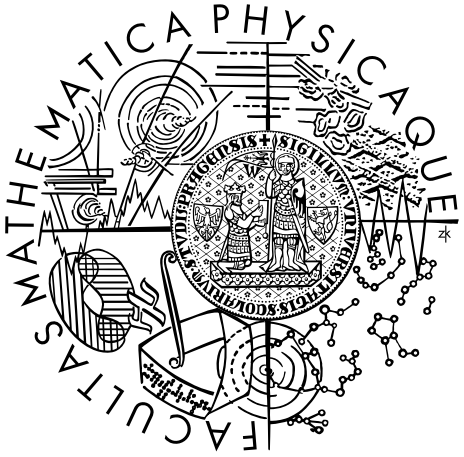
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Olga Andreeva

Cloud database for soil information

Department of Software Engineering

Supervisor of the bachelor thesis:	doc. Mgr. Martin Nečaský, Ph.D.
Study programme:	Computer Science (B1801)
Study branch:	IOIA (1801R008)

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Cloud database for soil information

Author: Olga Andreeva

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract:

The mere presence of scientific data is not enough. Reliable storage and ease of publication critically contribute to the data's usefulness.

The Africa Soil Information Service's mission is to describe and understand Africa's soil and landscape resources. To ensure its accomplishment the introduction of modern information technologies is essential - they provide the necessary means to safely store, easily share and analyse the information collected from soil samples and drone imagery.

In the presented work we develop a system that will tackle one part of the problem - how to store the information extracted from soil samples. The current method that is employed by scientists consists of making a record on paper by hand in one of the books in the laboratory's storage room. Such system, of course, does not allow to easily share the data, it is prone to damages (e.g. in case of fire) and leaves little room for analysis (especially using methods from machine learning, which require a large set of digital data). By contrast our system, developed in the form of a web application, stores soil information in the cloud secured by backups, provides a convenient user interface for searching and fetching the data and a flexible permission system to control access to the data.

The developed system underwent testing and is being successfully used in laboratories in Tanzania.

Keywords: cloud database web application soil information

Contents

1. Introduction

1.1 Food Security

The United Nations Food and Agriculture Organization estimates that about 10 percent of people globally were undernourished in 2015. This constitutes about 795 million people suffering from hunger, down 167 million over the last decade [The United Nations Food and Argiculture, 2015]. Economic growth is essential for reducing undernourishment, but its benefits must be shared by every section of society. One of the components of the inclusive growth is enhancing productivity and income security of smallholder family farmers.

Despite significant progress towards achieving the internationally established hunger targets (the World Food Summit, the Millennium Development Goals), progress is not distributed uniformly - it has been slower in the countries of Southern Asia and sub-Saharan Africa.

The situation in the United Republic of Tanzania seems especially disturbing. Despite average annual GDP growth of 2.3 percent since the beginning of the 1990s, the prevalence of undernourishment increased from 24.2 percent to 34.6 over the same period. This means that around 17 million people living in the United Republic of Tanzania suffer from the lack of proper nutritions. The agriculture sector is dominated by small family farmers producing for subsistence, however, little has been accomplished in modernization of their smallholdings [The United Nations Food and Argiculture, 2015].

1.2 The AfSIS project

The African Soil Information Service (AfSIS)[1] was started in 2009 by The Tropical Agriculture Program [2] and its partners to develop a digital map of soil properties of the non-desert portions of sub-Saharan Africa [The Earth Institute of Columbia University, 2012]. Understanding nutrient levels and other properties of soil allow for more precise selection of agricultural methods and fertilizers, which leads to increase in productivity of the farmers. Ethiopia and Tanzania became the first countries where the soil information is collected.

1.3 The AfSIS Database

Collected soil samples are geo-referenced and measured on a spectrometer. Spectrometer measures light absorbed by the sample at hundreds of specific wavebands across a range of wavelengths to provide an infra-red spectrum. Reference soil samples that chemical analysis was performed on are calibrated to the infra-red spectra. The calibration model are then used to predict soil properties of the whole sample set.

The collected data was initially stored in the laboratories' storage rooms locked with a key. The technicians had to write down the chemical properties

[1] `http://www.africasoils.net/`

[2] `http://agriculture.columbia.edu/`

by hand with a high chance of a mistake. The samples were numbered manually across several books which again raises probability of an error. One of the laboratories in Tanzania even experienced a fire that destroyed hundreds of records. And, of course, the opportunities for an analysis, especially on a large scale, are very limited in such a setting.

It became clear that a digital system was urgently needed. It started as a collection of spreadsheets that were emailed between scientists at different laboratories. But then the threat of losing without way of restoring or inadvertently modifying data is still high.

The next step was a decision to develop the project's own database. It had to eliminate all disadvantages of the approaches outlined above, namely it had to:

1. securely and safely store the input data
2. provide convenient interface for entering the data, of soil samples in particular
3. provide convenient interface for querying the data, by soil properties or geographical location
4. allow sharing of access to the data
5. support multiple sample tables with different columns, each corresponding to a sample type

The development of such a database, which from now on will be referenced to as African Soil Information Service Database (AfSISDB), is the goal of this thesis.

1.4 Document structure

The second chapter describes the requirements in greater detail and defends the proposed architecture from the point of view of their satisfaction.

The third chapter provides a technical background to the developed system. It describes technical tools which are critical to its functioning.

The fourth chapter introduces the project to a software developer and explains essential design decisions that were made during its development.

The fifth chapter describes how the developed system has been tested.

The sixth chapter contains a user guide which introduces the system to the users.

2. Analysis of the Requirements and the Proposed Solution

2.1 Requirements

In the Introduction (1) we have described the overall goal of the project and now particular requirements requested by the stakeholders will be investigated.

As was mentioned in the introduction the previous stage of managing soil information consisted of CSV files. One of the arguments in favour of their use is the ease of data extraction (using R language, numpy package in Python, etc.) and that virtually no software for their viewing and editing is required.

Considering these advantages, the format is included as the main input format of our system. The user is able to upload a CSV file with required columns and the data is inserted into the database. The export to CSV is also present.

CSV is a text format and the samples which are imported through it already have their chemical data defined in the file. There are also "binary" samples. They are imported as a collection of files, one file per sample, and must be preprocessed and uploaded to the storage service.

Some types of soil samples contain geographic information in the form of their latitude and longitude coordinates.

2.1.1 Registration and authentication of users

1. Registration is closed. The user must be able to send an access request which will be emailed to the administrators, specified in the settings.

2. Once account for the user is created, the user must be able to log in with their credentials.

2.1.2 Permissions

1. An administrator must be able to specify permissions available to the user. The following permissions must be available for each sample table:

   - *view*
   - *change*
   - *add*
   - *delete*

2.1.3 Access to the data

1. The home page must list sample tables that are available to the user.

2. On each sample table's page its records must be displayed.

   (a) The list of samples must be displayed in the table with one column containing a sample's SSN with a link to the sample's page.

(b) The records must be paginated.

(c) The user must be able to select multiple records and delete them.

(d) If the user has "add" permission, there must be an *"Add ..."* button.

(e) Search by SSN must be present.

(f) In case of "Spectroscopy OPUS records" table, private records must be displayed only if the user has uploaded them.

3. On each record's page all fields defined in the model and their values must be displayed.

   (a) If the user has *change* permission the fields must be editable.

   (b) If the user has *delete* permission, a "Delete" button must be present.

   (c) A link to "History" page must be available.

      i. On "History" page all the versions of a record must be displayed in a table with the following columns:

         - *Date/time* - when the change was made
         - *User* - who made the change
         - *Comment* - description of the change

      ii. If the user has *change* permission, the user must be able to revert to each of the versions.

2.1.4 Binary samples upload

The following requirements apply to "Spectroscopy OPUS records" table only.

1. "Upload binary file" link must be present on the table's page.

2. On "Upload binary file" page the user must be able to upload local files

   (a) The user must be able to select whether samples are private or not.

   (b) A new record must be created for each uploaded file.

   (c) Fields of a created record must be shown to the user.

   (d) The user must be able to delete newly created records.

2.1.5 CSV import

The following requirements apply to the text-based samples (currently all except "Spectroscopy OPUS records" table).

1. Importing from CSV must be available only to the users who have *add* and *change* permissions.

2. "Import from CSV" link must be present on the table's page.

3. On "Import from CSV" link the user must be able to select a local file.

   (a) Fields defined in the table's model are required to correspond to a column in the CSV file.

(b) Required fields are displayed on the page.

(c) If some fields are missing or the file is not in the CSV format, an error message is displayed to the user.

4. After the file is submitted, its rows are parsed and displayed to the user in a table with fields displayed as its columns.

5. The user must be able to confirm the import once all the rows are parsed.

6. The progress of the import is shown to the user with status of each rows displayed which takes the following values:

   - *"Importing..."*
   - *"Record was updated"* (in case a record with such SSN already exists, its fields are updated)
   - *"New record was created"*
   - *"Failed"*

7. After the import is finished, the user must be able to see all imported records or only failed or only updated records.

8. After the import is finished, the user must be able to see an explanation of failure for each failed record.

9. All imports made by the user must be available to them under "Your Imports" link.

   (a) "Your Imports" page must display a table with the following columns:

   - *Table* - a name of the table that the file was imported to
   - *Status* - a status of the import, one of the *imported*, *processed*, *processing*, *error*
   - *Page* - a link to the import's page
   - *Started* - how long ago the import was started

2.1.6 CSV Export

1. "Export to CSV" link must be present on the table's page.

2. The user must be able to select records on the table's page and export the selected records.

3. All export made by the user must be available to them under "Your Exports" link.

   (a) "Your Exports" page must display a table with the following columns:

   - *Table* - a name of the table that the file was imported to
   - *Status* - a status of the import, one of the *imported*, *processed*, *processing*, *error*.
   - *Page* - a link to download the CSV file
   - *Started* - how long ago the import was started

2.1.7 Connection to Mobilesurvey

Mobilesurvey, available at `https://mobilesurvey.qed.ai/`, is a web service that helps in conducting surveys with questions tied to a geographical location.

For a sample table that exists on AfSISDB there may be a corresponding survey on Mobilesurvey. Once this correspondence is known, additional data can be retrieved for each sample using its identifier from the database on Mobilesurvey.

1. "Connect to Mobilesurvey" link must be present on the table's page.
   (a) On "Connect to Mobilesurvey" page the user must be able to select a form from Mobilesurvey and fields that must be matched.
2. After a connection is created, on each sample's page relevant data from Mobilesurvey must be displayed. If a field "a" of the form on Mobilesurvey and a field "b" of the table were selected to be matched, the data of the entry on Mobilesurvey s.t. the value of "a" equals to the value of "b" must be displayed.
3. After a connection is created, it can be edited on "Edit connection to Mobilesurvey" page, a link to which must be present on the table's page.

2.1.8 Recovering of records

1. "Recover deleted" link must be present on the table's page.
   (a) On "Recover deleted" page for each record that have been deleted the last version must be displayed in a table with the following columns:
      - *Date/time* - when the record was deleted
      - *Name of the table* - SSN of the deleted record
      - *Deleted by* - who deleted the record.
   (b) If the user has *change* permission, the user must be able to recover a record.

2.1.9 Search across tables

1. In the left sidebar a link to "SSN Search" must be available.
   (a) On "SSN Search" page the user must be able to enter a search query and select across which table the search should be performed.
   (b) After the user clicks "Search", a paginated result set must be displayed.
      i. Results for each table must be in a separate table with one column that contains a record's SSN.
2. In the left sidebar a link to "Search inside circle" must be available.
   (a) On "Search inside circle" page the user must be able to enter *radius*, *latitude* and *longitude*.
   (b) After the user clicks "Search", a paginated result set must be displayed that contains all records that are located inside the specified circle.

i. Results for each table must be in a separate table with one column that contains a record's SSN.

3. In the left sidebar a link to "Search inside bounding box" must be available.

   (a) On "Search inside bounding box" page the user must be able to enter *minimum latitude*, *minimum longitude*, *maximum latitude*, *maximum longitude*.

   (b) After the user clicks "Search", a paginated result set must be displayed that contains all records that are located inside the specified boundaries.

      i. Results for each table must be in a separate table with one column that contains a record's SSN.

2.1.10 Filters

1. On each table's page "Filter" link must be displayed.

   (a) After "Filter" is clicked, a section with filters must appear with one filter displayed.

   (b) The user must be able to select a field and its value.

   (c) The user must be able to add more filters and reset filters.

   (d) For each field the user can add a filter, and in addition filters for maximum and minimum values for each numeric field can be added.

   (e) After clicking "Filter" button, the table of records must contain only records satisfying the filters.

2.1.11 API

1. For each table an API point that retrieves all the records must be available.

2. Each API point must accept the same parameters as in filters.

3. Each request to the API must require authentication.

2.1.12 Extension and Distribution

1. The resulting software must be extendible:

   (a) no or a limited knowledge of programming must be needed to add new sample tables

   (b) developers must be able to launch their own customized instance of AfSISDB with small overhead cost

2. The source code has to be open-sourced once the listed requirements are satisfied.

3. Usage of proprietary software has to be minimized (ideally, only open-sourced project must be used).

4. The final software must be as affordable as possible – in case another company or institution wants to launch its own instance of AfSISDB, it must be easy for them to do so (i.e. a configured virtual server with requirements installed from standard package managers is sufficient).

2.2 Proposed solution

Taking extension and distribution requirements into consideration, designing AfSISDB as a web application offers significant advantages: those who would like to have their own instance of AfSISDB need not to worry about maintaining a cross-platform software; installation and distribution are very user-friendly.

From the requirements to the data access and search, it follows that the project requires a non-commercial relational database with support for spatial querying. Essentially two engines fit the description – MySQL and PostgreSQL with PostGIS extension. The full comparison of these choices is unnecessary – both engines have been successfully used in production for large-scale web applications. Hence in the Table 2.1[1] we provide a comparison based only on the features crucial for our project.

Feature	MySQL	PostgreSQL + PostGIS
JSON field (used when importing CSV):	-	+
Spatial indexes:	-	+
Spatial Reference System Identifiers:	-	+
Supported in any web framework?:	No	Yes, Django

Table 2.1: Comparison between MySQL and PostgreSQL + PostGIS based on the project requirements. JSON field is used for CSV import, Spatial Reference System Identifiers is required to calculate the distance between two points in a non-planar coordinate system.

Combination of PostgreSQL and PostGIS emerges as a clear winner.

Taking an existing web framework as a foundation would not only decrease time and cost of the development, but would also make customization more straightforward for outsiders. With respect to our project, Django, which is written in Python, has several advantages:

- Support for PostGIS
- Administration dashboard, which will be extended in AfSISDB
- Ease of defining new classes (namely sample tables) even for users with little programming experience
- Python is a dynamically typed language which will allow us more flexibility when working with multiple sample classes

[1] MySQL 5.7.4 (released on 2014-03-31), PostgreSQL 9.3.4 (released on 2014-03-20) and PostGIS 2.1.3 (released on 2014-05-13) are considered.

- Extensive collection of libraries, some of which are used in the project (e.g. asynchronous tasks, API)

Based on the requirements and keeping key design principles in mind, the following architecture is proposed:

1. Represent each sample table by a separate class.

2. Separate each required feature into a separate module, e.g. connection to Mobilesurvey, API and search functionality must behave independently and must not be mandatory for each new table.

3. Follow the guidelines imposed by Django as much as possible in order to make the architecture more obvious.

4. Since design must not depend on the existence of any particular tables, i.e. all functionality must assume as little as possible about the structure of the data, create new samples by inheriting from a general sample table class which is compatible with independent functionality modules.

The proposed architecture is represented graphically in Figure 2.1. Each sample table is represented by an explicitly defined class (Sample model). Features that are available for each Sample model can be categorized into those that are part of the administration functionality and those which depend on the data only. The first class is activated through the creation of a ModelAdmin class for a Sample model, which can inherit functionality as necessary. Each ModelAdmin is registered in a single AdminSite class, which inherits from classes that provide site-wide functionality. "Functionality modules" do not depend on each other and are optionally inherited from.

The API, as an example of a functionality that is not available through the administration interface, connects directly to the model layer and has no communication with the administration functionality.

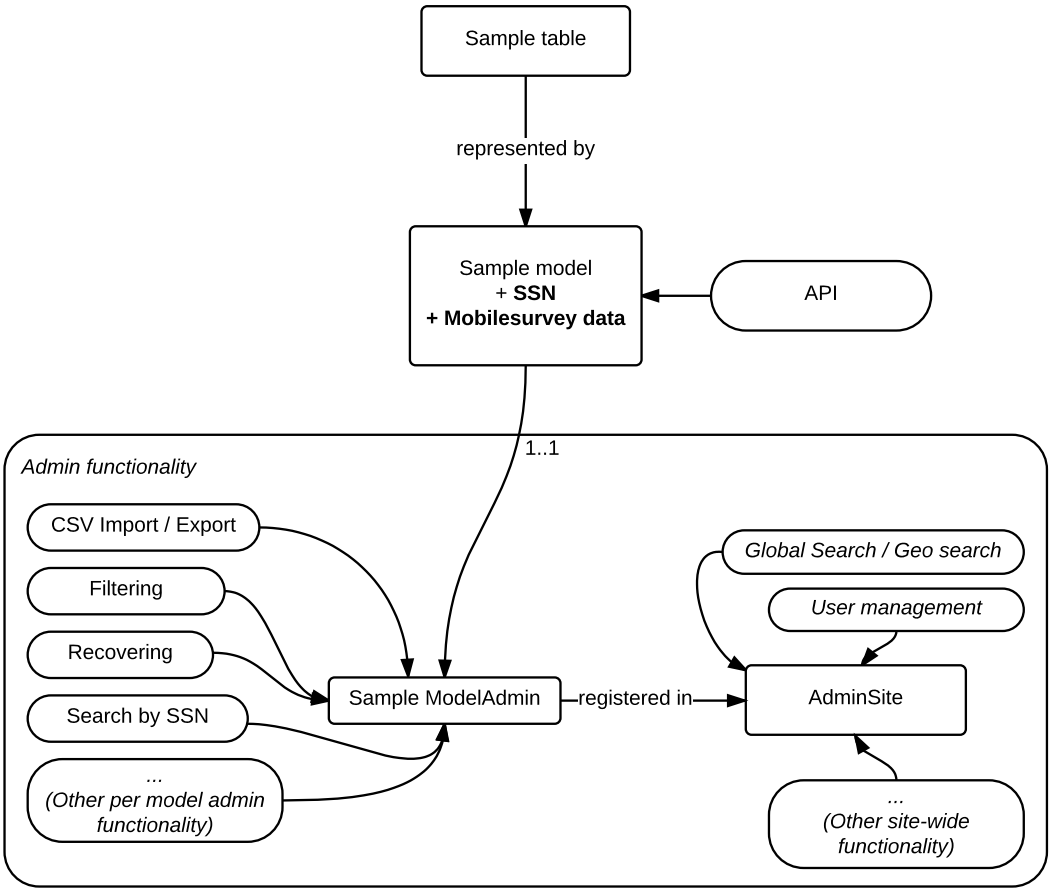


Figure 2.1: Architecture of AfSISDB.

3. Technical Background

3.1 Web framework

Django follows the Model-template-view (MTV) pattern. MTV consists of the following three layers [Django, 2015b]:

1. Model layer
2. View layer
3. Template layer

The model layer mediates between models, defined as subclasses of `django.db.models.Model`, and a database; the view layer selects data for rendering and sends it to a template; the template layer decides on the representation of data. From now on, when we speak about "sample models", we mean the definition of the corresponding sample class in the model layer; by "sample table" we mean the structure that is described in a particular sample class.

3.2 Environment

Apart from the described "core" tools, the project connects to several more services:

- Apache Solr[1] - a search platform, used for fast across-tables search.
- Amazon Web Services S3 bucket[2] - a cloud storage solution, used for binary samples.
- MongoDB[3] - a non-relational database engine, used in another AfSIS project (Mobilesurvey), to which the AfSIS database connects and fetches information from.
- Celery[4] - an asynchronous task queue, used in combination with RabbitMQ Server[5] as a broker to perform long-running computations outside of the request-response cycle.

The overview of the technical components is presented in the Figure 3.1. Parts which can be changed in a specific production environment are coloured in grey — the application does not depend on any kind of the communication to the HTTP server and the file storage.

[1] `http://lucene.apache.org/solr/`
[2] `https://aws.amazon.com/s3/`
[3] `https://www.mongodb.org/`
[4] `http://www.celeryproject.org/`
[5] `https://www.rabbitmq.com/`

3.3 Installation

To launch the application, it is necessary to install several dependencies first. The dependencies can be divided into two categories: python and non-python dependencies.

Python dependencies are listed in `requirements.txt` and can be installed using `pip`:

```
pip install -r requirements.txt
```

Non-Python dependencies are: PostgresSQL (¿=9.4), PostGIS, MongoDB, Solr (==4.x.x, tested on 4.10.2), RabbitMQ, AWS S3, Node.js. Node.js's packages are located in `package.json` and can be installed using the following command:

```
npm install
```

If the application is being launched on production server, the flag `production` must be used:

```
npm install --production
```

Node.js installs Bower which has its packages listed in `bower.json`. To install them, run:

```
./node_modules/bower/bin/bower install
```

Binary files which are uploaded to "Spectroscopy OPUS records" table are processed by a script, available as Attachment I, which must be placed into `cabinet/code/`.

3.4 Configuration

The settings are defined in `afsisdb_web/settings/base.py`, but this file must not be overridden. Instead, a new file similar to `afsisdb_web/settings/local.py.example` must be created where the custom settings are to be defined.

3.4.1 Solr

1. After installing Solr 4 the port number listed in Solr status must be specified in the settings file such that the port number follows 127.0.0.1:. For example, if the port number is 8983, this part of the settings file must be:

```
HAYSTACK_CONNECTIONS = {
  'default': {
    'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr'
  },
}
```

2. To generate the Solr schema:

```
./manage.py build_solr_schema
```

The output must be placed to `/usr/share/solr/conf/schema.xml` (Ubuntu, Solr 4.10.2, the path can vary on other systems). For more details, see [Haystack, 2015].

3.4.2 Keeping Solr up to date

1. If search indices have changed, new Solr schema must be generated.
2. If he schema has changed the index must be rebuilt:

```
./manage.py rebuild_index
```

3. The index of Solr is *not* updated automatically. It is done by running `python manage.py update_index`. Locally, one can use `SimpleEngine` backend of haystack (then Solr is not used), as specified in `base.py`. On production, it is advasible to set up a cron job that will update index periodically, like this:

```
*/5 * * * * python manage.py update_index --remove \
--start="$(date --date='-5 minutes'
         +\%Y-\%m-\%dT\%H:\%M:\%S)" \
--end="$(date +\%Y-\%m-\%dT\%H:\%M:\%S)" \
--settings=afsisdb_web.settings.production
```

   Here, every 5 minutes records which are not older than 5 minutes will be updated. Paths to python and manage.py have to be specified according to the configuration.

3.4.3 Celery

Install celery init scripts: celeryd.

3.5 Update

Before updating, make sure that these tests pass:

```
# The test suite
./manage.py test
# Python code quality
flake8
# JavaScript code quality
./node_modules/eslint/bin/eslint.js
```

3.6 Creating Users

Each user must be given staff status ("Permissions" panel) to be able to log in.

In order to create the first superuser, run the following command:

```
./manage.py createsuperuser
```

More users can be created through the web interface.

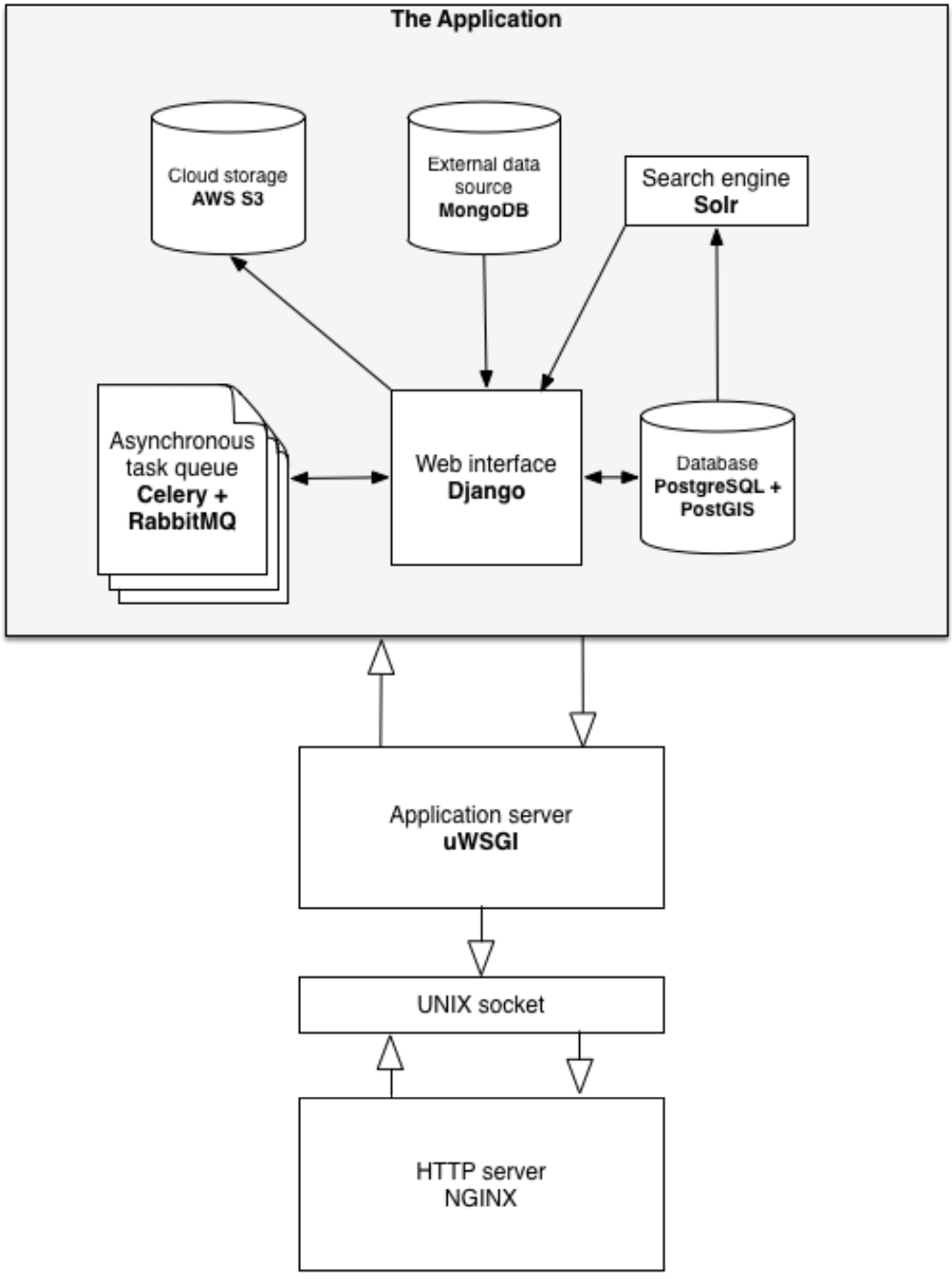


Figure 3.1: The overview of the project's technical environment.

4. Developer Guide

4.1 Overview

The core of the project is written in Python using Django framework and it follows structure guidelines defined by the framework, i.e. it consists of *apps* - submodules which are responsible for a part of functionality and can be potentially used outside of the project, each of which follows the MTV pattern.

On a high level, our project is a customization of the Django's built-in admin site, which is automatically generated. The main classes are described in the Table 4.1.

Class	Description
`django.db.models.Model`	A source of information about the structure of the data
`django.contrib.admin.ModelAdmin`	Encapsulates the administration functionality of a particular model
`django.contrib.admin.AdminSite`	Represents an instance of the administrative site

Table 4.1: Main classes of Django for AfSISDB [Django, 2015a]

On the level of apps, our project consists of 7 submodules, described in the Table 4.2.

The dependencies between modules are described in the Figure 4.1. For simplicity only submodules with external dependencies are shown and `tests` app is omitted.

To describe the architecture on a lower level, we will trace the path of one sample table definition, from the model layer to the search indexes.

4.2 Model layer

Suppose we want to define a table of carbon nitrogen samples that has the following fields:

1. `ssn` - a unique sub-sample number
2. `total_nitrogen` - chemical value
3. `acidified_nitrogen` - chemical value
4. `acidified_carbon` - chemical value

The corresponding model layer definition would be the following:

```
class CarbonNitrogenSample(core.CabinetModel):
  ssn = SsnField()
  total_nitrogen = NonNegativeFloatField(blank=True, null=True)
  total_carbon = NonNegativeFloatField(blank=True, null=True)
```

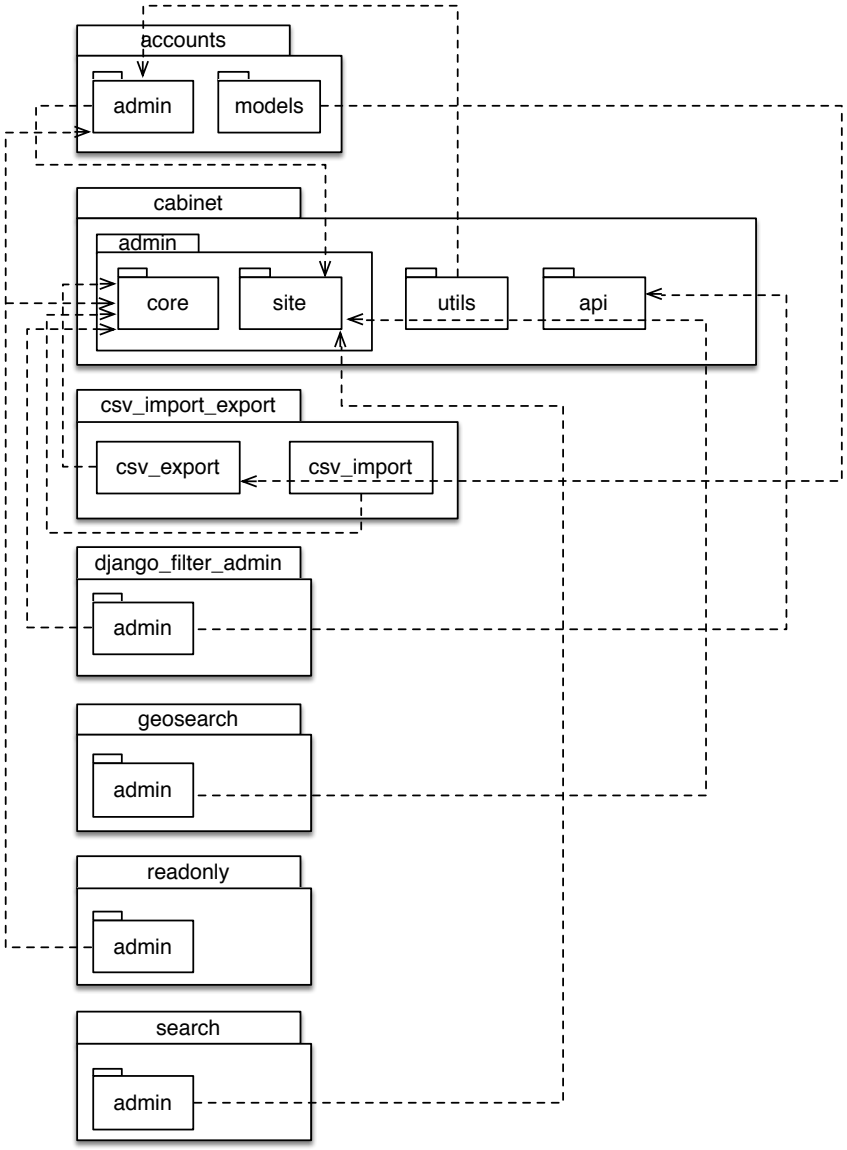


Figure 4.1: Extra-module dependencies between apps.

App	Description	Public classes
`accounts`	User-related functionality	`User`, `UserProfile`, `CustomUserAdmin`
`cabinet`	Functionality related to sample models, specific to the project	-
`csv_import_export`	Abstract handling of data in the CSV format	`CSVImportAdminMixin`, `CSVExportAdminMixin`
`django_filter_admin`	Abstract dynamic filtering	`FilterAdminMixin`
`geosearch`	Search across geographically referenced models in the administration site	`GeoSearchAdminSiteMixin`
`readonly`	Addition of the read-only permission to the administration site	`ReadOnlyAdminMixin`
`search`	Full-text search across models using Solr	`SearchAdminSiteMixin`
`tests`	Auxiliary definitions and methods for unit tests	-

Table 4.2: Apps present in AfSISDB

```
acidified_nitrogen = NonNegativeFloatField(
  blank=True, null=True)
acidified_carbon = NonNegativeFloatField(blank=True, null=True)
```

where `CabinetModel` is inherited from `Model`. Usage of `CabinetModel` accomplishes many things immediately:

1. the administration functionality of the model is activated, including:
   (a) import from and export to CSV
   (b) possibility of connection to the external data source (Mobilesurvey)
   (c) assigning model-related permissions (*change*, *delete*, *add*, *view*)
2. the API becomes accessible
3. model is able to store data from the external data source

If carbon nitrogen samples had geographic coordinates associated with them, we would define CarbonNitrogenSample as:

```
class CarbonNitrogenSample(core.GeoCabinetModel):
    ssn = SsnField()
```

```
    total_nitrogen = NonNegativeFloatField(blank=True, null=True)
    total_carbon = NonNegativeFloatField(blank=True, null=True)
    acidified_nitrogen = NonNegativeFloatField(
      blank=True, null=True)
    acidified_carbon = NonNegativeFloatField(
      blank=True, null=True)
```

`GeoCabinetModel` inherits from `CabinetModel` and includes all fields necessary for geo-referencing.

4.3 Model administration

The following piece of code ensures that any model inherited from `CabinetModel` and defined in the `cabinet` app has corresponding subclass of `ModelAdmin`:

```
for model in utils.get_geo_cabinet_models():
    admin_model_name = "{}Admin".format(model._meta.object_name)
    if admin_model_name not in globals():
        site.admin_site.register(
          model, core.SpreadsheetGeoSampleAdmin)

for model in utils.get_cabinet_models():
    admin_model_name = "{}Admin".format(model._meta.object_name)
    if admin_model_name not in globals():
        site.admin_site.register(
          model, core.SpreadsheetSampleAdmin)
```

As we can see, it registers either `core.SpreadsheetSampleAdmin` or `core.SpreadsheetGeoSampleAdmin`, which inherit from `core.CabinetAdmin`, for every model defined provided that there is no existing definition.

`CabinetModelAdmin` in turn inherits from `ModelAdmin`:

```
class CabinetModelAdmin(FilterAdminMixin, ReadOnlyAdminMixin,
                        admin.ModelAdmin):
    search_fields = ['ssn']
    ordering = ['-id']
    import_excluded_fields = ['id', 'updated_at', 'created_at']
    export_excluded_fields = ['id', 'created_at', 'updated_at']
    # ...
```

The full diagram of classes that register models in the administration site is shown in Figure 4.2. `SpreadsheetSampleAdmin` is used for text samples which require import and export in CSV, `SpreadsheetGeoSampleAdmin` supports sample tables with longitude and latitude columns; when such functionality is not required a sample model must be registered with `CabinetSampleAdmin` (or its subclass).

4.4 CSV import and export

From the definition of the `SpreadsheetSampleAdmin` it follows that every model inherited from `SpreadsheetSampleAdmin` acquires a possibility of its data to be exported or imported using CSV format. This functionality is provided by two mixin classes located in `csv_import_export` module - `CSVImportAdminMixin` and `CSVExportAdminMixin`.

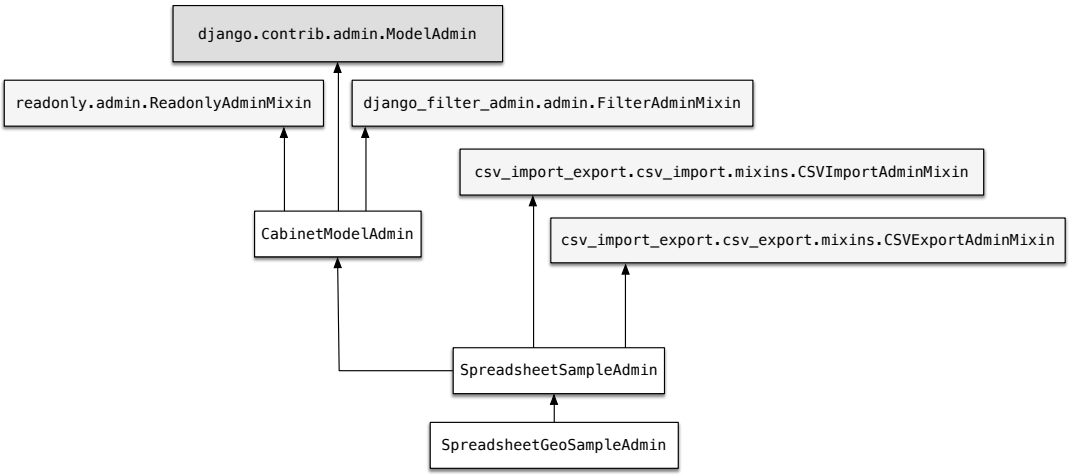


Figure 4.2: Classes which register models in the administration site in `cabinet` app. The base class `ModelAdmin` is shown in dark grey, abstract mixins from other modules are shown in light grey.

On a basic level, both mixins extend the URL configuration. For example, `CSVExportAdminMixin` overrides `get_urls` method in the following way, adding new URL point at `/cabinet/model_name/export/`:

```
def get_urls(self):
    urls = super(CSVExportAdminMixin, self).get_urls()
    info = self.get_model_info()
    return [
        url(r'^export/\$',
            self.admin_site.admin_view(self.csv_export),
            name='\%s_\%s_csv_export' \% info),
    ] + urls
```

It assign executions of a view `csv_export` whenever `/cabinet/model_name/export/` is accessed. Depending on the number of records to be exported the view either returns a CSV file by chunked transfer encoding[1] or starts an asynchronous Celery job. The maximum number of records that can be exported without the background processing is specified in the class field `export_streaming_limit` and can be overridden.

Importing requires several stages:

1. *processing* - the file was uploaded and currently is being processed
2. *processed* - the file was processed successfully
3. *importing* - the data from the processed file is being imported
4. *imported* - the data was imported
5. *error* - the file can not be processed

In particular, the distinction between *processing* of the file and *import* of the data from the file allows to present the would-be imported data to the user and

[1] `http://tools.ietf.org/html/rfc7230#section-4.1`

ask for the confirmation. The intermediate data is stored in a separate model called `ImportRow`, which has many-to-one relationship with `ImportTask` model. Their relationship and structure are shown in the Figure 4.3. When the file is being processed, `ImportRow`s are created with data written in the `data` column which, thanks to the `jsonb` type, can store row structure as dictionary. Once the import begins, their content is transferred to a corresponding model table passing through validation.

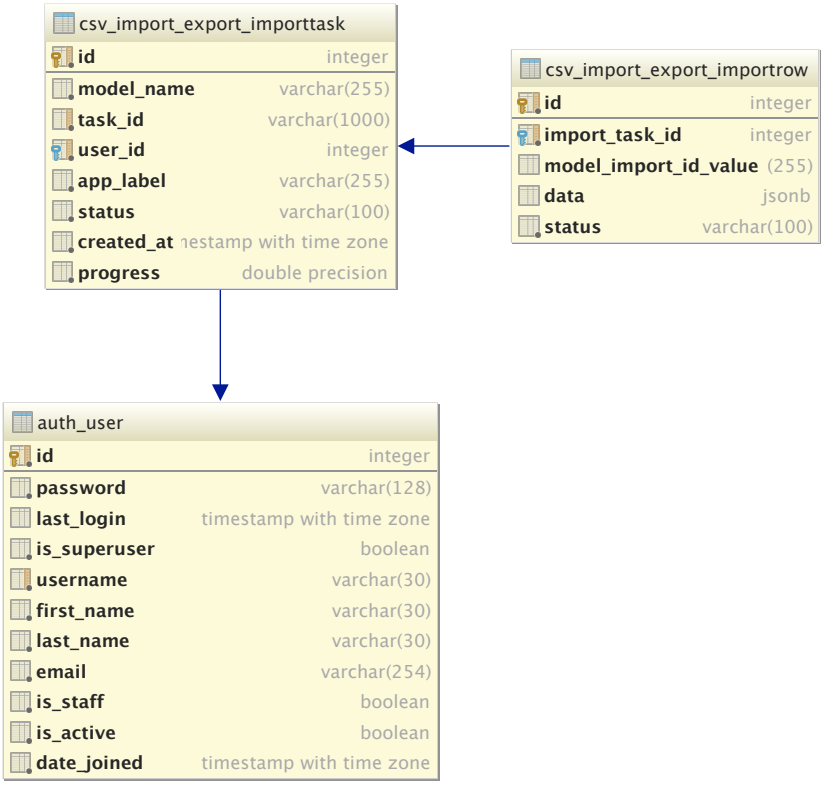


Figure 4.3: Underlying database structure of CSV importing.

4.5 Connection to external data source

The primary function of AfSISDB is to store information and while now it is mainly chemical decomposition of samples, in future AfSISDB will connect soil analysis, soil maps, drone imagery and other sources of data that help to answer the main question - what is the most efficient way of farming in Africa?

The already available external data source is the database of Mobilesurvey project[2] which provides a platform for geo-referenced data collection. Every soil sample that arrives to the lab was sampled somewhere in the region by a group that in addition answered several questions about the locality on Mobilesurvey. The answer data and the chemical data is matched by a field in the database and should be available on AfSISDB.

[2] `http://mobilesurvey.qed.ai/`

Such connection is specified by a model called `MobileSurveyRelation`, that stores the name of a survey form on Mobilesurvey and the corresponding model on AfSISDB, a field on Mobilesurvey and the matching field on AfSISDB:

```
class MobileSurveyRelation(models.Model):
    afsisdb_ct = models.ForeignKey(ContentType, unique=True)
    mobilesurvey_form = models.CharField(
      max_length=255, validators=[validate_mobilesurvey_form])
    mobilesurvey_tag_field = models.CharField(max_length=255)
    afsisdb_tag_field = models.CharField(max_length=255)
    #...
```

Every `CabinetModel` has methods to find its corresponding MobileSurveyRelation (the validation ensures that only one such connection exists) and fetch data from the survey form, by connecting to the MongoDB instance located on the Mobilesurvey server.

4.6 Model permissions

The `readonly` app provides a mixin `ReadOnlyAdminMixin` that ensures that the administration interface respects the *view* permission when allowing access to pages. Other permissions (*change*, *delete*, *add*) are handled by `ModelAdmin`.

4.7 Filtering

`FilterAdminMixin` adds filtering functionality to the sample model - records can be searched using multiple filters, each of them corresponding to one of the models' fields. In addition, fields that are defined as numeric and datetime data types can be filtered by maximum and minimum values; carbon nitrogen samples, for example, can be filtered to those that have total_nitrogen less or equal to 2.

Since `FilterAdminMixin` can be inherited from by any model, it has to generate filters dynamically. This is achieved by `filter_set_factory` method. This method returns an instance of the `base_filter_class`, with all filter fields created dynamically from the model definition. By default, `FilterSet` from an external package `django_filter`[3] is used, which already implements filtering.

4.8 API

For the API, Django Rest Framework[4] is used. It allows several formats of view definitions, and in our case we use dynamically generated `rest_framework.viewsets.ModelViewSet`'s.

Since scientists would like to filter by chemical values when fetching the data from the API, we again use `filter_set_factory` in the `ModelViewSet`, but now `base_filter_class` is `filters.FilterSet` - a class provided by Django Rest Framework and inherited from the `FilterSet`:

[3] `https://github.com/carltongibson/django-filter`

[4] `http://www.django-rest-framework.org/`

```
class CabinetViewSet(viewsets.ModelViewSet):
  filter_backends = (filters.DjangoFilterBackend,)

  def __init__(self, *args, **kwargs):
    self.filter_class = filter_set_factory(
      self.serializer_class.Meta.model,
      base_filter_class=filters.FilterSet)
    super(CabinetViewSet, self).__init__(*args, **kwargs)
```

As in case with `ModelAdmin`, `ModelViewSet` inherited from `CabinetViewSet` is generated for each sample model dynamically.

4.9 Geo-search

Since geo-search is performed across all models, it has to be implemented on `AdminSite` level, not on `ModelAdmin` level. With that in mind, `geosearch` app provides `GeoSearchAdminSiteMixin` that adds additional view (with a corresponding template) to the URL configuration.

Currently there are two types of geo-search: search inside a circle and search inside a bounding box. Search inside bounding box can be easily implemented without any extensions to the database, as it is done in `search_inside_box_across_models` method. Search inside a circle is more complicated, since computation of distance depends on the projection used to specify the coordinates. Thus it is better to perform it on the level of PostGIS extension which has `distance` (`D` in Django) function computed based on the projection of the table:

```
def search_inside_circle_across_models(models, longitude,
                                       latitude, radius):
    geom = Point(longitude, latitude)
    search_results_by_model_name = defaultdict(list)
    min_distances_by_model_name = {}

    for model in models:
        model_name = model._meta.model_name
        for i, obj in enumerate(model.objects.filter(
            geom__distance_lte=(geom, D(m=radius))
        ).distance(geom).order_by('distance').iterator()):
            # ...

    # Sort models by the minimum distance
    sorted_models = sorted(
        search_results_by_model_name.items(),
        key=lambda x: min_distances_by_model_name[x[0]])
    search_results = []
    for _, results in sorted_models:
        search_results += results
    return search_results
```

In `search_inside_circle_across_models` we also sort all records by their distance to the circle centre.

4.10 Search

The search is implemented using django-haystack[5] which communicates to the Solr server. Appropriate modification of the Solr configuration is specified in the `solr.xml` file.

In the same manner as geo-search, simple search operates across all models. Hence it also provides a mixin called `SearchAdminMixin` that is used to extend the default `AdminSite`.

django-haystack relies on search indexes to be defined in `search_indexes.py`. As before, there are generated based on model definitions and inherit from one main class - in this case, `CabinetSearchIndex`:

```
class CabinetSearchIndex(indexes.SearchIndex):
    text = indexes.NgramField(
      document=True,
      use_template=True,
      template_name='cabinet/cabinet_index_text_template.txt')
    ssn = indexes.NgramField(model_attr='ssn')
    ssn_exact = indexes.CharField(model_attr='ssn')

    def get_updated_field(self):
        return 'updated_at'
```

We add two fields corresponding to the `ssn` field. One is of `NgramField` type and will allow partial match with the search query [Solr, 2015], the other one is of `CharField` and will only allow exact matches. In this way, we allow partial matches, but records that have an exact match with the query will be ranked higher.

4.11 Conclusion

We have seen how from one definition of `CarbonNitrogenSample` the following functionality is automatically generated and become available to the users:

- Administration interface
- Export from CSV and import to CSV
- Synchronization with Mobilesurvey
- Filtering of records by individual fields
- API
- Search based on geographical coordinates
- Search of samples by `ssn`

[5] `http://haystacksearch.org/`

5. Testing

The project was tested by unit tests with coverage of 96%.

The project was also tested by a Q.A. Engineer who went over the scenarios corresponding to the requirements for each sample table. The results are recorded in the spreadsheet described in the Attachment II. The following features were tested:

1. Registration and authentication
2. Search (by SSN, inside circle and inside bounding box)
3. For each sample table the following tests were performed:
   (a) Search inside table
   (b) Importing from CSV
   (c) Exporting to CSV
   (d) Creation of new sample
   (e) Edition of a sample
   (f) Recovery of deleted samples
4. For "Spectroscopy OPUS records" table the following additional tests were performed:
   (a) Uploading of a binary file
   (b) Hiding samples by making them private
5. Connection to Mobilesurvey
6. Filters
7. API

The whole test suite is available in the Attachment II, but is too large to be included here. It brought to light some gaps in filters (users are able to specify invalid values and use multiple filters for the same field), as for the rest, the application was found to function properly.

6. User Guide

6.1 Registration

AfSISDB registration is filtered by the administration. The user must send a request from the login page explaining their affiliation and motivation for the registration. Once the request is accepted, the user receives an invitation to log in.

6.2 Inserting data

There are two types of data that can be inserted - text data in form of CSV files and binary data.

Samples that support binary data have the button "Upload binary file" on its page. This button leads to the page where files on the local computer can be selected and uploaded. Successful uploading results in the creation of a new record in the table and transferring of the file to the S3 bucket (or, in a general case, to the file storage specified in the settings).

A sample can be chosen to be private or public. Private means that while other users will be able to see its presence, sensitive data (the binary file) won't be available for download and the binary file in the bucket will not be accessible. Conversely, public samples display the link to the key in the bucket, which is opened for reading.

If the upload is successful, new record is displayed in the table below, the link to the administration interface and the link to delete the record become available. If the upload has failed, an error message is displayed. It either says that the sample is a duplicate of an already existing one, or it displays the error returned by the script that processes binary files. An example is shown in Figure 6.1.

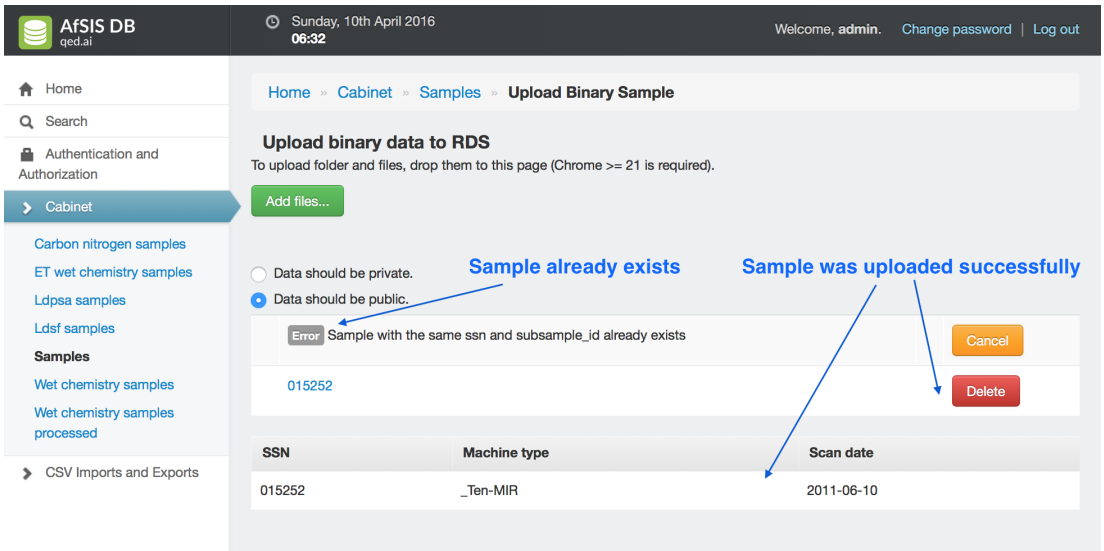


Figure 6.1: Upload page for binary samples.

Samples that support CSV import have "'Import" button on their page. This

button leads to the first page of the import process - selecting a CSV file with the correct fields (which are displayed in the hint in the form). After the correct file is selected, the importing process starts and the user can see its progress. If the user agrees to import the processed data, they must click the "Confirm import" button. Only after that the records begin to be created and their status is displayed to the user. The final number of updated, created and failed records is shown in the end. The user can also select to see only updated, created or failed records. A record is updated if a record with the same ssn already exists. An example is showin in Figure 6.2.

A link to each import is later accessible in "CSV Imports and Exports", as showen in Figure 6.3.

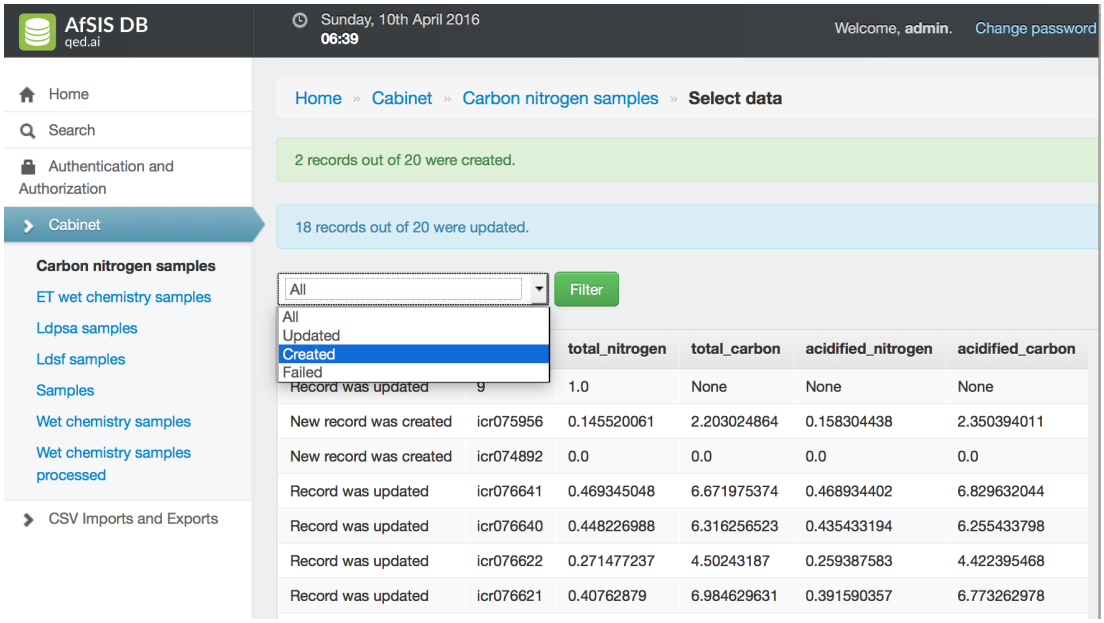


Figure 6.2: Import page for text samples.

6.3 Accessing data

Each table is presented as a list of records, with links to each record's page. Alternatively, records can be accessed through the API. All access points of the API are listed on `/cabinet/api-docs/` and can be interactively tested there. In short, for each table there is an API point at `/cabinet/api/table_name/`, where *table˙name* is the name of the table without spaces and underscores. For example, GET request for carbon nitrogen samples, which by default returns all available to the user samples, should be made to `/cabinet/api/carbonnitrogensample`. `POST` method is available at the same `URL`. `GET`, `PATCH` and `DELETE` are accessible at `/cabinet/api/table_name/ssn/`, where *ssn* is the ssn value of the required record.

`GET` request also accepts dynamic parameters. For each numeral field in each table, records can be filtered by two parameters: max_*field* and min_*field*. For example, to find all carbon nitrogen samples with acidified carbon at least 2 and acidified nitrogen at most 200, the following query must be used:

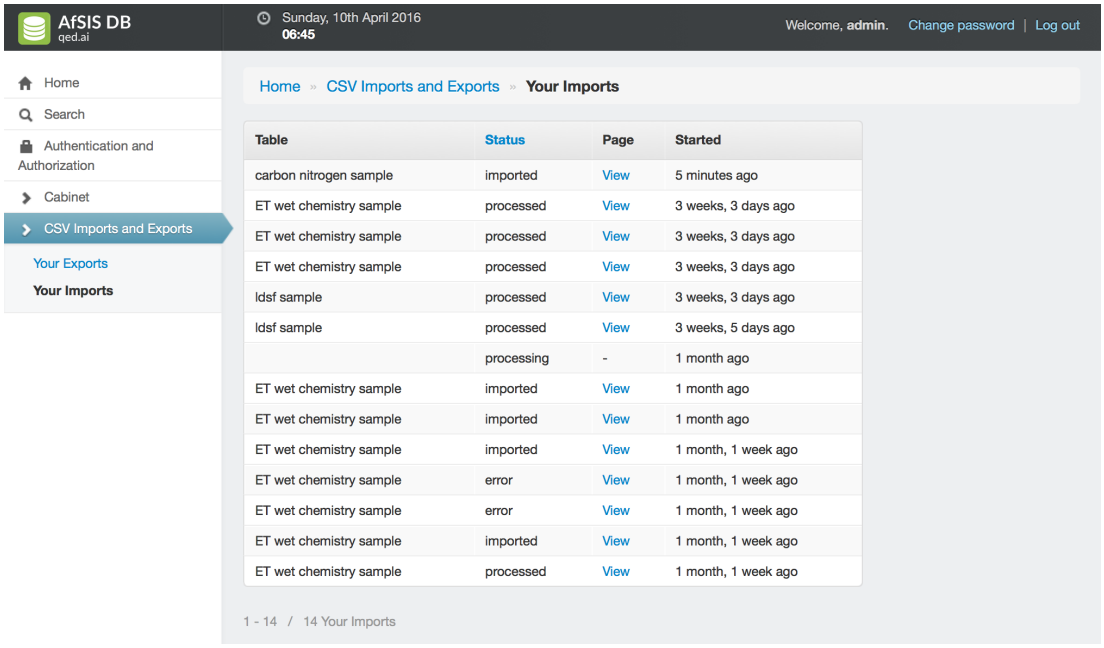


Figure 6.3: User's imports.

```
/cabinet/api/carbonnitrogensample/
?min_acidified_carbon=2&max_acidified_nitrogen=200
```

In this way, geographically referenced samples can be filtered to be contained in a latitude-longitude box.

6.4 Searching data

The data can be searched by ssn, with optional selection of tables that should be included in the search. The search field is located in the left sidebar.

Moreover, samples that are georeferenced can be searched inside a circle, specified by radius, latitude and longitude and a bounding box. The geo-search page is accessible through "Geo Search" link.

6.5 Filtering

On each table's page, filtering is available in the right sidebar. An example of its usage is showen in Figure 6.5.

6.6 History

For each sample, a history of changes is kept, with the possibility to revert to the previous version. It is available on a sample's page under the link "History". There user can see a list of all versions with the short description of the action (i.e. which fields were updated in case of update) and the user responsible for the action. After clicking on a particular version, the user is able to choose to revert to that version. The reversion is considered to be one of the actions and kept in the history as well.

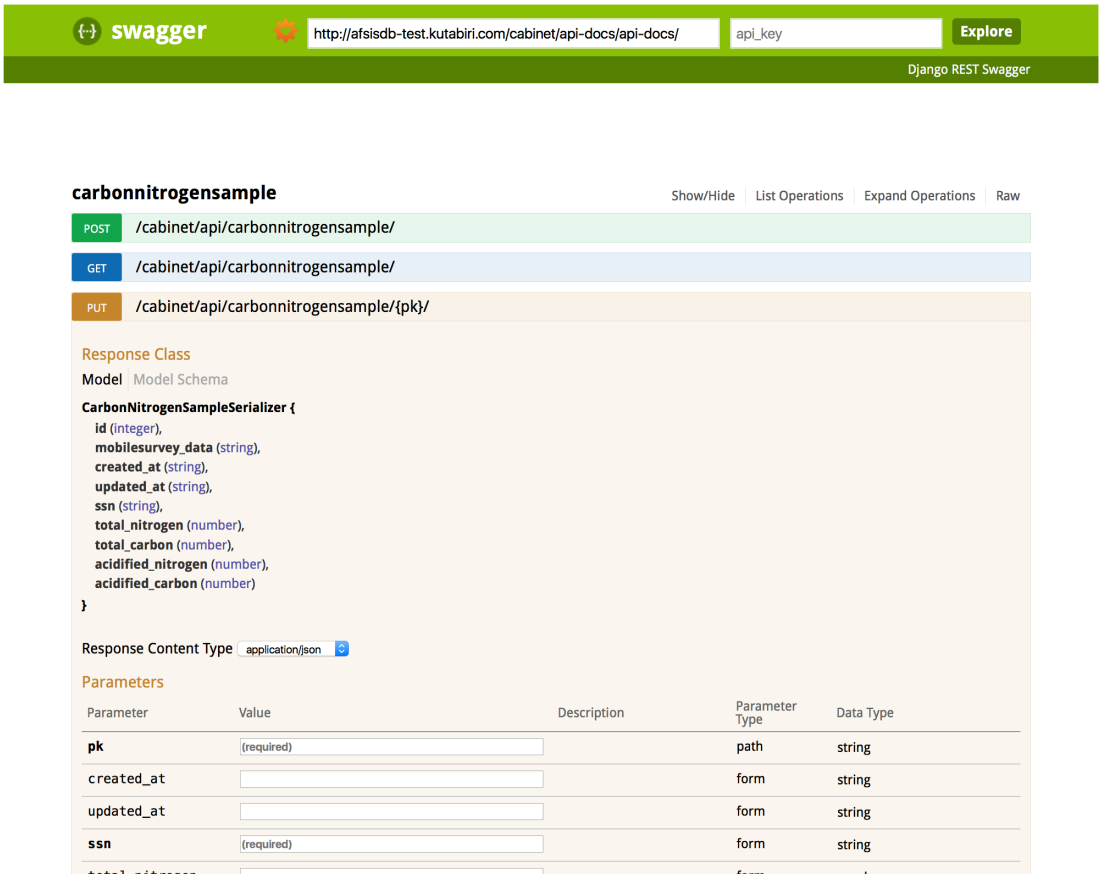


Figure 6.4: API points are listed at `/cabinet/api-docs/`.

Deleted records can be restored from the table's page, under "Recover deleted" link.

6.7 Connection to Mobilesurvey

The table's samples can be synchronized with data on Mobilesurvey on "Connect to Mobilesurvey" page. There the user selects the name of the form on Mobilesurvey, the field of the form and the field in the table that should be matched. After that, each time a sample is accessed (either through web interface or through API), `mobileusrvey_data` is updated in the record. Later the connection can be edited on "Edit connection to Mobilesurvey" page.

6.8 Adding new sample tables

To add new samples to the website, the user must edit `cabinet/models/tables.py` file, for example, by adding the following definition of `CarbonNitrogenSample`:

```
class CarbonNitrogenSample(core.CabinetModel):
  ssn = fields.SsnField(unique=True)
  total_nitrogen = fields.NonNegativeFloatField(
    blank=True, null=True)
  total_carbon = fields.NonNegativeFloatField(
    blank=True, null=True)
```

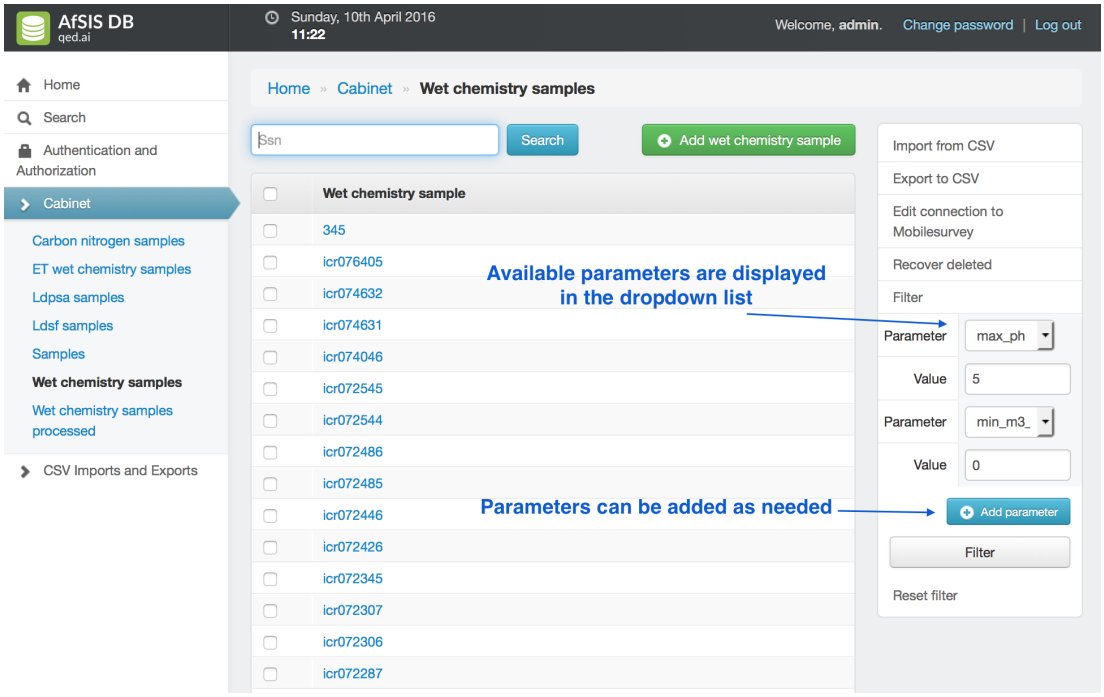


Figure 6.5: Usage of filters.

```
acidified_nitrogen = fields.NonNegativeFloatField(
  blank=True, null=True)
acidified_carbon = fields.NonNegativeFloatField(
  blank=True, null=True)
```

Description of all available fields is available in the Django documentation on model fields[1], additional fields are listed in `cabinet/models/fields.py`.

The next steps depends on the deployment process. In general, the change made to `cabinet/models/tables.py` should appear on the server and the standard procedure of deployment must be performed (e.g. restarting application server), plus extra steps specific to the project (see Keeping Solr up to date, Update) – this can be automated using an automation engine like Ansible[2].

[1] `https://docs.djangoproject.com/en/1.9/ref/models/fields/`
[2] `https://www.ansible.com/`

7. Conclusion

The developed system satisfies the requirements stated in the chapter "Analysis of the Requirements and the Proposed Solution". It allows to manage data sets of samples divided into tables, each with its own structure through importing to and exporting from the CSV format, searching, filtering and accessing through the API. New sample tables can be added by creating one class and updating the server through a standard deployment procedure for a Django application.

The developed system is being used by scientists and laboratory assistants in Tanzania. Currently there are 24 users and 112,666 samples uploaded in total.

AfSISDB already provides a reliable storage system and there are more features to implement in the future. The work on a more adaptable permission system that will allow binding of samples to groups is in progress, and association of samples and measurements taken in a specific laboratory (instead of one unique sample only) is going to become available in the near future.

AfSISDB is supposed to become a central place for all information gathered by AfSIS project – it should contain not only chemical information, but references to information about locality (Mobilesurvey), drone imagery and further predictions about the quality and effective usage of soil.

Once the project passes the initial release stage, it will be open-sourced. That is why modularity and extensibility of the current architecture is very important. It is hoped that projects similar to AfSIS will find benefit in using their version of AfSISDB.

Bibliography

Django. Django Documentation. Available online at https://docs.djangoproject.com, 2015a.

Django. Faq: General. Available online at https://docs.djangoproject.com/en/1.9/faq/general/#faq-mtv, 2015b.

Haystack. Haystack v2.4.1 Documentation. Available online at http://django-haystack.readthedocs.io/en/v2.4.1/index.html#getting-started, 2015.

Apache Solr. Analyzers, tokenizers, and token filters. Available online at https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters, 2015.

The Earth Institute of Columbia University. New Grants to Extend Reach of Africa's Green Revolution. Available online at https://www.earth.columbia.edu/articles/view/2978, 2012.

The United Nations Food and Argiculture. The state of food insecurity in the world 2015. Available online at http://www.fao.org/3/a-i4646e/index.html, 2015.

List of Figures

List of Tables

List of Abbreviations

AfSIS African Soil Information Service 3

AfSISDB African Soil Information Service Database 4

MTV Model-template-view 13

Attachments

Attachment I

Name of the file: opusupload_checks

Format: Linux executable

Description: A C++ script developed by AfSIS software development team that extracts metadata – SSN, subsample_id, scandate, machine – from a binary file created by a spectrometer.

Attachment II

Name of the file: Testing Matrix

Format: OpenOffice Spreadsheet

Description: A spreadsheet maintained by a QA Tester, which keeps record of the status of features according to the requirements.

Index