CoADD Coding Guideline

|  |  |  |
| --- | --- | --- |
|  | **DOCUMENT PROPERTIES** | |
| Document Title: |  | |
| File Name: |  | |
| Version: | v0.5b | |
| Status: | draft | |
| Date: | 04/02/2023 | |
| Author: | Joanna Zhong | |
|  |  | |
|  | **REVIEW / APPROVAL** | |
| Reviewed by: |  | Date: |
| Approved by: |  | Date: |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **CHANGE HISTORY** | | | |
| **Version** | **Date** | **Description of action** | **Pages** |
| V0.2 | 04/11/2022 | Prerequisites add documents,  Section2,3,4: add naming rules, changing programming structure, adding tests | P 2, 4, 5, 6, 9 |
| V0.3 | 10/01/2023 | Updating specific construcion | P8-15 |
| V0.4 | 12/01/2023 | Updating Section 1 directories | P2-6 |
| V0.5 | 29/01/2023 | Updating section1, section5, adding append1 | P4, p10 |
| V0.5b | 04/02/2023 | Updating section 5: uploading file | P18 |

Table of Contents

[Coadd Coding Guidelines - 5 -](#_Toc127376980)

[Prerequisites - 5 -](#_Toc127376981)

[Documents: - 5 -](#_Toc127376982)

[Project Overview: - 5 -](#_Toc127376983)

[Section 1. Project Directories - 5 -](#_Toc127376985)

[1 Project level - 5 -](#_Toc127376986)

[2 Applications - 6 -](#_Toc127376987)

[3 Static, Templates, Docs - 6 -](#_Toc127376988)

[Section 2. Naming Rules - 6 -](#_Toc127376989)

[1 Variable Name - 6 -](#_Toc127376990)

[2 Project level naming - 6 -](#_Toc127376991)

[3 Application-level naming - 7 -](#_Toc127376992)

[Section 3. Programming and Code Blocks guidance - 8 -](#_Toc127376993)

[1. Import packages, Environment variable and constants - 8 -](#_Toc127376994)

[2. Modules and Packages - 8 -](#_Toc127376995)

[3. Model class construction - 9 -](#_Toc127376996)

[4. Form class construction - 9 -](#_Toc127376997)

[5. Views for models’ construction - 10 -](#_Toc127376998)

[6. URLs - 11 -](#_Toc127376999)

[7. Template and Js - 11 -](#_Toc127377000)

[Section 4. Functional testing: - 13 -](#_Toc127377001)

[Section 5. Specific construction - 14 -](#_Toc127377002)

[1 Concurrency - 14 -](#_Toc127377003)

[2 Ldap login - 14 -](#_Toc127377004)

[3 multi\_schema set up in postgresql and Django project - 18 -](#_Toc127377005)

[4 Construct a group of customized editable tables for detail view - 20 -](#_Toc127377006)

[5 htmx editRow for list/table view - 21 -](#_Toc127377007)

[6 DataTable with children-DataTables - 23 -](#_Toc127377008)

[7 Data import - 24 -](#_Toc127377009)

[8 Vitek-import - 29 -](#_Toc127377010)

[9 Modelfilter, Filterbaseview - 33 -](#_Toc127377011)

[10 Data visualization - 33 -](#_Toc127377012)

[Append 1 project directory tree view: - 33 -](#_Toc127377013)

# Coadd Coding Guidelines

Coding guideline is to ensure the code is consistent, minimize the errors and help maintain easily, which provides guidance for building a project and functionalities structure, naming rules and coding reusable code blocks in different section. Also provides certain constructions explanation in the last section.

## Prerequisites

### Documents:

* Django Documents for name convention
* “GetStartedWithCoadd”: Computer setting, Environment setting
* *“Orgdbapp\_Model-View-TemplateMap”*
* “OrgDB Functional tests List and Issue Log”

### Project Overview:

## 

Running on server by manage.py

## Section 1. Project Directories

The Directory includes project level and application (whole directory see append 1).

### 1 Project level

Project root directories contains:

1. core package(adjCOADD) configuration global constants: variables, path, packages, collecting apps and urls in other applications.
2. application package apputil
3. application package dorganism
4. applications package ddrug
5. root folders: static, templates and z\_doc
6. manage.py command to execute and interact with project e.g. run development server and data migrations.

### 2 Applications

Inside of each applications package, there are basic modules: models.py, forms.py, views.py and urls.py for building an interaction with database. Also, there is a templates folder contains Web UIs associated with each model.

1. **models.py** contains model classes for creating data tables in DB and methods for database operation.
2. for most of model classes Django ModelForm **in forms.py** are used for create/update rows in DB.
3. the interaction between users and database are controlled by classes or functions in **views.py.**
4. **urls.py** access or call each control functions and classes in views.py by web input, click, drag actions.

Apart from the above basic modules, each application will have a few more functional modules:

1. util.py and utils\_dataimport.py(only for apputil)
2. <application/models name>.py which are used to provide functions or classes to perform a range of certain functions (data import, validation log, parse pdf or csv files etc..
3. templates/<appname>/<model name>… folder contains templates of CRUD views and data visualization views per each model

### 3 Static, Templates, Docs

Static folder contains front programming resources, in css, js, images 3 folders.

Templates folder contains base.html and utility html files used to extends by and included in other app templates.

z\_Docs folder is where to write readme, help documents.

z\_log folder contains debug information, which can be tracked when project running in product stage on the server.

## Section 2. Naming Rules

### 1 Variable Name

* General Variables:
  + context: variable collection will be sent to html template in render
  + form.errors: build-in variables hinting class Form errors.
  + i or item: item in iterable collection
  + instance: model instance used in function based create, update views, class forms
  + objects: a queryset objects from a model, used to put query results to html template.
  + object\_: a single model query result
  + qs: queryset
  + req: request
  + \_fk: indicate Foreign Key type
  + \_list, \_tup, \_dict...: variable is list, tuple and dictionary... type
  + page\_num, page\_obj, p(pagination): for Pagination class
  + form: represent Form Class
* Special variables will be commented in Functions and Classes

### 2 Project level naming

A project name starts with a\_, therefore the core file name will be with "a\_..." positioned on the first line under project root. For applications' name following the below:

* utility application (for admin management, functions sharing...purpose) starts with "a"
* django applications start with "d" other django files and files following django and python name convention: templates, static, uploads...

### 3 Application-level naming

Applications' module name following django and python name convention: admin.py, models.py, utils.py .... In coding process following the below rules:

* Model naming:
  + class names in models.py are in Singular
  + Field Names can contain an underscore starting with a capital letter
  + CharField with max\_length: 10, 15, 25, 50, 125, 150, 200, 220, 500 ...
  + Model variable is in camelcase, starts with "semantic word(purpose) in small letters", follows with referred model field name if exists, ends with dataType.
  + ForeignKey setting: **need to discussing with DO\_Nothing or Set\_Null??**
* Forms name is in CamelCase with (form purpose if needs to do: Create, Update...) (Modelname)\_form
* View Functions variable names
  + Views in Class will be defined with (Modelname)(Functions: List/Card/Update/Create...)(View); Views in Function will be defined with (Modelname in small letters)(Functions: List/Card/Update/Create...)
  + For django build-in function variables following django convention e.g., for pagination, using pag\_num ...
  + An objects collection or object of model instances, which need to pass to template, will be defined with **objects / object\_ (model name in small letter in template)**
  + For other variables and bridge variables passed to Templates/Model instance/Forms or received from Templates input in camelCase. with belonged model name at the beginning all letters in small case with following semantic word (capital letter at the beginning) and ending with data type e.g., organismNameFkChar
* URL names:
  + Pass primary key by using .../<int:pk> or .../<str:pk>. refer to django document [link](https://docs.djangoproject.com/en/4.0/ref/models/instances/#the-pk-property).
  + path defined with /(projectname)/(modelclassname)\_(overview\_list or card)/detail or/delete or/update) or (/ID\_name) except homepage with /(projectname)/home
  + URL name defined with: (model name) (\_functions:list/card/create/update/delete/detail)
* Templates names defined with: (model name)/(functions name(create or update or …)) in small letters
  + with /(projectname)/(modelclassname)/(overview\_list/detail/delete/update) or (ID\_name) except homepage with /(projectname)/home
* JS files name and variables name...

## Section 3. Programming and Code Blocks guidance

This provides guidance where to program each module, functions, classes, and the general rules applied in the programming, for some detailed programming refer to section5.

### 1. Import packages, Environment variable and constants

In general, importing packages and modules are on the top of a python module file and from up to down, there are 4 sections:

1. Other system and python packages, e.g. import rdkit;

2. Django build in package, e.g. from django.contrib import messages;

3. From other project application packages, e.g. from app.models import Dictionary;

4. Import modules from the same package, e.g. from .utils import \*.

All the basic settings, environment variables and directory path constants are in settings.py, while for model constants are in constants.py in the same core package(adjcoadd).

### 2. Modules and Packages

The core package(adjcoadd) is to collect and organise all application packages in the project. It has modules:

│   ├── asgi.py

│   ├── constants.py

│   ├── \_\_init\_\_.py

│   ├── prod\_settings.py

│   ├── routers.py

│   ├── settings.py

│   ├── urls.py

│   ├── utils\_dataimport.py

│   └── wsgi.py

Further explanation:

* \_\_init\_\_.py is initiated.
* asgi.py and wsgi.py are only used in product phase.
* To program model constants in constants.py
* To config project settings (templates/media/static paths, middlewares, backend-authen & Ldap backend, organise applications, debug, database settings, deployment settings, security, timezone) in settings.py.
* To program utils functions and classes for the other applications in utils.py and especially for the import file in utils\_dataimport.py
* To program modules for using multiple databases/schemas in routers.py and refer the modules in settins.py
* Prod\_settings.py is another settings.py only for product phase.

The application package: apputil, provides other applications utilities, which contains: ApplicationUser, Audit, Dictionary, ApplicationLog modules which accessible by all other applications. So, utility functions will be used for each other applications programming here, e.g., def slugify or class Filterbase. And each model’s views, templates, urls.

Other application packages: dorganism, ddrug…, are used to connect and perform operations with the specific databases/schemas. Each package will include programming class models represented tables in databases/schemas and views UI templates and urls for interactive operations in databases.

### 3. Model class construction

In models.py of each application package except apputil contains database models.

The class ApplicationUser in apputil is inherited from the Abstract Model.

The class AuditModel is models.Model, while all other database models are inherited from the class AuditModel. Therefore any methods and class variables needed for other database models will be programmed in AuditModel as maximum as possible.

### 4. Form class construction

* Using ModelForm to inherit models’ fields.
* Avoid making programs in Views Function and move these to Form class methods(\_\_init\_\_, get\_..., clean…, validate...) e.g. (Exceptions: Problem of passing variable from Views to Forms’ methods):

class CreateOrganism\_form(ModelForm):

Organism\_Desc=forms.CharField(widget=forms.TextInput(attrs={'class': 'form-control'}), required=False,)

…

Oxygen\_Pref=forms.ChoiceField(choices=(('',''),('','')), widget=forms.Select(attrs={'class':'form-select', 'id':'Oxygen\_Pref\_choice'},))

…

Organism\_Name=forms.ModelChoiceField(queryset=Taxonomy.objects.all(), widget=forms.HiddenInput(),required=False,)

def \_\_init\_\_(self, Organism\_Name=None, \*args, \*\*kwargs):

self.Organism\_Name=Organism\_Name

super(CreateOrganism\_form, self).\_\_init\_\_(\*args, \*\*kwargs)

Strain\_Type\_choices=querysetToChoiseList\_Dictionaries(Dictionaries, Organisms.Choice\_Dictionaries['Strain\_Type']) #

self.strainTypeChoices= Strain\_Type\_choices

self.fields['Strain\_Type'].widget = forms.CheckboxSelectMultiple(choices=self.strainTypeChoices)

self.fields['Strain\_Type'].widget.attrs.update({'class': 'special'})

self.fields['Oxygen\_Pref'].choices=querysetToChoiseList\_Dictionaries(Dictionaries, Organisms.Choice\_Dictionaries['Oxygen\_Pref'])

…

def clean\_Organism\_Name(self):

data=self.cleaned\_data['Organism\_Name']

data=get\_object\_or\_404(Taxonomy, Organism\_Name=self.Organism\_Name)

return data

class Meta:

model=Organisms

exclude = ['Organism\_ID']

### 5. Views for models’ construction

In views.py of each application, for simple model Class (only defined with model fields) using GenericView for list/card View. Detail, Create, Update and Delete view using function-based view (at this stage). For complexed model Class list/card View may use Function based view. Keep Views functions as minimal.

* function based view functions for each model class:
  + example (create new entry):

def createOrgnisms(req):

'''

Function View Create new Organism table row with foreignkey: Taxonomy and Dictionary.

'''

kwargs={}

kwargs['user']=req.user

if req.method=='POST':

Organism\_Name=req.POST.get('searchbar\_01')

Strain\_Type\_list=req.POST.getlist('Strain\_Type')

form=CreateOrganism\_form(Organism\_Name, req.POST)

print(f"request.Post.get {Organism\_Name}")

try:

if form.is\_valid():

print("form is valid")

try:

with transaction.atomic():

instance=form.save(commit=False)

instance.save(\*\*kwargs)

print("saved")

return redirect("org\_list")

except IntegrityError as err:

messages.error(req, f'IntegrityError {err} happens, record may be existed!')

return redirect(req.META['HTTP\_REFERER'])

else:

print(f'something wrong...{form.errors}')

return redirect(req.META['HTTP\_REFERER'])

except Exception as err:

print(f'error is {form.errors} with {err}')

return redirect(req.META['HTTP\_REFERER'])

else:

form=CreateOrganism\_form()

return render(req, 'aa\_chem/createForm/Organism.html', { 'form':form, })

* class based view functions( for displaying model database records):

class OrganismListView(LoginRequiredMixin, ListView):

model=Organisms

template\_name = 'aa\_chem/readForm/Organism\_list.html'

paginate\_by=3

def get\_context\_data(self, \*\*kwargs):

context=super().get\_context\_data(\*\*kwargs)

context['filter']=MySearchbar03(self.request.GET, queryset=self.get\_queryset())

return context

def get\_queryset(self):

queryset=super().get\_queryset()

return MySearchbar03(self.request.GET, queryset=queryset).qs

And for other class view just need to inherit as the following:

class OrganismCardView(OrganismListView):

template\_name = 'aa\_chem/readForm/Organism\_card.html'

* "Debug prints" will be used in development stage, some will be deleted in production stage.
* If only to find a queryset has at least one recorder, using querySet.exists() instead of querySet. e.g. replace if querySet to if querySet.exists() [link](https://docs.djangoproject.com/en/4.1/ref/models/querysets/#django.db.models.query.QuerySet.exists)

### 6. URLs

### 7. Template and Js

Templates and Js are wrapped in the project, distributed in templates folders and static folder. For detailed JS functions refer to [here](#_Append_2_JavaScript).

The templates are HTMLs with Django settings (refer to settings.py):

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [

BASE\_DIR/'templates',

# BASE\_DIR/,

],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

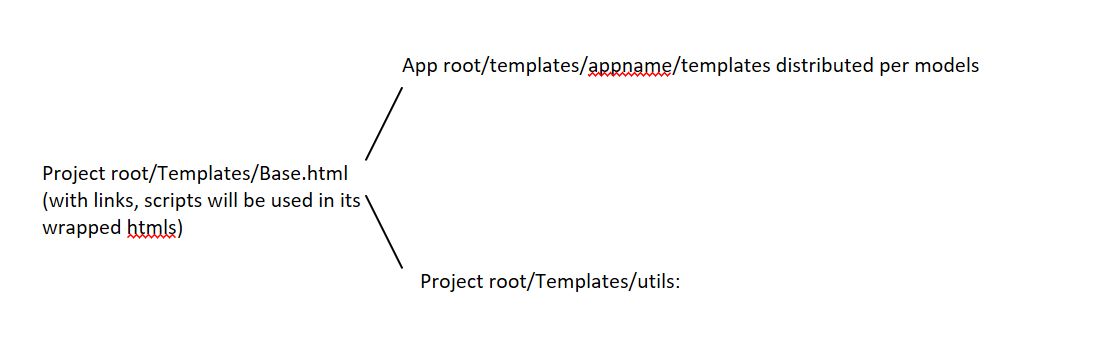
],

},

},

]

Are ablet to render information passed by views.py and run some logic in template languages. The templates’ structure is:

Programming nav, datatable(for listview), cards, filter, sidebar etc. styles in templates/utils/… are generally used and included in App root/templates/appname/datamodel-name/ or ../datamodelname/Foreignkey datamodelname/… , in this approach all views’ displays could keep the same style and save amount of codes. In app root contains card/list-filter, create, update and delete views templates for each data model.

Frontend programming is used to manipulate the Html elements and process data on the HTML pages, which is dominated by jQuery and JavaScript. A quite of few packages import directly in base.html and individual html pages e.g. HTMx, jQuery UI, etc..

In general, to manipulate html elements using jQuery selectors and events functions.

For example:

….

To send and receive data using Ajax jQuery and after receiving data manipulate DOM will be wrapped in the async function .done(()=>{})

#### Handle data on HTML

Ajax call sendDatatoServer and receive Data to display or operate further in the HTML. A Ajax call is :

$.ajax({

url: "whereTopost", // the endpoint

type: "POST", // http method

data: { new\_data: $("#{some FormName}").val() }, // data sent with the post request

// handle a successful response

success: function (json) {

$("#{some FormName}").val(""); // remove the value from the input

console.log(json); // log the returned json to the console

console.log("success"); // another sanity check

},

// handle a non-successful response

error: function (xhr, errmsg, err) {

$("#form\_result").html(

"<div class='alert-box alert radius' data-alert>Oops! We have encountered an error: " +

errmsg +

" <a href='#' class='close'>&times;</a></div>"

); // add the error to the dom

console.log(xhr.status + ": " + xhr.responseText); // provide a bit more info about the error to the console

},

});

## Section 4. Tests:

There are two parts of tests unit test and functional test.

For unit test including:

1. Urls test
2. Model and Views test

The test is coded in each application per each view and model.

For functional tests including:

1. Filtering Function Test
2. Creating / Updating / Deleting Test
3. Ladp login test
4. Other new function Test (if implemented)

The test is via black box test and referred to the ‘test and Issue Log’

## Section 5. Specific construction

### 1 Concurrency

To solve multiple users access data entries problem (concurrency), using django build\_in with transaction... select\_and\_update.....

### 2 Ldap login

In project, the Ldap authentication system is to used for login. To perform this, a few configurations need to add in settings.py and create corresponding forms and views functions.

Step 1 in settings.py, use the following code to config authentication backend:



Step 2 Database model (Authentication Model) construction based on ldap name and password:



Step 3 build login form:



Step 4 URL for users to click on login/out:



There are two urls for login purpose. The blank one ‘ ’ is used to login before landing pages while another one is used during process when session expired or permission denied.

Step 5 Build views functions called by URL:



### 3 multi\_schema set up in postgresql and Django project

In postgresql, schema public is a default schema with rdkit extension installed and shared by other schemas. Schema apputil, ddorganism and ddrug contain data tables for data models built in application apputil, ddorganism and ddrug respectively. Adding a new application in the project will have a new schema created in the postgresql.

The function DatabseRouter in adjcoadd.routers.py will manage each application’s migration database.

Database configuration in settings.py is:



With Transaction level-serializable isolation.

### 4 Construct a group of customized editable tables for detail view

Currently only used to render single organism object updating view in “organism detail” template.

On template side, the first step is constructing a group of tables wrapped by Django updating form and in the tables the object values display firstly on cells and the updating form fields hide in the corresponding cells. The lines of codes are presented below:



Then, in JavaScript implement dbclick eventHandler function to switch between displaying information and updating object field values cases through double click each cell.

There are two blocks of JavaScript codes for the input/single-select form fields and the multi-select form fields respectively:



(The first block is for the input/single-select form fields and the second for multi-select form fields.)

The advantage of this technique is a conveniently utilising of Django updating form, to display and update object information in a customized group of tables with a few lines of JavaScritp code.

The shortcoming is some hard coding needed in a template.

### 5 htmx editRow for list/table view

Htmx editable rows are used in tableview to edit rows. To perform the function Htmx package has been implemented in base.html:



, which can be used in ApplicationUser’s templates wrapped by base.html. A coding example is displayed below:

Example: Rendering ApplicationUser update view:

step 1 – on UI side to construct the table  
In this step, the key parts are: tbody – for displaying information, and tr/td– make the row editable and tr–updating with form fields.  
An example is displayed below with highlighted kern parts:

– the table with display “tbody” and “td”(contained editable button)

  
- the tr with updating form:



The tds and their positions in the displayed row should be matched to these updating form fields.

–  step2 on Server-side to establish the responses  
After  table construction then we need view function to perform the editable function which means return the http response to editable actions.  
The following one is the corresponding example function:



### 6 DataTable with children-DataTables

The Children-DataTables generated by Javascript receiving data from server via Ajax Call.

Following are code block for generate stock table under batch Table in datatable\_batch.html:



### 7 Data import

The general file import modules Scope View:

Graphical user interface, text, application, email

Description automatically generated

For each application package, the data import via file uploading module contains: Part I general modules and Part II Individual modules based on each Model.

The code scope covers across apputil/utils\_dataimport.py, js/importhanlder.js, templates/utils/importhandler.html to:

1) itself special model check methods in <app>/models.py

2) file parse and generate model objects functions in <app>/<model name>.py (e.g., vitek.py) and

3) class-based view in <app>/views.py.

Therefore, to construct data import/file upload module, there are two parts:

**Part I.** The Importhandler class, Validation log functions, Templates and JavaScripts are reusable.

**Part II.** For itself model check methods are class method to check foreign key and validate data field errors. The file parse function is used to read pdf files and extract content to make an object list which will be validated to be able to import to DB. And the view class will control the file upload, file validate, and data import to DB steps as server responses to user input. These functions and classes should be constructed per different models.

PART I -a. The Importhandler Class and Validation log Class

Importhandler Class is basically a django view class with defined class variables, instances methods: def get & post, def validates, def delete\_file.

The class variables are generally applied for all type of data import in the following usages:  
- form\_class to submit uploaded files, which is Django form class with the file field and a customized file validator (class FileValidator(object) in apputils/utils\_dataimport.py) to check uploaded file type, size, etc..

-file\_url to store list of files uploaded in server, which also will be sent to display on FrontUI(HTML web page)

-data\_model is to indicate model type(organism, vitek, drug…)

-validate\_result is to store new objects validate status, which are parsed and built from uploading files

-file\_report is to store each files validation report from Validation log Class

The methods except def post will be specified due to different model

PART I -b. Templates and Javascripts

The import template, project\_root/template/importhandler.html, contains:

1. Section 1 “Import\_step1” uploading files, if file errors occur, next Section cannot display.
2. Section 2 “Import\_step2” validation: listing the files with select box, validation button, delete button, reports table, if file errors occur, next Section cannot display.
3. Section 3 “Import\_step3” save to database: save button, list saved files, delete button, using the same table in “Import\_step2” for error report
4. Section 4 “Import\_step4” to refresh page, option to delete files.

Import JS, project\_root/static/js/importhandler.js used for:

* Control Section 1-4 displaying sequentially.
* Send user input information to class Importhandler\_VITEK via Ajax Call
* Process data received from server via files selecting, conditions checking with looping and filtering, parsing text to show result.

In short, the main functions are sending data and parsing result to generate report performing, setting file status via the following steps:

* 1. Listing files before uploading
  2. Submit form to uploading via normal httpResponse
  3. Click button trigger Ajax call with JSONresponse for:
     + Send input data
     + Receive data from server
     + Looping and filtering data to get report for generating report table, setting status to files based on validation result (showing different colour: green meaning no errors, red meaning before validating or errors occurring)

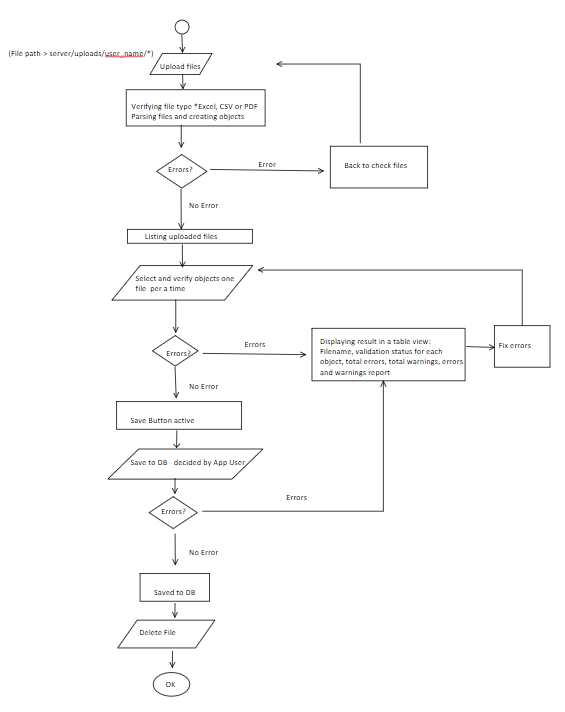
Code:



### 8 Vitek-import

Vitek Data Import Process is showed in the following diagram:

* 1. Vitek Data Import Process Diagram



* 1. Code Structure

1) itself special model check methods in ddrug/models.py

Example:



2) file parse and generate model objects functions in ddrug/vitek.py

3) class-based view in ddrug/views.py.

The view name is class Importhandler\_VITEK which is inherited from class Importhandler in apputil/utils\_dataimport.py. Then class Importhandler\_VITEK is to control the steps for data import descripted in “Vitek Data Import Process Diagram”.

In short, there are 4 functions to perform the steps: 1) **validating-upload** with parsing files and generating error messages, 2) **validating-objects** created by parsing files from selected files and generating reports, 3) **validating-save-objects** with only generating reports if errors, 4) **deleting** selected files from uploading folder on server.

Code as the following:



### 9 Modelfilter, Filterbaseview

#### Single datatable filter

To filter through a data table, Django-filter are implemented in the project. To construct a card/list filter view firstly need to build a model filter. Then implement it as a class variable in somemodels-Filterview. This somemodels-Filterview is inherited from a Filterbaseview class.

The codes are:

For Filterbaseview:

#### Multiple datatable filter

To filter through different data tables.

### 10 Data visualization

## Append 1 project directory tree view:

### 

## Append 2 JavaScript Functions

getCookie

utility function to get “csrftoken” put on xhr headers

**input parameter**

**usage**

editablerow

utility function for class “editablerow”

**input parameter**

**usage**

import script to html, using class name ‘editablerow’ for tr html element

editableTable

utility function for class “editableTable”

**input parameter**

**usage**

import script to html, using class name ‘editableTable’ for td html element

editableTable

utility function for class “editableTable”

**input parameter**

**usage**

import script to html, using class name ‘editableTable’ for td html element. prevented from keypress event

editableTablechoices

utility function for class “editableTablechoices”

**input parameter**

**usage**

import script to html, using class name ‘editableTable’ for td html element. Prevented from keypress event, which may result in miss-submitting form

importhandler

utility function to control file uploading process.

**input parameter**

**usage**

import script to the html.

send\_organism / send\_organism\_id

utility function to search and select organism name or organism\_id in the taxonomy and organism table respectively and assign the selected value to create and update Django form input

**input parameter**

scheduled\_function – functionally delay the sending data to server for a certain ms, initial value is False and will be assigned with a setTimeout function with 500ms inside the keyup event function.

searchInput – searching input

resultsBox – displaying selectable searching results

**usage**

import script to search\_organism.html / search\_organism\_id.html, using in organism create and update view, batch and culture create view.

sendToServer

utility function to post data to server

**input parameter**

data – data sent to server side in JSON stringify form, eg. data={key: value\_string, …}

url – post url address

**usage**

sorting\_table

utility function to sorting data entries through data table for list/card view.

**input parameter**

localStorage.setItem( ‘dorder’, ‘-‘) – desc ordering

localStorage.removeItem(‘dorder’) – asc ordering

order\_name – ordering field, input field in Django filter form

**usage**

import script to the html, add ‘order\_field’ class to th html element

vitek\_datamap

utility function to create pivot table for vitekcard.

**input parameter**

**usage**