SandDB - Community Sand Database

Vasco Guilherme Brás Pinho (67726)
MEEC - Instituto Superior Técnico

A large number of geological samples is collected each year around the world which are usually stored in a way that prevents discoverability and collaborative research. With this thesis we present a better online tool to allow researchers, teachers, students and geology enthusiasts to store, share, analyze and collaborate on sedimentological data.

Current methods used by geologists for sediment sample collection are presented, as well as which tools currently exist to facilitate these processes and the problems we found with them. A set of requirements is identified in order to improve on existing tools, and the implementation of the SandDB.

We implemented a georeferenced database of sediment samples accessible by users and web services through a REST API and an online website. The user interface allows the easy management of information and new samples input is integrated with desktop spreadsheets. With this thesis we implement the basis for full fledged GIS system based on open-source web software. Our aim with this system is to empower users with tools for easy storage and sharing of sediment data.

This work demonstrates how it is possible to develop a GIS based system resorting only to widely available open-source web technologies, not requiring any specific GIS server programming knowledge.

Index Terms—Georeferenced databases, sediments, geological samples, online platforms, sediment database, REST API.

I. INTRODUCTION

SEDIMENTS have embedded in their physical and chemical properties answers to their origins and those of their environments[5]. Many scientific questions, policy issues and social problems can benefit from the answers provided by the analysis of sediments[7]. In Geology, this study is composed of several techniques to gather information on trends in the processes that regulate transportation and deposition of sediments[1]. A current problem is the lack of knowledge of what field work was already performed. Several researches even from different study areas (civil engineer[4] or Petroleum Engineering, Geochemistry, Hydrology [12]) will perform duplicated field and laboratory work in samples from the same location because there is no way to know samples were already collected and that the data exists elsewhere. For the data that exists, a non-trivial amount is stored in old fashioned paper format or local spreadsheets in a department computer[13], [6]. There is a very high cost involved in reproducing results since it is difficult to validate where samples were collected, where they are stored and exactly what tests have already been conducted. There is no way to create “big picture” research (either in terms of large timespans, large areas or both) unless a given research group incurs in a single concerted effort. Even in this case, there is no way for outside groups to participate and contribute to the effort (for instance it is hard for high school teacher to gather new and relevant information [3], [18]).

The main objectives of the work here presented can be summarized in:

• Facilitate the discoverability of data.
• Prevent the duplication of efforts regarding sample collection in the field.
• Create of global communities with shared research interests and facilitated collaboration opportunities.
• Facilitate the reproducibility of results by other researchers by:
  • Allow new research from combined data samples and patterns in large data volumes.
  • Allow new research from the availability of historical data samples.
  • Allow the education community, policymakers, the private sector and society in general to study, make decisions or take action based on the available information.

We aim to build a sediment database, publicly accessible through an online web application. Users will be able to add information to this public database. Universities with research samples, museums or enthusiasts with private collections will be able to store their data in this system in a more organized and easy to access way. Data will be openly accessible according to user defined permission models and users will also be able to search and visualize theirs and other users’ sets of data.

A. Document Outline

The rest of this document will present the technology stack that is needed to create an online accessible georeferenced database, a website and the advantages of each tool analyzed to accomplish what we propose (Sections II and III). Afterwards we present our actual solution describing the architecture, (Section IV), the software used, and data model designed and the actual development (Section V). Finally, in Section VII) we summarize the work performed and conclude with possible future work.

II. STATE OF THE ART

Sediments are usually collected in the field in order to bring them to a laboratory where further tests can be performed on the samples. As such, there are protocols that have to be
followed and the methods of collection have to be carefully reported[16]. Sometimes though, there are inconsistencies in the data. Existing samples might have been collected at an earlier date and no records can be found. They might also be collected by laymen enthusiasts and so this data may be absent. Afterwards, in the laboratory there are a number of tests that may be performed, or none at all. Tests might have different outputs based on the tools and methods used or its intended purpose. Finally, it is important to differentiate between a test run by a highly trained and expensively equipped research team and a weekend collector. It is easy to imagine that data is usually not consistent. This will be a recurring theme in this work, data is usually not uniform nor organized, and we have to try and make the most of it. Nonetheless, geologists already collect a lot of data and usually have standardized procedures to follow regarding what data should be kept. In the following sections we will describe what these data points are without being exhaustive, covering what we believe to be the most common subset of techniques and procedures, mostly from reference.

A. Open Geospatial Consortium

The Open Geospatial Consortium (opengeospatial.org) is a standards organization which develops, implements and maintains the specifications and standards for open geospatial tools. Many of the tools presented henceforth have originated from OGC.

1) Web Map Service

The Web Map Service (WMS)[11] allows a client to display georeferenced map images from a server using data from a GIS database. These map images can, in addition, contain an overlay of vector graphics in the form of geometric shapes and even text. These overlays, or legend, give a visual guide to map elements.

2) Web Coverage Service

The Web Coverage Service (WCS)[9] provides an interface to access multi-dimensional coverage data. Coverage data is an extended feature which has different (and possibly multiple) values at each location. This allows for the representation of properties that vary along dimensions. Examples of these kind of properties are elevation, pollution, pressure and others.

While WMS can return images, WCS can be used for complex modeling by representing variable features along the different dimensions. WCS supports some extensions which also allow for more complex querying and processing.

3) Web Map Tile Service

Building on the WMS specification, the Web Map Tile Service (WMTS)[8] was created to address problems in rendering WMS maps and serve them in an acceptable amount of time. By creating a service where previously rendered tiles can be served, performance is greatly enhanced and the system becomes much more scalable. Most consumer online mapping systems currently use some variant of this standard, by serving responses in tiles.

4) Web Feature Service

The Web Feature Service[10] (WFS) proposes an interface for describing data manipulation operations in a distributed manner on geographic features by using HTTP. A feature is described by a set of properties where each property can be thought of as a name, type, value tuple. Geographic features are those that may have at least one property that is geometry-valued. Basic WFS supports both simple and complex queries based on spatial and non-spatial constraints in the data, allowing for querying and retrieving features. An optional extension to WFS, WFS-T, or transactional WFS creates an interface which supports all CRUD (Create, Read, Update, Delete) operations on feature instances.

B. Web Mapping

In order to present and share maps and spatial data in the web, a lot of engineering power had to be devoted to building interfaces and being able to properly load geospatial data into a user’s browser. With the appearance of Google Maps and their API[17], it suddenly became much easier to integrate a map into a web page. By using their map tiles and processing infrastructure one can (with some limits) easily use maps on the web. After the appearance of Google Maps some open source projects have appeared to also allow web mappings. Currently two open source solutions are widely used.

The first open source widely used library to display maps on a web browser was OpenLayers[23]. The project is a member of OSGeo and has a very large number of features. It is compatible with OGC formats presented in and a number of others.

Leaflet[2] appeared five years after OpenLayers with the goal of making a web mapping library with a more easy to use API. Along with that goal, Leaflet has a much smaller footprint and supports many of the OpenLayers project functionalities via a plugin system. Because of its much smaller footprint it is also a much better choice for mobile compatible applications.

C. GIS Desktop and Server Applications

1) QGIS

QGIS[24] is an open source GIS desktop application for creating, editing, visualizing, analyzing and publishing geospatial data. It is a member of OSGeo and compatible with most operating systems (Mac, Linux, BSD and Windows). It also features a server where you can configure how to publish your
data as OGC compatible WMS and WFS services. It also includes a web client to consume services exported on the web. It allows integration with other open-source GIS software such as PostGIS and MapServer and includes a plugin framework to extend its capabilities. It is a full featured geographical information system competing with the proprietary and closed source ArcGIS.

2) **MapServer**

MapServer[20] was the first open source GIS server. Originally developed with funding and for NASA, it has been releasing new versions since 1997. It is a member of the OSGeo foundation and compatible with OGC standards. It integrates with a number of data sources including PostGIS, supports reprojections and leverages GDAL and PROJ.4. It can present vector and raster data, and has support for complex spatial and attribute based querying. Additionally it includes MapCache which is compliant with WMS and WMTS and serves as a tile caching server. Its interface is minimalistic which makes for a steep learning curve.

3) **GeoServer**

GeoServer[22] offers many of the same functionalities of MapServer. It is more recent than MapServer but its major differentiation is the front end administration panel it offers, which makes it much easier to start using and allows for a better experience exporting data. Additionally GeoServer has support for transactional WFS (WFS-T) and WPS which make it possible to users to alter the database through its API. Its implementations of WFS and WCS are the reference implementations of these standards by OGC.

### D. Geospatial data storage

Geospatial data storages or geodatabases are special databases optimized to deal with data representing features in a geometric space. These optimizations usually refer to the querying of data, by allowing complex queries based on distances, intersections, overlaps and others. In order to improve the efficiency of these kinds of queries, data can also be stored in specialized structures. Since these special modifications to data storage and querying can be added on to regular databases, they are usually added as special plugins or modifications to existing well-known systems.

Table I presents various existing database management systems capable of storing and managing geospatial data, evaluation also they main capabilities.

<table>
<thead>
<tr>
<th>Database</th>
<th>Multiple User</th>
<th>Relational</th>
<th>Spatial Queries</th>
<th>OGC Project</th>
<th>OGC Compliant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatialite[15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PostGIS[23]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MS SQL[21]</td>
<td>✓</td>
<td>✓</td>
<td>✓ (limited)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MongoDB[19]</td>
<td>✓</td>
<td>✓</td>
<td>✓ (very limited)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**TABLE 1**

**DATA STORE FEATURE COMPARISONS.**

---

E. **Architecture requirements for a Geospatial Web Application**

Any generic web service needs some pieces of software from which we cannot deviate. End users will connect to the service typically using a browser or through an API which is served by a web server. In order to have a dynamic application and serve more than static files the web server will have to connect to a programming language through an standard interface, or in some cases this kind of functionality might even be embedded in the server (for example Lua embedded in the nginx web server), or be the server itself (for example node.js). Requests to a programming language are handled and a result is returned. In order to do this a set of libraries can be put together as a web framework to speed up development. These can facilitate access to databases, handle request routing and deal with other common tasks. For some kind of persistence a database of any sort can be used. If we aim to provide search abilities we might plug in a search server which indexes database data and prepares it for search. For long running processes that can be made asynchronously we might plug in a distributed task queue system, and so on. In the case of geospatial systems, another system is usually introduced. A special kind of webserver that understands OGC standard queries and replies appropriately is exposed both to the application and the end user. This puts the geospatial heavy lifting related work outside the main application which only concerns itself with geo-agnostic requests such as authenticating users and creating the context for the interactions with the geospatial data.

The general architecture of a web based GIS application is presented in Figure 2.

### F. Front-end requirements for a Geospatial Web Application

A web application front-end consists on what the user interacts with in a browser. Generally the application will serve HTML formed by some kind of templating engine and from then on Javascript will allow for any dynamic presentation of content. These interactions can be made easy by the use of Javascript libraries to abstract browser and operating system dependent inconsistencies and to ease some common tasks. Additionally, in the case of a geospatial web application we will need some way to receive and present geospatial data to the user. This will include a mapping display with the ability to present features as layers, cluster points, annotate geometries and other common user interaction features on a mapping display. Another front-end use of the service is through an API in which case the response will usually just contain data in some common format like XML or JSON.

---

### III. Supporting Technology

#### A. Database

A database management system (DBMS) had to be chosen from existing open-source solutions. Furthermore, by acknowl-
Fig. 2. Graph of software and protocol relationships of a typical GIS system deployment

Edging that most of the data would be associated with geographic features, the chosen database would have to support geographic information system (GIS) to facilitate the storage and querying of this kind of data. From our analysis of a subset of existing systems, we can already exclude SQLite from the requirement of a multi-user system. Regarding multi-user environments with highly concurrent queries a NoSQL system like MongoDB would be the front runner, but since geospatial functionalities are very limited we also discard this system. Going on number of geospatial features and remembering that MySQL returns spatial queries only for rectangle bounding boxes and is not OGC compliant, PostgreSQL is the chosen system. PostGIS has numerous features, a very large community compared to other open systems, is more stable and has much more comprehensive documentation than any other we found. It is also the database back-end for a large number of software products, from ArcGIS to the OpenStreetMap website, to the Uber mobile and web applications to name just a few.

B. Web Framework

For users to access data, an API has to be designed on top of our data store. Early on we decided on a RESTful (Representational state transfer) architecture, and needed to find a framework with the ability to interact with a generic database driver, which allowed for authentication and authorization methods, had good documentation and had already reached a stable release. Additionally it would be helpful to use a stable and widespread underlying programming language, both to guarantee good library support and to better our chances of getting future project maintainers. Since we needed to develop this service with direct feedback from our end users we also needed to decide on a web framework for the primary website. Stemming from that fact that it had extensive library support to deal with spatial data, plotting and statistic analysis, which would greatly speed up development, Python was selected as the primary programming language for the project. From there, we chose the Django Web Framework[14] since it had the largest feature set, most complete documentation and an active community and development team. Because it fulfilled all of our previously stated requirements, we chose the Django REST Framework to implement our REST API. Most significantly it provided a web browsable API which greatly facilitates the development process.

C. Programming language Interface

A web server is not enough to serve our Python web application and all needed static files (photos, css, javascript files...). We need to use an application container which implements Web Server Gateway Interface (WSGI) to allow messages to be passed from a web server to our Python application. One such container, uWSGI is a fast, self-healing and developer/sysadmin-friendly application container server coded in pure C. Django and the REST Framework extension will be served from uWSGI.

D. Web Server

A web server is needed to expose the application to the Internet. Both the nginx and Apache web servers were analyzed. Both servers have similar capabilities, community and support. Trying to choose based on other capabilities, such as speed, is hard before having a developed application since benchmarks will be very dependent on the use case. Since this was the case we ended choosing nginx for its simpler configuration and its embedded uWSGI support. With nginx we can make communications between the web server and uWSGI go through a UNIX socket which uses uwsgi protocol which is more efficient than if not.

E. Search engine

The database is only accessed from the Django application. Django should interact with a full-text search server so we can provide users the ability to search the website in an efficient way. Two competing fulltext search servers dominate in terms of features and community, Solr and ElasticSearch. Since no major limitations were encountered in any one of them ElasticSearch was chosen for its from the ground up distributed nature, and arguable ease of use and configuration.
F. GIS Server

Regarding the GIS Server, the needed functionalities were analyzed. At first the overwhelming amount of features they provided may entice us into using a full fledged solution and try and integrate as many of these features as we can. What was found out as the project went on though, was that only a minor subset of these features would be truly important for our specific use case. As such a decision was made to not use any external GIS server and implement the needed functionalities into the application. Since Django provides a modular way of developing "apps" with contained features if it ever becomes apparent that an external GIS server is actually needed, we should not incur in much added development effort to pull the functionality out.

IV. DJANGO ARCHITECTURE OVERVIEW AND REQUEST LIFE-CYCLE

![Django architecture overview](image-url)

In order to explain the implementation of the data model and some of the decisions that were made, it is important to understand how all the moving parts of the Django web framework work together. A Django project starts by having a small amount of configuration files, a file with URL patterns and a small script to allow the framework to interact with a web server. There are applications, which are used to perform a particular task. As an example, if a website has a discussion board such as a forum, an application might be responsible for user private messaging, another for the forum threads, and another for registration. These applications can be re-used and shared, making them effectively like an extension, or plugin, for Django. Inside each application folder are its models, views and templates, following the ModelView-Controller (MVC) architectural pattern (Figure 3). In Models the data is defined, in Controllers (Django views) actions are performed, and in Views (Django templates) we typically return HTML pages following some presentation logic.

V. SANDDB

A. Data modeling

We started to develop the data model by assuming a registered user would input samples and their properties into the system. A sample has an owner which is the original uploader of the sample. A user which has ownership of a sample (write privileges) is able to input further data and test results. There is also the concept of grouping samples. The first possible grouping is a Campaign which exists when a user collects one or more samples in an outing by performing some field work in a limited location and time-frame.

![Sample model](image-url)

Listing 1. The sample model implementation.

class Sample (models.Model):
    user = models.ForeignKey(User)
    expedition = models.ForeignKey(Expedition)
    description = models.TextField()
    sandDB_code = models.CharField()
    retrieval_date = models.DateTimeField()
    retrieval_location = models.CharField()
    latitude = models.FloatField()
    longitude = models.FloatField()
    retrieval_depth = models.IntegerField()
    retrieval_height = models.IntegerField()
    geopoint = models.PointField()
    # manager for geodjango
    objects = models.GeoManager()
    institution = models.ForeignKey("Institution")
    collection = models.ForeignKey("Collection")
    creation_date = models.DateTimeField()
    update_date = models.DateTimeField()
A sample has an owner which is the original uploader of the sample. A user which has ownership of a sample (write privileges) is able to input further data and test results. There’s also the concept of grouping samples. The first possible grouping is a Campaign which exists when a user collects one or more samples in an outing by performing some field work in a limited location and time-frame.

Finally there are Classifications which are a kind of tagging system. A sample can have multiple classifications and samples can be searched, grouped and filtered by classifications. A site administrator can create any number of classifications types with possible values. These are implemented as ClassificationGroup’s and Classification’s in a way that’s extensible by a non-developer user (Figure 5).

We only have to identify common analysis and their output data, create a table which models it and link it to our Sample model by a foreign key. The first test we decided to support was the granulometry [6], analysis since it is a widely performed operation with very useful output data. This test has a variable number of possible sieves, but the possible sieves are generally standardized. In this case we pre-fill the database with the default sieves and a user is then able to choose which sieves were used in a given test. The actual implementation of this classification scheme applied to granolometry is presented in Figure 6.

Another simple analysis which was also implemented was the pH test, which again, consists simply in a table with a relation to the Sample and in this case a DecimalField representing the actual pH value. Users can also upload photos and generic PDF documents in order to support other tests which are not currently implemented in the data model. This demonstrates the ability to simply plugin data to the sample model and support more tests as they are deemed necessary. In the special case of photos there are several types of photos which can be inserted by an admin and a user can then select which type it belongs to (for example x-ray vs microscope photos).

B. Users and permissions

An implementation of users and model level permissions is provided by the Django framework. We leverage these models and only needed to create a Profile model which will store extra information about a user (Figure 14). As for the model level permissions, they allow us to create groups of users and grant permissions based on group membership. These are useful to define users with administration or moderation rights per-model, but not per-object. Row level permissions can be implemented to allow several users to have different read/write permissions on samples instead of a basic single owner model. Row level permissions, or per object permissions can be implemented in Django as an object aware authorization backend by linking users to generic "content types" (Django’s way to generically refer to data models).

C. REST API design

Every endpoint returns data in the format specified by the HTTP request headers, which in the case of the browser is HTML. The format can also be specified by adding ?format = json to the URL which will then return data in the JSON format. In order to exemplify our URL schemes we will use regular expressions (regexes) which are commonly used for pattern matching in strings and in this case are used by Django’s default URL routing code. The implemented REST API supports the GET, POST, PUT, DELETE, HEAD and OPTIONS HTTP verbs. Every resource answers to HEAD exactly as it would if it had been a GET request but only with the response headers. OPTIONS requests are answered with the possible HTTP requests and what each of them expects.
from the user in terms of submitted data at that specific endpoint. Contrasting with the previous request types, we also implement methods with side effects on the server. These are used to create, update and delete resources. We use POST to create resources by hitting the base resource URL. In order to update a given resource we PUT to its specific URL (using its identifier). Likewise, if we only mean to apply partial modifications to a resource we would hit its specific resource URL with a PATCH request. Finally we can delete a resource by making a DELETE request to its specific URL. Together with the safe methods we get full CRUD (Create, Read, Update and Delete) functionality on our data models with these types of endpoints.

<table>
<thead>
<tr>
<th>URL Style</th>
<th>HTTP Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource/$</td>
<td>POST</td>
<td>create</td>
</tr>
<tr>
<td>resource/(?P&lt;pk&gt;[0-9]+)/$</td>
<td>PUT</td>
<td>update</td>
</tr>
<tr>
<td>resource/(?P&lt;pk&gt;[0-9]+)/$</td>
<td>PATCH</td>
<td>partial update</td>
</tr>
<tr>
<td>resource/(?P&lt;pk&gt;[0-9]+)/$</td>
<td>DELETE</td>
<td>destroy</td>
</tr>
</tbody>
</table>

TABLE II
SIDE EFFECT METHODS OVERVIEW FOR A GENERIC RESOURCE.

D. Overview of implemented endpoints

Using the methods presented above, we’ve implemented endpoints for the resources in our presented data model.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>users/$</td>
<td>Users of the system</td>
</tr>
<tr>
<td>groups/$</td>
<td>Groups of users (used for model level permissions)</td>
</tr>
<tr>
<td>samples/$</td>
<td>Samples inserted by users</td>
</tr>
<tr>
<td>granulometries/$</td>
<td>Granulometry tests associated with samples</td>
</tr>
<tr>
<td>campaigns/$</td>
<td>Field work grouping several samples</td>
</tr>
<tr>
<td>photos/$</td>
<td>Photographies associated with a user or sample</td>
</tr>
<tr>
<td>files/$</td>
<td>PDF files associated with samples</td>
</tr>
</tbody>
</table>

TABLE III
IMPLEMENTED API ENDPOINTS.

Adding endpoints to the API is very simple and consists in defining both a model Serializer and a ViewSet. Using the Sample class model presented in Listing 1 we present both its Serializer and its Viewset. A ViewSet consists of a query to the database and a definition of its serializer class in case we want to customize the output JSON.

Listing 2. Sample ViewSet.

class UserViewSet(views.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = serializers.UserSerializer

The Sample serializer (Listing 3) is one of the most complex in the project and as such is used to give a good example of available features. It defines a special hyperlink fields which link to the specific objects for the actual sample, its owner (User) and the expedition on which it was collected. It also embeds the Granulometry information inside. Additionally, we declare a geo_field and inherit from GeoFeatureModelSerializer allowing us to turn the JSON object into a GeoJSON object instead, with a feature of type Point on the geopoint attribute and so allowing the Sample table to be consumed by any GIS service compatible with GeoJSON.


class SampleSerializer(HyperlinkedModelSerializer, GeoFeatureModelSerializer):
    url = serializers.HyperlinkedIdentityField(
        view_name='api-sample-detail',
    )
    user = serializers.HyperlinkedRelatedField(
        queryset=User.objects.all(),
        view_name='api-user-detail',
    )
    campaign = serializers.HyperlinkedRelatedField(
        queryset=Campagn.objects.all(),
        view_name='api-campaign-detail',
    )
    granulometry = GranulometrySerializer(
        read_only=True,
    )

class Meta:
    model = Sample
    id_field = False
    geo_field = "geopoint"

During the evolution of the SandDB system, it will be possible to create additional endpoint to export the newly created data types.

E. Interface

Regarding the visual aspect of the interface, in its current form it looks as its shown in the following figures. All of the presented data is for testing purposes only and doesn’t represent real samples.

Figure 8 shows the user Dashboard, the main page a user sees upon login. In this page a user can see his latest Campaigns and Samples and has controls to add new ones.

Figure 9 shows a Campaign detail page where anyone can see a Campaign and its properties. It includes a map with all the Samples associated with the Campaign.

Figure 11 shows a Sample detail page where anyone can see all the information associated with a given Sample. It shows
all the analysis that were performed, the classifications that the Sample was given as well as pictures and associated PDF’s. We can also see where the Sample was collected in an interactive map.

Figure 10 shows a search page. It is showing a Sample search and it shows a way to refine the search by the user which collected the Sample. These searches are powered by the external search server and can be refined by different parameters.

F. Importing and Exporting data

Importing data into SandDB was a main concern of ours when developing the platform. Even though we have built an API and regular browser forms to import data, each of them has a problem. The API will only be used by advanced users which integrate some tool they already use. On the other hand, the forms, while perfectly good for a small amount of information, quickly become an herculean effort if we imagine data sets with hundreds or thousands of samples. As such, we devised other methods of importing data into the platform.

The first is a bulk import form which allows for faster data entry of multiple samples. It expands to insert as many samples as we want at the same time. This method is shown in figure 12.

The second bulk import builds upon the first and can be accessed by pressing "switch to spreadsheet view" in the previous screen. It emulates a spreadsheet in the browser and is compatible with tools such as Excel regarding copying and pasting cells and formatting both into and out of the browser. It also allows for any number of samples to be inserted as the spreadsheet auto-expands with insertion. This interface is shown in figure 13.

The final bulk import method works by creating (or downloading our default file) a .xls file with the appropriate columns and filling the data directly in a desktop spreadsheet program of choice. This file can then be imported into the platform and new data will be imported, as well as supporting smart updates to existing data.

The import of data is accomplished performing a simple copy-and-paste from the desktop spreadsheet into the SandDB insert form, easing and reducing errors in the data insertion procedures.

VI. DJANGO-WFS APP

We made the decision of not using a full fledged external GIS server as a dependency because we believed it would add more complexity than needed for the features we had in mind. The features we did have in mind were interoperability with existing GIS services like the QGIS desktop application, where researchers usually load external web services. QGIS has support for GeoJSON data formats and so we could load this data in QGIS from our API. Even though GeoJSON is quite good for modern web applications, it only serves very basic needs in QGIS. QGIS’s support for GeoJSON does not include any kind of filtering or pagination of results which makes it quite unsuitable for any type serious usage. For these reasons, we decided to also provide users the ability
to export Features (Samples) as a WFS 1.0.0 service. After searching for a suitable project compatible with Django or even Python without using a full external server, the decision was made to implement our own service. Our implementation of WFS was based on the official Web Feature Service Implementation Specification, a 90+ page document describing all the capabilities of a compliant service. Implementation started by creating the data models for the WFS application. The basis of this is the Service which exposes multiple FeatureType’s. Each FeatureType can contain BoundingBox’s and MetadataURL’s. Generically each Service would represent a different kind of geospatial data, which in the case of SandDB, currently, only supports the Sample model. A FeatureType on the other hand can represent any grouping of objects under a Service. In SandDB for example this was used to autogenerate FeatureTypes for every User and every Campaign.

VII. Conclusions

In this thesis the problems geology professors, students, researchers and enthusiasts face with sample archival, sharing and discovery were presented. We analyzed existing tools which tackle this problem and found them inadequate for their broad scope, for not allowing everyone to participate and for being hard to use. We defined some subjective goals of a platform which could eliminate these problems and turned them into technical requirements. A lot of time was dedicated to the comparisons of existing supporting tools, frameworks and libraries that would be a good fit for development and future maintainability of the project. The SandDB platform is a first step to tackle a bigger problem of collaboration and sharing of scientific data, in the specific case of sediment samples. It proves it is feasible to implement a web based GIS system using open-source technologies and web frameworks. This is currently not obvious to many people who work with these types of systems, specially when mostly exposed to proprietary solutions. The open-source ecosystem around geospatial data is in fact very mature and ready to be used by this genre of applications. In a more broad scope, the implementation of the WFS standard into Django can be a first step to further implementations of other OGC standards and allowing for the creation of a full OGC compliant stack on top of Python and Django without requiring a full-fledged external server as a dependency for geospatial projects. This implementation is specially important to make evident that going from official specification to working software is not that hard. While proving that implementing a web-based GIS system in a short period of time is possible, it would be desirable to test it with more extensive (and real) datasets. Some efforts were made in order to obtain collaboration but these were not immediately successful, which does not mean future attempts ca not be. While most of the thesis is aimed at the particular case of sediment samples, by using the core of the work - REST API, User management, front-end mapping capabilities, Django-WFS - one has most of the pieces all ready to develop other similar GIS systems.

With SandDB we managed to implement a fully functional GIS application resorting to commonly available open-source tools. The development of the systems followed and was made with the common knowledge of regular web applications. This demonstrates that the development of this class of applications...
is not restricted to develops specialized in GIS systems.

The develop system is modular and easily configured (for instance it is possible to create aditional classification categories, without modifying the overall system.

The availability of external services APIs allows the integration of SandDB with external systems. The implemented REST API will allow the reuse of the data stored in different applications, for instance the creation of educational applications that retrieve information and data from the SandDB. The implementation of a WFS service allow the use of the information on current GIS systems (QGIS or even ArCGis).

A. Deployment

We predict three possible deployment cases for this thesis' project, consisting of private and/or public nodes. Private nodes would satisfy institutions for which data privacy is the most important requirement. Public nodes would satisfy our requirements of discoverability and community participation. A hybrid solution would allow users of private nodes to dynamically share partial datasets with some set of other nodes in the network. This hybrid solution is out of the scope of this project because its dynamic permission models would have to grow from a more stable platform. The three possible types of deployed networks are shown below:

1) Single node with several institutions and users.
2) One private node per institution, one node for public users.
3) A hybrid federated network of public and private nodes.

B. Future Work

From this first stage of research and development we have created a base system which supports most of the functionalities we believe are needed for the tool we proposed to create. As for other future work that we anticipate:

1) Implement more analysis as users require them
2) Implement a more fine-grained permission system
3) Implement user interaction directly into the platform in the form of Sample requests.
4) Increase discoverability in the form of map based interactive searching.

Another non-technical goal can also be defined, which will nonetheless arguably be of great importance to the viability of this work, getting users. By getting users into the development loop, providing feedback and using the platform, bugs will be more rapidly discovered, most used features will be identified and improved upon and unnecessary features might be discarded.