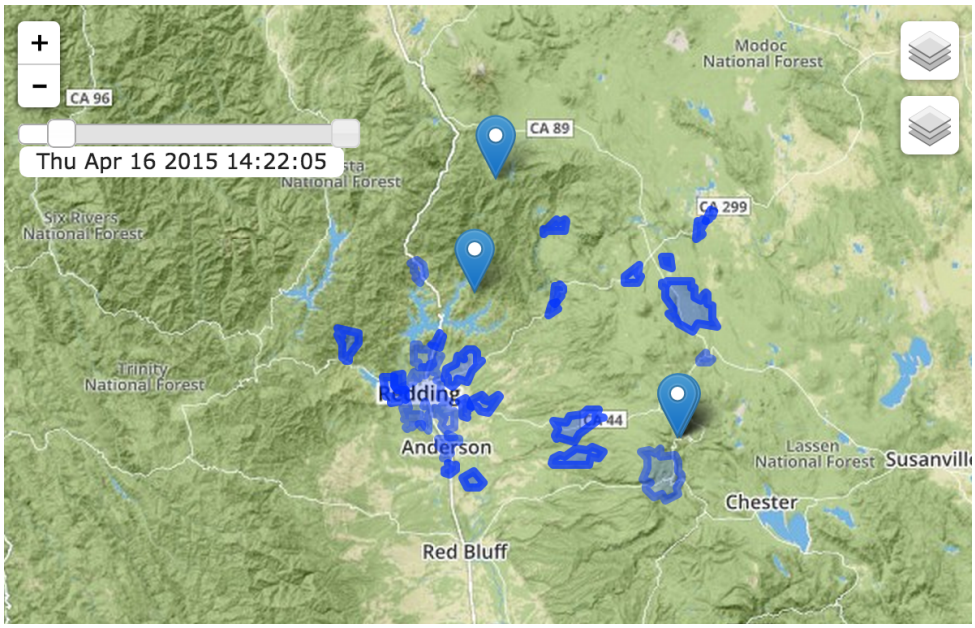


Figure 5.1: RAPID UI screenshot showing two DataLayers' Features in a Shasta County GeoView.

quently value hazard and encroachment *prevention*, using the most up-to-date information [11].

5.2 Performance

The previous section shows that, given our data model and business logic, RAPID operates properly; we now use this section to highlight our core data model's performance, showing that it is an efficient foundation for future additions. We point out specific interface actions that typify DSS work and compare their runtimes.[1]

Spatial Feature querying is RAPID's main bottleneck. During region-specific lookups (like with GeoViews), the system must filter each Feature in a DataLayer based on spatial calculations. Automatic spatial indexing in PostGIS amortizes this, but the process is still more demanding than `SELECT` statements using one indexed unique identifier.[2]

[1]Performance testing spatial schemas and source code could be a thesis to itself. Here, we cast a wide net, but still tackle our most visible and potentially-taxing operations.

[2]GeoDjango (by default) creates *bounding box* indexes in PostGIS with R-Trees (see the `CREATE TABLE` and `CREATE INDEX` statements in Figure 4.2).

Dataset	Size	Best	Middle	Worst	Average
Points	Small	0.003	0.003	0.003	0.003
	Medium	0.007	0.007	0.008	0.007
	Large	0.055	0.056	0.057	0.056
Polygons	Small	0.002	0.002	0.002	0.002
	Medium	0.012	0.012	0.012	0.012
	Large	0.393	0.395	0.398	0.395

Table 5.1: Performance testing results (measured in seconds).

To study RAPID's spatial querying, we run several performance tests. One high-reaching goal for the system's future is to manage and fetch data for a whole continent; however, we just as likely expect some customers to work with smaller numbers of Features. Therefore, we test three dataset sizes, each separated by a multiplication factor of 100 Features: small (10), medium (1,000), and large (10,000).

Because the number of coordinates pairs can be orders of magnitudes different, we test spatial filtering on both Polygons and Points. Our Point dataset uses the previously-discussed USGS earthquake data—spanning the entire globe—and we specify Shasta County (in northern California) as a region of interest for filtering. Although we also use Shasta County to cull our Polygon dataset, we use United States cities for the Feature geometries. We perform each query three times and average the runtimes. See the results in Table 5.1.

Two characteristics are especially notable: 1) taken as a whole, spatial queries are currently efficient enough in RAPID, and 2) spatial queries scale well—not showing exponential growth, as one might expect. PostGIS indexes and `WITHIN` queries are constructed such that, even with hundreds of thousands of table rows (and, for polygon geometries, encoded data for tens of millions of data points), GeoView results can be evaluated in under half a second on commodity hardware.[3] In addition, increasing the number of Features in a DataLayer by two orders of magnitude only shows, at most, a thirty-three times slowdown. In general, each leap in size only adds fractions

[3]These test results come from running the test queries on our development server—a virtual machine with 1 GB RAM and a 2.5 GHz CPU.

of a second to the whole query—less than a ten times slowdown.

5.3 Unit testing discussion

The following recommendations could be used to further validate RAPID with a thorough unit test suite.[4] Once each required field for geospatial features is located for each possible input and output, exhaustive unit tests can ensure that fields in an input format can be converted to the corresponding fields in any output format. In the event storage or conversion fails, unit tests for the following components would reveal where and why:

- Writable and parseable digital format like GeoJSON or Shapefile.
- In-memory objects (our own models or GEOSGeometry objects[5]).
- WKT or GeoJSON—abstract text formats for geospatial features.
- Abstract data persisted in PostGIS.

Data can move up and down that list; each step and transition (all outlined in Chapter 4), should be considered a data integrity checkpoint that deserves a unit test. For instance, a Polygon made up of coordinate pairs (0, 0), (0, 10), and (10, 0) still needs to have the same meaning—a triangle with those three points—no matter its format or original location.

GeoJSON, Shapefiles, Python objects, WKT, and PostGIS data structures can all be inspected to see that the same data is contained in each. If not, it was misshaped somewhere along the line—traceable with these "checkpoints."

[4] On the back end, RAPID could use the Python's built-in testing module, `unittest` [9]
[5] We discussed GEOSGeometry briefly in Section 4.3.2.

Chapter 6

CLOSING DISCUSSION

Largely concluding our discussion on RAPID, we review our initial mandates and subsequent approach to creating and validating the system. Last, we describe future work that can directly build on our contributions.

6.1 Project summary

Over the course of this document, we introduce novel and complex GIS topics, showing how geographic datasets are standardized, stored, and queried. We, additionally, explain Django's advantages for designing and programming a spatial, multi-tenant data store. We discuss the high-level requirements we created and were assigned—with attention paid to pipeline-operating colleagues, non-expert web developers, and OGC standards—and how RAPID was built to satisfy them: our data model, coupled with Francis's public-facing API, enables the expected functionality. Validating our design and implementation, we 1) show another developer's accomplishments made possible by RAPID and 2) illustrate efficient performance in everyday situations.

6.2 Future work

There are high aspirations for the data that RAPID collects and serves: letting artificial intelligence plan pipeline routes and sound urgent alarms would be a boon to integrity management processes [11]. Even without advanced algorithm development from other parties, by gathering and merging these data sets—always subject to business policies—we have set the stage for unique and useful DSS analyses. The following list sums up future work that we recommend (or already anticipate occurring) to upgrade the entire RAPID system of geospatial data aggregation and analysis.

- We previously discuss that RAPID Features include a timestamp, allowing DataLayers to be queried geospatiotemporally. As a stop-gap, this timestamp

is populated when the Feature is first added to the system, setting it to the current system time. However, timestamps are not a standardized component of Features in the Abstract Specification or Simple Features Access; therefore, we cannot count on a set of abstract steps to extract a relevant date and time [6].

As a data mining exercise, Feature properties could be searched to extract custom timestamps. Alternatively (and with more API design), users could specify specific property fields that contain timestamps in a known format.

- While GeoJSON and Shapefiles are popular—and many other GIS formats can be converted to them for use in RAPID—the more formats RAPID supports natively, the better. Additional parsers and generators could be added to improve the import capabilities' user-friendliness. As we mentioned in Chapter 4, this is not terribly difficult, as the most critical RAPID logic is already encapsulated in our `select.py` methods.

- Logic is already present for retrieving GeoJSON and Shapefiles from URLs and local files, but *not* automatically over time. This is primarily because it requires relatively-complicated scheduling utilities (as well as web server support). Future developers would define the API mechanics for this too—that is, how users (or applications) specify regularly-fetched locations and at what intervals.

- We currently use commandline scripts to generate and (manually) distribute API tokens. This works for our initial small-scale offering, but for a large, active system with many users, a web-based user interface would make the process hands-off and much smoother.

  Furthermore, we would recommend adding functionality for email-associated user accounts that may generate (and expire) API tokens at-will.[1] Incorporating permission management tools—so that people can view and modify shared-tenant setups—would also move RAPID closer to commercial-readiness.

- Our testing and discussions for RAPID often revolve around public datasets, easily found on the Internet or available through local governments and re-

[1] It is reasonable to allow multiple tokens per user: a user could manage a RAPID account for a whole organization, but they may still think it important to define permissions per application. That is, one organization might use multiple RAPID-connected programs that require different permissions.

search bodies. We realize, though, that certain customer data could be so highly-protected that it needs to be encrypted at rest (not just hidden from unprivileged API tokens—we already account for that).

Although certain levels of encryption are possible, there are significant tradeoffs and performance implications: one could encrypt only geometry data or both geometries and properties.[2]

  - By encrypting a Feature's properties, information *about* a place would be confidential, but the place itself is not encrypted. RAPID querying would work mostly as it does currently, but those attributes would have to be decrypted before use, adding extra processing time.
  - Encrypting geometric data of a Feature would hide *where* data occurs. Because encrypted geometries cannot be directly queried—the spatial information is scrambled—system performance would take a massive hit. In other words, PostGIS loses its ability to create spatial indexes; an entire data set must be decrypted before performing *any* spatial queries (not to mention nicely-indexed ones).

- Quite interestingly, our choice of JSON as a key-value property store could make RAPID querying significantly more powerful with just a PostgreSQL update: PostgreSQL 9.4—RAPID runs on 9.3—introduces the `jsonb` data type, allowing for SQL queries *within* a JSON field's key-value pairs. Rather than retrieving Features only by space and time (using GeoViews), third parties (with API changes) could look up and analyze Features by any arbitrary properties.

- When RAPID imports files, brand new data is (likely) available to the system—previously unavailable for analysis through our tools. Because there is new, potentially-important information to act on, pipeline operators would appreciate getting notified of certain occurrences or changes in RAPID. For example, a certain type of Feature (like construction equipment) in a particular region might be worthy of a warning. Specific properties, too, could be useful for on-the-fly analysis, notifying users of specific conditions in a location. For example,

[2]We limit this discussion to Feature encryption, but there may also be interest in encrypting DataLayer or GeoView metadata. That would have its own requirements and complexities, but it would be doable, and one does not have to consider geometry encryption on DataLayers.

a certain amount of rain or seismic activity in an area might require immediate investigation.

This functionality would tie in nicely with emails and user accounts, as well as JSON querying through PostgreSQL.

- When considering full OGC Compliance for RAPID (like the WFS standard), we had to balance its potential advantages with the resources it would take to correctly implement it. However, we ensured that RAPID is a *standard-enough* base for this future work. In other words, we worked hard to make Features standard and portable; moving toward a WFS implementation would overhaul the fashion and formats in which they are queried and transferred.

BIBLIOGRAPHY

[1] American Petroleum Institute. Improving Liquid Pipeline Safety, 2013.

[2] Basudeb Bhatta. Remote Sensing and GIS, 2011.

[3] Boundless. Introduction to PostGIS.

[4] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Tim Schaub, and Christopher Schmidt. The GeoJSON Format Specification, 2008.

[5] Steve Chastain. Pipeline Right of Way Encroachment: Exploring Emerging Technologies that Address the Problem. *Right of Way*, pages 22–27, 2009.

[6] OpenGIS Consortium and Others. OpenGIS simple features specification for SQL. *OpenGIS Project Document 99*, 49:49–99, 1999.

[7] Django Software Foundation. GeoDjango Database API.

[8] Django Software Foundation. GEOS API.

[9] Django Software Foundation. Writing and running tests.

[10] Django Software Foundation. Models, 2014.

[11] James Dunning. Improved Satellite and Geospatial Tools for Pipeline Operator Decision Support Systems Project Proposal. Technical report, California Polytechnic State University, 2013.

[12] ESRI. ESRI Shapefile Technical Description. Technical report, 1998.

[13] Esri. What is GIS? 2014.

[14] Corey Farwell. geojson Python Library, 2014.

[15] FeatureServer.org. FeatureServer, 2012.

[16] Alexa Francis. REST API to Access and Manage Geospatial Pipeline Integrity Data, 2015.

[17] C Kottman and C Reed. The OpenGIS abstract specification, topic 5: features. *Open GIS Consortium Inc*, D:56, 2009.

[18] Joel Lawhead. pyshp Python Library, 2014.

[19] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. Technical report, Internet Engineering Task Force, 2005.

[20] Inc. Michael Baker Jr. Pipeline Corrosion. Technical report, U.S. Department of Transportation, 2008.

[21] National Imagery and Mapping Agency. Department of Defense World Geodetic System 1984. Technical report, 2000.

[22] Open Geospatial Consortium. FAQs - OGC Abstract Spec.

[23] Open Geospatial Consortium. Reference Implementations, 2007.

[24] Open Geospatial Consortium. OGC History. 2012.

[25] Open Geospatial Consortium. OGC GeoPackage, 2014.

[26] Chris Palmer and Yan Zhu. How to Deploy HTTPS Correctly, 2013.

[27] Kim Peterson. Development of Spatial Decision Support Systems for Residential Real Estate. *Journal of Housing Research*, 9(1):135–156, 1998.

[28] Pipeline and Hazardous Materials Safety Administration. Annual Report Mileage for Natural Gas Transmission & Gathering Systems. Technical report, 2014.

[29] Slashgeo. The Shapefile 2.0 Manifesto.

[30] Spatial Decision Support Knowledge Portal. Spatial Decision Support Systems, 2008.

[31] Stormpath. Secure Your REST API... The Right Way, 2013.

[32] Tim Sutton. A Gentle Introduction to GIS. 2009.

[33] United States Geological Survey. National Geospatial Program Standards and Specifications.

[34] Panagiotis Peter a Vretanos. Web Feature Service Implementation Specification. (May):1–131, 2005.