NAME: VINAY KRISHNAMURTHY

MATRICULATION NUMBER: 809699

SUPERVISER: Prof. Dr. ANDREAS FISCHER

Web Scraping Call for Papers and Proposals with Scrapy and Web Development with Django

Table of contents:

Abstract

Here we use Scrapy and Django Frameworks. Scrapy is used for crawling webpages and scrape data from the webpages and store it in the database, the webpages can be accessed by the xpath and the parse method can be used to get scrap the data and these data are stored in item containers, these items are then yielded to the pipelines where it is then saved to the database. Whenever there is a request from the server django will analyze this request and then the required actions are done in the views and data will be fetched from the database with models and the data or response got is then given back from the views to the template.

A JSON API is created to search the fields and get the data in JSON format and Spider will run when the server restarts.

Introduction:

Here we are going to discuss about two frameworks Scrapy and Django.

Scrapy is a framework used to extract data from websites.

We can install it in command line by pip install scrapy and version used here is Scrapy==2.1.0

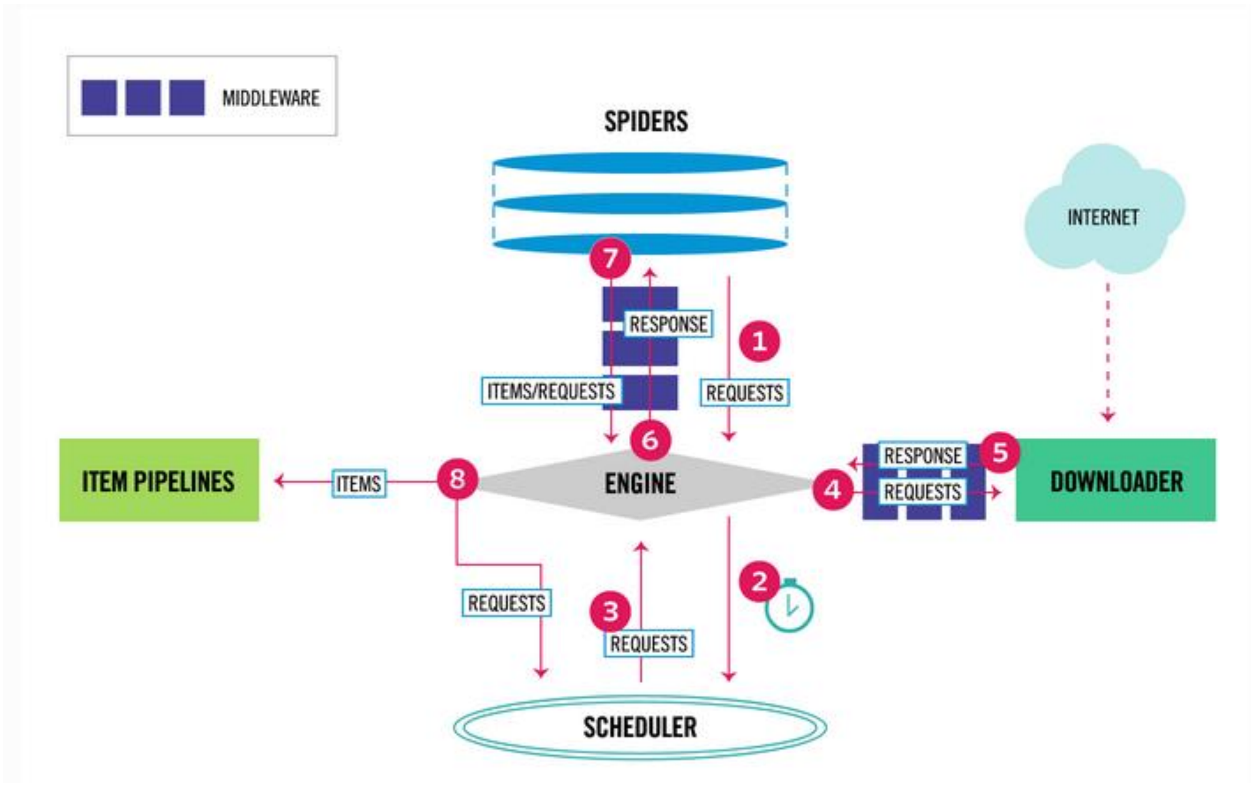


Fig1: Scrapy Engine.

Working of spider engine

1. The Engine gets the Requests from Spider to crawl.
2. Requests are schedules by the Engine and it will ask for next Requests to crawl.
3. The next Requests will be returned by the Scheduler to the Engine.

4. The Engine will then send Requests to the Downloader, passing through the Downloader Middlewares.
5. Then page will finish downloading and the Downloader will generate a Response (with that page) and will send it to the Engine, passing through the Downloader Middlewares.
6. The Engine will receive the Response from Downloader and will send it to the Spider for processing, by passing through the Spider Middleware.
7. Spider will then process the Response from the Engine and it will return the scraped items and new Requests (to follow) to the Engine, passing through the Spider Middleware (see process_spider_output()).
8. The Engine will then send processed items to Item Pipelines, then send processed Requests to the Scheduler and asks for possible next Requests to crawl.
9. The process will then repeat again from step 1 till there is no more requests from the Scheduler.

Scrapy Engine: The data flow between all the components are controlled by the engine and also triggering of events when there is any actions.

Scheduler: As the name indicates it will receive the requests from the engine and it will queue them to feed it to the engine when requests them.

Downloader: It will mainly fetch the web pages and then feed it to the engine which is then given to the spiders.

Spiders are nothing but the classes which users write in order to parse the responses and get the items from them or other requests.

Item Pipeline: Once the items are extracted from the spider then Item Pipeline is responsible for the processing of this extracted items. Normally the operations here include the cleansing, validation and persistence (like storing the item in a database).

Downloader middlewares: Middlewares are nothing but some backside functions which will be going on in the background when like when a class is called all the other functions what it has to do will be done by the software this is the Middlewares**. So** Downloader middlewares are present between the eingine and the downloader and this function is to process the requests when they pass from engine to Downloader and vice versa.

Some of the functionality of the Downloader middleware is:

- Processing of request before it is sent to the Downloader
- change received response before passing it to a spider;
- send a new Request instead of passing received response to a spider;
- pass response to a spider without fetching a web page;
- Drop some requests.

Spider middlewares These are present in between the Engine and the Spiders and will be able to process the spider responses(input) and requests(output)

Spider middlewares are used for:

- post-process output of spider callbacks - change/add/remove requests or items;
- post-process start_requests;
- handle spider exceptions;
- call errback instead of callback for some of the requests based on response content.

The normal function of the scrapy will have 3 main files.

1. File which contains the main spider class here in the program we can see that this spiderbmbf class is importing from scrapy.Spider.

   This class will mainly contain the name of the spider and the start_urls or the domain on which it has to perform the scrapy tasks.

   The spider class will contain function called parse which contains the paths to the required data to which it has to go and collect the data.

Finally after collecting the data it has to be given or yielded to the pipelines to send it to the database.

2. pipelines.py will contain the class ScrapyProjectPipeline which is mainly used to get the yielded data from the Spider and then here we can save the file to the database.
3. Items.py here a Spider after getting the data should store the data in some thing this is the container items which is used to store data after the scraping work is done. Here the class inherets from scrapy.Item and then different xpaths will have different data so each of them have to be given a particular field (scrapy.Field()) so that the data can be stored after scraping.

Django Web Framework

It is a framework which uses model-template-views (MTV) architectural pattern.

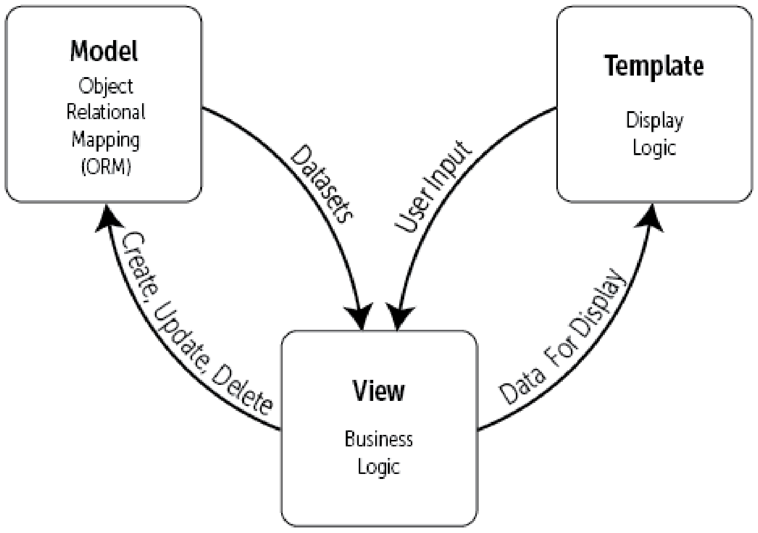


Fig2: Model Template View

Django uses http request and response objects to communicate between the client and the server. The basic operation of django is it view retrieves data from the database via the model, formats it, bundles it up in an HTTP response object and sends it to the client (templates to browser).

So in django we have to mainly create the models to interact with the database. So for interacting with the data from the database we use ORM.

The ORM mediates between data models (defined as Python classes) and a relational database ("**M**odel") i.e, it is a code library that automates (instead of writing SQL) the transfer of data stored in **relational database tables into objects(Model)** that are more commonly used in application code. So without writing the SQL codes we can interact with the data of the database from the models in django by ORM.

So with the ORM we can do the CRUD operations on the data of database by taking requests from the user and View will help us to do this.

A view function is the main part of django, it is a Python function that takes a Web request and returns a Web response, in more detail after getting the request if there are any requirements to operate on the database it will do so by creating a query as we can use models and operate on the database data.
When a query is done the following queryset is taken queryset is nothing but objects in the database.
So in order to get the queryset we have to use the manager of the model and in django the manager of the model has a default name as objects, so in order to use the all the objects we have to use the **Wiki_model.objects.all(),** here we can see that the objects is the manager of model, we are taking all() the objects of the Wiki_model, this will be returned as a queryset which contains objects of Wiki_model(table data in Wiki_model). The python code **objects** (Wiki_model) will be **related** and **mapped** to the database table containing data.
But if we want a particular objects from the queryset then we have to use filters so that we can get the particular objects from the queryset.

If a request is got view function will analyze the request and will give a response or will return the contents of what we have worked(query) on in the function to the Template. The response can be HTML contents of Web page, or redirect, or 404 error, or an XML document, or an image. The view contains arbitrary logic to return that response. The views function can be written anywhere but should be in the Python path. Or we write all the request and response functions in one file called **views.py**

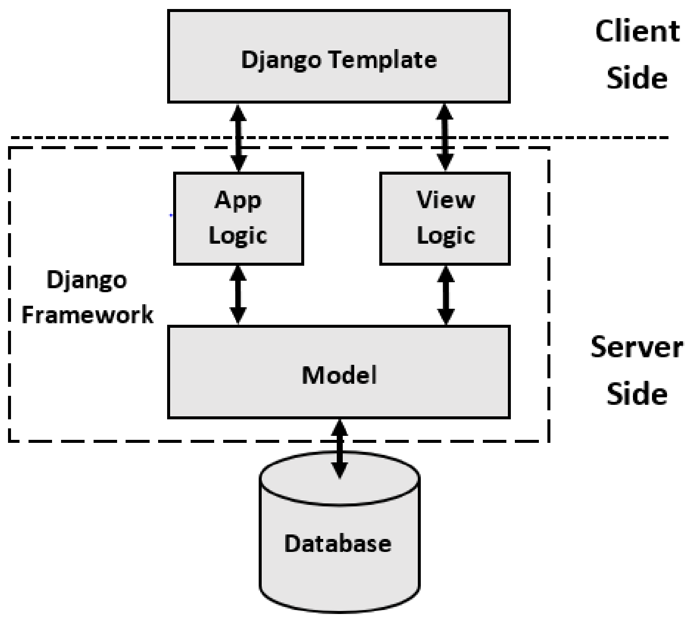


Fig3: Django Working.

As explained in the Fig3

- We can see that client will make a request from browser(will be template view), the required request is processed in the logic.
- If it is a request and wants to return a response then it is in the view function (View logic) but if it is anything related to the view logic(return response) then it is an app logic.
- The View logic will use the models and its manager to query database, this is nothing but using ORM to get the objects from the relation tables i.e, python code objects in model (id, name) are mapped with the related tables in database and when accessed can be taken from database as a queryset objects.

- The queryset objects if wanted are filtered and returned to the template from the view functions.
- The request and response from the template are routed to the views functions via a url.py file which contains the path

This is the main working process of the Django with MVT architecture.

Packages to be installed

Here we have used

1> Scrapy(Scrapy==2.1.0)
2> Django version(Django==3.0.8)
   2.1> djangorestframework==3.11.1
   2.2> django-filter==2.3.0
   2.3> django-bootstrap-form==3.4

Code what are code architecture

The main working of the code is explained here first we have to install all the packages.

1> Checking what are the data to be stored from the required webpage and which data type are the data. Then based on the required datatypes we can create the models and each website can have a single model and all the data with different datatypes are named here. In the models.py we can see that the each website bmbf and wikicfp has different models and their names are Bmbf_model and Wiki_model respectively. This different models are inheriting from the models of the django and we can create this tables in the database as explained previously with ORM. The models and datafields presented in django models.py file has to be migrated to the database.

Commands are 1. python manage.py makemigrations
2. python manage.py migrate

After the migrations the tables are ready to be filled with data so we can move to scrapy and then we will go through all the scrapy process and then we will move back to django in order to display data.

2> So now we have to check scrapy to get the data from the websites and all the basic functioning of the middlewares are explained and what are the basic functionality of spider. So going through the code, first we have to create the items called containers to store the data this can be done in the items.py we can create a container for each website here for bmbf and wikicfp the class name is EventItemBmbf and EventItemWiki.

3> Now looking at the actual spider we have created two spiders with name SpiderBmbf and SpiderWiki for bmbf and wikicfp and the url name (domain name can also be added)has to be given in order to see which page has to be crawled here it is start_urls. The xpath of the different data to be scraped is taken it is written here because we can use this paths more flexible in the methods and in other files as well. Then the main function parse is where we do all the scraping.

3.1> **Parse** is a method which will allow to check the behavior of different parts of the spider. It will not allow debugging code inside a method. The method is called for all the response downloaded for each of the request made. The response parameter is an instance of TextResponse that holds the page content and has further helpful methods to handle it.

The parse() method parses the response and extracts the scraped data as dicts and also finding new URLs to follow and creating new requests (Request) from them. It will also process the response and return the scraped data (as Item objects) and more URLs to follow (as Request objects). The items are handed over to the item pipelines and item exporters. The requests will be put into the Scheduler which pipes the requests to the Downloader for making a request and returning a response. Then, the engine receives the response and gives it to the spider for processing (to the callback method).

In scraping we have to extract structured data from unstructured sources, i.e, web pages. Spiders will normally return the extracted data as *items*, Python objects that define key-value pairs.

The objects used to gather the scraped data are created from the Item class of the scrapy and scrapy supports multiple types of items(dictionaries, Item objects, dataclass objects, and attrs objects.). Items are used to store the data which is crawled. scrapy.Items are basically dictionaries. In order to declare items, a class is created and then add scrapy.Field in it:

```
import scrapy

class Product(scrapy.Item):
    url = scrapy.Field()
    title = scrapy.Field()
```

Item objects: Item provides a dict-like API

Item allows defining field names, so that:

- KeyError is raised when using undefined field names (means prevents typos going unnoticed)
- Item exporters will export all fields by default even if the first scraped object does not have values for all of them.

Item subclasses will be declared as simple class definition syntax and Field objects.

Field objects are used to specify metadata for each field. The main goal of Field objects is to provide a way to define all field metadata in one place. Field objects used to declare the item do not stay assigned as class attributes. Instead, they can be accessed through the Item.fields attribute. The Field class is just an alias to the built-in dict class and Field objects are plain-old Python dicts

Item loaders: Item Loaders will provide a mechanism for populating scraped items i.e, items provide the container of scraped data and the Item Loaders provide the mechanism for populating that container.

To use an Item Loader, first instantiate it. Do it with an item object or without one, during that case an item object is automatically created in the Item Loader with __init__ method using the item class specified in the ItemLoader.default_item_class attribute.

Selectors will collect values into the Item Loader then we can add more than one value to the same item field and the Item Loader will use a processing function to “join” those values later.The Collected data is internally stored as lists, allowing to add several values to the same field.

The Title field data is collected from xpath using add_xpath() method and this data is assigned to the Title field and so on. And it will be continued for date and Url fields and then at the end when all the data is collected ItemLoader.load_item()method is called which will return the item (name) populated with the data extracted and colledted from add_xpath calls.

Item Loader contains one input processor and one output processor for each (item) field. The input processor processes the extracted data as soon as it’s received (through the add_xpath()method) and the result of the input processor is collected and kept inside the ItemLoader. After collecting all data, the ItemLoader.load_item() method is called to populate and get the populated item object. That’s when the output processor is called with the data previously collected (and processed using the input processor). The result of the output processor is the final value that gets assigned to the item from the Title or name Field. For example Data from xpath1 is extracted, and passed through the *input processor* of the name field. The result of the input processor is collected and kept in the Item Loader (but not yet assigned to the item). Data from xpath2 is extracted, and passed through the same *input processor* used. The result of the input processor is appended to the data collected in before.

The data collected from all the xpath is passed through the *output processor* of the field The result value assigned to the field in the item.

Basically telling values returned by input processors are collected internally (in lists) and then passed to output processors to populate the fields.

Item Pipeline

After an item has been scraped by a spider, it is sent to the Item Pipeline which processes it through several components that are executed sequentially.

Each item pipeline component ("Item Pipeline") is a Python class that implements a simple method. It receives an item and performs action over it, it will also decide if the item should continue through the pipeline or should be dropped or no longer processed.

Some of the uses of item pipelines are cleansing HTML data, validating scraped data, checking for duplicates and removing them and finally storing scraped item in database.

Activating an Item Pipeline component

To activate an Item Pipeline component you must add its class to the ITEM_PIPELINES setting, like in the following example:

```
ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 300,
    'myproject.pipelines.JsonWriterPipeline': 800,
}
```

The integer values you assigned to classes in this setting will determine the order in which they will run: items will go through lower value to higher value classes. It's customary to define these numbers in the 0-1000 range.

Mainly each item pipeline component is a Python class which must implement the following method:

process_item(*self*, *item*, *spider*)

This method is called for every item pipeline component.

item is an item object, see Supporting All Item Types.

process_item() must either: return an item object, return a Deferred or raise a DropItem exception, the dropped items will no longer be processed by further pipeline components.

Parameters

- **item** (item object) – the scraped item
- **spider** (Spider object) – the spider which scraped the item

I have used Pycharm as IDE and we have to install Django and Scrapy as in the requirement.txt

SCRAPY Framework code explanation

Working procedure of the scrapy.

1. In the code we have a spider class which is inheriting from the scrapy spider class.
2. The spider class has a name which can be changed to anything here we have given SpiderBmbf and SpiderWiki. Then we can give the domain name and the url which has to be scraped, we can also give the xpath and use it later(TitleXpath, DateXpath and UrlXpath).
3. Then we have a parse method where the spider will be give the xpath and it will use the xpath to crawl the webpage and items data from this paths are loaded to the item loaders. The ItemLoader will have many fields here we have used item=EventItemBmbf() and EventItemWiki(). The response is the url of the webpage the response from the webpage.
4. The loader will contain the response and the item container then the add_xpath will be the command to put all the data from the required TitleXpath, DataXpath and UrlXpath, these data from different add_xpath will be loaded to the item loader then in the loader.load_item() all the data will be loaded to the item.

5. Testing

5.1> The data present in the xpath is checked by using response.xpath(self.TitleXpath).get() Here self.TitleXpath is the xpath of the Title and we are checking whether the response from the webpage have this part and get is used to check that data is present or not, if nothing is present in the Title or date or Url then a error is raised. There will always be a value in this fields from the Bmbf and the Wiki fields like(TitleXpath, UrlXpath and

DateXpath from Bmbf webpage and in wiki extra fields are DeadlineXpath, PlaceXpath.)

5.2> Then we test if the items are really loaded to the item so if TitleBMBF or DateBMBF or UrlBMBF is present then it is alright if the data is not loaded properly then there is a problem with the xpath or the data might have other problem.

6. Here mydict is temperory dictionary to store all the item data, so that we can manipulate the data how we need to send the data to the database i.e, different datetime format or to change to other format or discussed further. In the for loop we will loop till how much data in present in the UrlBMBF and I have not used other TitleBMBF or DeadlineBMBF because we will always have URLBMBF and it will never change to any other format but other fields can change or they might not be present.

The mydict[**"title"**].append(item[**"TitleBMBF"**][k]) will directly append the TitleBMBF to the mydict["title"].

Here Mydict["date"] is the main because we have to convert the date to the datetime format so we can put filter functions in the django to display it properly. So in order to change this date in the string format from the webpage to the datetime format we have to use the strptime which will mainly convert the string to the datetime format and this uses the library datetime.

mydict[**"date"**].append(datetime.strptime(item[**"DateBMBF"**][k].replace(**" "**,**""**).split(**"-"**)[0], **'%d.%m.%Y'**))

So from the above line we can see that to mydict["date"] we are appending the data from the rest of the command, so in the append command I have used from datetime library we change the format from the string to datetime format. To convert to date time we will remove all the spaces in the string by replace command we replace the spaces, then if there are two dates present[0] will be the first date and [1] is the second date. The second date after the hyphen (Firstdate-Seconddate) that seconddate is the [1]. For taking the second date I have used the if statement [len(item[**"DateBMBF"**][k].split(**"-"**))>1] If the hyphen is present then we have to take the second one[1] as the deadline so the if statement, otherwise it will store that there is no deadline and first date[0] is the normal date.

The format in DateBMBF is of '%d.%m.%Y' day.month.year we have to give this dataformat from the DateBMBF to the strptime to convert this format string to the datetime format. the date format from the string is different then we have to change it to the different format like wikicfp date is different so we have used (%b %d, %Y) so if the date changes then we have to change this particular line to the changed format. So final output is in datetime format.

mydict["url"].append('https://www.bmbf.de/' + item["UrlBMBF"][k])

This url line we are adding the domain name to the whatever data we have collected from the item["UrlBMBF"] and storing it to mydict.

There is a bit change in the fields of the Wikicfp page. Here in the for loop TitleWiki is better because no much manipulation has to be done for this, it is same as UrlBMBF as it is safer. The for loop will go up to the TitleWiki and int is used to make it integer and length will show how much items TitleWiki has it should be 20 items. The title of wiki is direct mydict["title"].append(item["TitleWiki"][k])

Then in the date we have to change as the format is different from the BMBF Date. So here if date field is "TBD" or "N/A" then nothing is loaded to the mydict["date"], else if other format is present then

mydict["date"].append(datetime.strptime(item['DateWiki'][k].split("-")[0][0:12].rstrip(),'%b %d, %Y'))

The format is '%b %d, %Y' in wikicfp. Rstrip will remove all the last spaces in DateWiki. Then the DateWiki is split if two dates are there at "–" with split command. [0] gives the first date with only 13 spaces in the data are taken and more can be taken but we cannot give less number than 12 as the total string is that much. Then again we give it to the strptime to convert it to the datetime format.

Then Url is the same we can directly append it with adding the domain name 'http://www.wikicfp.com'

Then again in the DeadlineWiki same thing as the DateWiki is checked and if TBD and N/A is present then it is not important to convert it to datetime format so none is stored when we get this. Then for DeadlineWiki same thing as the DateWiki is done.

7. Then at the end all the values which are appended to the mydict are yielded to the pipeline and there here we have directly called the Pipeline class ScrapyProjectPipeline()

with process method myPipeline.process_item(mydict,SpiderBmbf) and have passed mydict and the name so we can identify from which spider it is from.
In the Pipeline file when Spider calls it The method process_item will get the mydict which is the item and the spider name which is spider here. Then in the for loop we have just taken the the number of data in the (item[**"title"**])
The spider name is checked if data is from SpiderWiki or SpiderBmbf.

We have to import models from django models then Here we have used MyModel to take the Wiki_model and the Bmfb_model. From this we can directly use the tables of the class and store the data from item into that
MyModel.Title = item[**'title'**][item_data]
Here MyModel maybe Wiki or Bmbf Model and the item or data from the spider mydict will be passed to item in class ScrapyProjectPipeline then all the items in the dictionary like [title, date, deadline, papertype, where, url for Wikicfp] and [title, date, deadline, url for BMBF] can be stored to the model and then MyModel.save() command will save it to the database.

8. **Scrapy Settings**: The interaction from scrapy can be done by changing the system path to django_project settings
(sys.path.append(**'callspiderproj/scrapy_django/django_project'**))
sys.path.append(os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(__file__))), **".."**))

1. abspath returns absolute path of a path
2. join join to path strings
3. dirname returns the directory of a file
4. __file__ refers to the script's file name
5. pardir returns the representation of a parent directory in the OS (usually ..) stands for ".." which means one directory above the current one

The os environment should be directing to the django setting module which is in the django_project.settings which is present in the settings of the scrapy

The application = get_wsgi_application()

As we are using wsgi there was an error that I was not using the wsgi application so had to include this.

DJANGO Framework code explanation.

Command to check django version django-admin –version

Procedure to install django

1. Install virtualenvwrapper-win
   ⇨ mkvirtualenv test   for testing if the environment is created.
2. pip install django
3. First project creation django-admin startproject django_project

   In Django we have to create the app which we are to work on, here I have named it as questions.

   The command used for it is python manage.py startapp questions
   In the INSTALLED APPS in django settings we include our app so we type in questions

4. We have to tell the path of the app in order to access it so we have to add the path as path('',include('**questions.urls**')), Now the project knows that we can access the app called questions

5. Then to display whatever data we want on the webpage we have to define all the functions in views.py which is in the app questions. So in order to access this views we have to give the path of the functions which are in views so in the app(questions) url file we have to add the path which wever function or class we are using for example which I have used is path('**dashboard/**',views.get_SnippetListView, name="**dashboard**"), Here the path is the keyword to say it is the path and dashboard/ is the name in which we can access as the webpage and views.get_SnippetListView, is the function name which is present in views and the name to call it in the webpage is dashboard.

6. Then the function in the views.py have to be displayed so in order to do it we have to run the server by going to the path of the project which we have the manage.py file here outside of the folder of django_project folder, then we have to type the command python manage.py runserver. If the server is running successfully then we can see the function data which had to be displayed from the function in views.py or default that the server is running successfully.

   So Basically what happened is we have the django project and we created app and told the project that we have an app which is directed by giving the path in project url, then it will know the app is present and then in the app url we gave the path to the function to views.py so that it can display it to the webpage so if a request in got from webpage the function will receive it in views.py and it will return a response.

7. So the responses to be returned should have a particular style and it should display it properly so in order to display all the data with style we have to create a html template where we can take all the data from the views and display it in the html and css as the webpage. So these files are called templates in django so we can make this directory and can keep all the html files here so all the responses can be displayed and requests can be taken.So inorder to tell the app that you have to go to the templates to display it we have to include this templates directory in the TEMPLAES of django settings (**'DIRS'**: [os.path.join(BASE_DIR,**'templates'**)],). So till here is the basic function of how django works.

8. There are **2 types of http REQUEST METHODS** GET(request to fetch data from server) and POST(request to submit data). In the GET method(Fetching data from server) we need something from the server i.e, data like images or links. For submitting the data to the server we have to use POST. **FORMS are used to submit what ever the Request methods data**. In the GET method the main thing is even if we want to fetch the data we have to give that data in the search bar and then only we can fetch from the server but if POST is used then all the data is not shown on the search bar and it is directly sent to the server. I have used a GET method for the searching the dates so in order to show some

fields on the search bar and to fetch the data from the server I have used GET method instead of POST but if I wanted to just send all the data without showing it on the search bar or if many fields were there then I would have used POST. For example in the dashboard.html we have a form which is of the get method then when we click on the submit button all the data which was given in that box is stored in it with the <**input value="Lesserdates" type="radio" name="extremes"**>. When submit button is pressed the data which I gave will be fetched from the server (and in database) and can be accessed by its name extrems in the views.py I have used it as extremes = request.GET.get(**'extremes'**) so I can fetch this extrems data by the given command (from the server) and can work on it or use it for different purpose. If I wanted to use POST I would have done extremes = request.POST.get(**'extremes'**) and in order to get the data I would have used .get(‘extremes’).

CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they’re currently authenticated it will occur when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. So Django has a middleware to protect from this attack and so we can use this in form tag [{% csrf_token%}]

The normal working of User to Django and back will be

[USER(Request methodsGET POST DELETE)]<=>[DJANGO(API, middlewares)]<=>[URL(requests directed to views)]<=>[VIEW(receives request and returns response)]
1<=> [Model]<=>[DATABASE]
2<=>[TEMPLATES]

9. The database used here is SQLite and it is the default for Django.

10. In the fig1 explanation of ORM is done here how a model is created and how it will work is explained. So in the models.py a class has to be created so that it will create all the tables and columns. A class called Wiki_model is created and it will inheret from django models and this is created as a Wiki_model table. Then all the fields that it should have has to be created to create the number of columns. Title = models.CharField(max_length=1024) Here Title is the column name and type of it is characters this is given by CharField and we can restrict the max length. The other fileds can also be created with integer filed and etc but here another field used is the date field Dates = models.DateTimeField(null=**'True'**) So here the table name is Dates and its type is DateTimeField so it can store only datetime format and due to this it is possible to use the django filters to filter out the deadline and other dates accordingly and the null field is given to be true that is the column can have nothing in it then it will not through any error if it is null. So this is how a model has to be and if required a class Meta can also be added and the table name can be changed to the given name or we can use the normal class name.

    Now after knowing how the models structure should we have to migrate this models to the database in order to create those table and columns in the database. The commands used are as follows

    ```
    $ python manage.py makemigrations
    ```

    to make the migrations to the database and we can name the migrations differently as well from this we can get the migration file in migrations directory which is created itself in django. The migration file is created but it is not yet migrated to database.

    ```
    $ python manage.py migrate
    ```

    To migrate the migrations file to the database we use the above command and we can see the migrated table in the database.

11. Now to access the data from the database we can use the model name and its objects like Wiki_model.objects.all() will access all the data from the database and we can then

manipulate the data how we need it and send it to the template which will in turn display it. Many commands similar to this with the objects have been used here like displaying the last 20 rows(Wiki_model.objects.order_by('**-id**')[:20]) and with distinct keyword only those field with unique data can be accessed (Wiki_model.objects.values_list('**Url**').distinct()) and etc.

12. **Templates:** These are html files which will help to interact with the user, it will help to get the data from the user or display the data on webpage to the user from the database. In django we can display static data and also dynamic data by using {{}} this is called the **Jinja format**. With this format we can use different types of loops as well like the if and for loop like {% if condition %}{{Data}} {{end if}} here if the condition is right then it will enter into the loop.
    {{for item in items}} {{item}} {{endfor}} this will print all the item data in items.
    These are some of the functions of the templates we can get to know better when we go through the code.

13. Django will provide a **User Authentication System**. This can be accessed by the user, the command python manage.py createsuperuser will create a user and we can have all the admin functionality of django in it. After entering the command we have to give the Username and password, then we can access the django authentication system. We can also create other users and can restrict the access of the users, only admin will have all the access to the database but other users are restricted from all the privileges. Here in the for the user authentication I have used three urls for login register and logout.
    The form registration is done using already existing UserCreationForm here only the username, password1 and password2 fields are present and if we want to add a field like email we have to create new class for registration and we can find it in the django documentation(https://overiq.com/django-1-10/django-creating-users-using-usercreationform/)

1> Here in the registration form we will get the 3 fields.

2> In register.html a normal form is created with button.

3> Then the url path is given in the url of app
path(**'register/'**,views.registerView,name=**"register_url"**),

4> This will take us to the views and here form was empty so I have added a dummy string and the POST data is taken and is checked if all the values given are correct or some alteration should be done to password.

5> If all the data given is not conflicting then the data is saved and it will take us to the login page where we can login with the just created username and password.

LOGIN: For the login we just need the username and the password. The login page has a form which uses the POST method to send data to server. The django library **from** django.contrib.auth.views **import** LoginView, LoginVIew class will help us to give the login functionality we have can use it directly in the path. The data from the form is passed to the path which contains the LoginView class and django will take care of the login function. Checking of the users can be done in the Django Admin panel. After the login we have to give the to which page it has to go after the login so we have to give a command in the settings.py file as LOGIN_REDIRECT_URL = **'/dashboard' after login it will go to dashboard we can change it which page it has to enter after login.**
@cache_control(no_cache=**True**, must_revalidate=**True**, no_store=**True**)
@login_required(login_url=**'login_url'**)

1. The decorator at login_required is used so that an unauthorized person cant access the webpage and this can be used in the views.py which will be bound to the function in which template it will be used.
2. Once we logout in a certain page we have to stop the user from entering to the same page using the back button, so cache_control decorator is used to avoid the reentering to the previous page when back button is pressed as it will clear all the cache memory of the browser so the back button can’t go to the previous page.

Logout: The logout view path is used to logout from the webpage when the user is authenticated.

path('logout/',LogoutView.as_view(next_page='login_url'),name="logout"),

The logout view is also the same as Login view it is imported from django library. So in the path directly we can use the LogoutView class so after the logout we can provide to which page it has to go as the next page, here it will go to the login_url (as next page). This logout is just a class and it does not require any template.

So from the above procedure we have got to know how to create django project, app, models i.e, tables in the database and how to access them and also user authentication. Now below a brief explanation of how the code works.

1. Run the server by python manage.py runserver
2. Models are created as the procedure and migrations are done to the database.
3. Run the spider with python manage.py run_spider, the data will be stored to the database.
4. The functions in views are the main working procedure of the whole system.

5. The get_DashboardView function:

   In this function all the objects from wiki and Bmbf are taken and and the OrderFilter class is called to apply filter functionality on them.

   The OrderFilter and OrderFilter1 are classes where we should specify the model which we will be using and the fields which has to be filtered like for Wiki_model we filter Title, Url and Where fields, for Bmbf_model we filter fields like Title and Url.

   So in the contextwiki and contextBmbf we have to we have to see if the request has come for the fields and all those model objects are given to this class so by getting this we can get the search blanks of Title Url on the webpage.

   Bmbf_model.objects.values(**'Title'**, **'Url'**,**'Deadline'**).distinct()[::-1]

   stuff_items and stuff_items1 are variables into which we have to give the above command so we can filter out all the distinct values and display them from the reverse order.

   Then this stuff_items and stuff_items1 filtered result is sent to the Paginator

Paginator is a builtin django class which will help to make all the data to be ordered in a particular page format in a page and Paginator(data,number of items to be displayed) so here I have displayed 10 values in one page.
Page_num will get the value of the page from the request method, this page_num got from the request is given to the Paginator function with variable name paginator, so the ordered data of the database wiki and bmbf is ordered into a particular page and the link of the pages (nextpage previouspage, or page_num) requested is taken and if any page_num request comes it will navigate to that page_num or other next or previous pages. If the user gives a page which is not present then we should always show the 1 page otherwise error will occur so in th except block paginator.page(1) is present.

In the template dashboard.html we can see that a form is created for the different page_num previous page or nextpage.

1. *<!—part2*-->Here {% if contextpage.has_previous %} is a function for the page to see if a previous page is present then it will take us to that previous page and foe the next page is also the same.
2. {{contextpage.number}} will provide the pagenumber and {{ contextpage.paginator.num_pages }}
   Will give the total number of pages in contextpage.
3. for loop is used to display the number of pages and if the ?page is used to search the pages in the search bar and it will take us to that page and span tag is used to display to check if we are in that page and display that page name else is there to display all the page numbers except for the one page which we are in currently. This is about the display of number of pages previous page and next page and to navigate to it.
4. In the dashboard.html another form is there and the action will be done in the Call_for_Papers template. Here to display the from and to date of the radio button we have used From: and To: <part1>, the type of the button is radio and the name of it is extremes and we can take this value from the user webpage to the views function by the name="extremes" and its value can be verified if it is value="Lesserdates" or value="Greaterdates" then that value from the calendar can be accessed from the request.get in the views function.

5. <!—part3--> Is used to display all the data from the Bmbf model inside the table.

In the function **Search_Call_for_Papers** we will get values from Call_for_Papers template in that form we had given the fromdate, todate and radiobuttons foe displaying all the values less than the from data and all the values greater than the from date and at last radio button for deleting all the repeated values in the database, these data are used here in order to do the functionality like delete repeated data from database and display all the dates before or after the particular date.

1. In the above function if the todelete radio button is pressed then I have taken all the items and only nonrepeated items(wiki_all_items) from the database by the distinct functionality(wiki_itmes).
   Two for loops are used just to loop through all the items first for loop is we take only the nonrepeated item and then loop it through the all the wiki_items then we can count if there the non repeated value is present in the database more than once by count variable, if the count is more than 1 then we have repeated values so we will delete those items with the help of the id i.e, the repeated value id is take and it is deleted. This is how the for loops work.

2. To display only the dates before or after the particular date.
   Extremes is the keyword used which values we will get from the template Call_for_Papers its values are Lesserdates(to get the dates only lesser than the given fromdate) or Greaterdates(to get dates greater than the particular date)
   So in the code we have taken if extremes value is lesserdates and I have checked if fromdate is greater than the deadline of Wiki_model then store only those values to the dictionary lesser_dates dictionary.
   Else if the Extremes is Greaterdates check those dates if from date is less than that of the Wiki_items deadline if yes store it in greter_dates dictionary. These dates dictionary is sent to the templates and finally if the fromdate, todelete radiobuttons, Title or Url are not given or pressed then we can only get the full values of the corresponding model.

3. Template Call_for_Papers_Search_Result.html

   This template is mainly used to display all the Title, Place and Url of the Wiki_model which is received from the Call_for_Paper template. Other Functionality includes the above mentioned in the functions like all lesser or greater dates and delete all repeated values.

1. Part1 only displays all the values of the wiki_model without the search function.
2. Part2 will display only the from and to dates this is got from the from_to_date which gets the value only of the fromdate and the todate with the raw SQL command. Here The SQL commands also has the functionality to select only dates lesser than the given date or greater once and then we can order them by the ascending order of the dates.

   Wiki_model.objects.raw('select * from questions_wiki_model where Deadline between "' + fromdate + '" and "' + todate + '" ') will dates between those choosen dates.
   Wiki_model.objects.raw('select * from questions_wiki_model where (Deadline <= "' + fromdate + '") ORDER BY Deadline') this will give only dates less than the fromdate and it will be ordered in the ascending manner of deadline.

   Then in Part 3 searching of the dates is done with only python instead of SQL.
3. In Part3 we have we have used nested for loop to get the values from the dictionaries i.e mydict Title and it will enter to next foor loop and it will check If the both the values are of the same count if they are of the same count next it will go to the forloop of the Url again check if counter is same and go into foor loop of Where and it will take that place, Hence if the values of Title, Place, Url and Deadline is same then all the values are displayed with Jinja format so for loops are used to get the data from the dictionary and it is the same for the greater dates as well.

The above explanation is the same for the **Search_Call_for_Proposals** function and to get the value of the Title, Url and Fromdate and Todate we have the search bar in the dashboard and everything for display of the same result is done in the Call_for_Proposals.html file.

JSON API

The django-filter library includes a DjangoFilterBackend class which supports highly customizable field filtering for REST framework. So we have to install it with pip install django-filter

We sould also add it to installed apps as 'django_filters',

And we have to add the below in the settings.py of django REST_FRAMEWORK = {

```
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend']
}
```

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

The model for the serializers should be created in other python file here it is serializers.py, then in this we can create classes for the required models in the database and we can choose on which fileds the operations like filter, ordering or search should be applied here I have used all fields.

We have 2 classes to display the JSON API bmbfjson and wikijson. These classes will inherit from the django generics class, the fields which we have to give a serializer_class as BmbfSerializer or WikiSerializer, this is nothing but the definition where we are giving the model instances and data of the models as queryset to convert it to JSON format.

filter_backends= (DjangoFilterBackend, OrderingFilter, SearchFilter)
in this we can include which functionality we need like search or ordering etc.

So for filtering the fields of serializer_class we have used all the fields like filter_fields = ('id','Title','Url','Dates','Deadline') this can also be extended to the search_fields and ordering_fields.

Filter_fields will filter the required data on particular fields where as Search field will search or operate on all the fields i.e, it will search all the field once we use it unlike filter field and ordering is also done for all the fields.

To filter the data we can use for example if we want to operate on id field we can type the following in the search bar of browser. http://127.0.0.1:8000/bmbfjson/?id=2 for bmbf model and http://127.0.0.1:8000/wikijson/?id=2 for wiki model.

This searching can be extended in the search bar or we can use the filter button on the right side of the page and we can enter which field is required and we can get the all the searched data it will automatically do it in the search bar of the browser.

Search field: The SearchFilter class supports simple single query parameter based searching, The search_fields attribute should be a list of names of text type fields on the model, such as CharField or TextField

In the filter button there are other functionality as well like ordering the contents by id or by ascending, descending or by Title etc.

As used we can use all three functionality in the filter button which are Filter fields, ordering filter and Search.

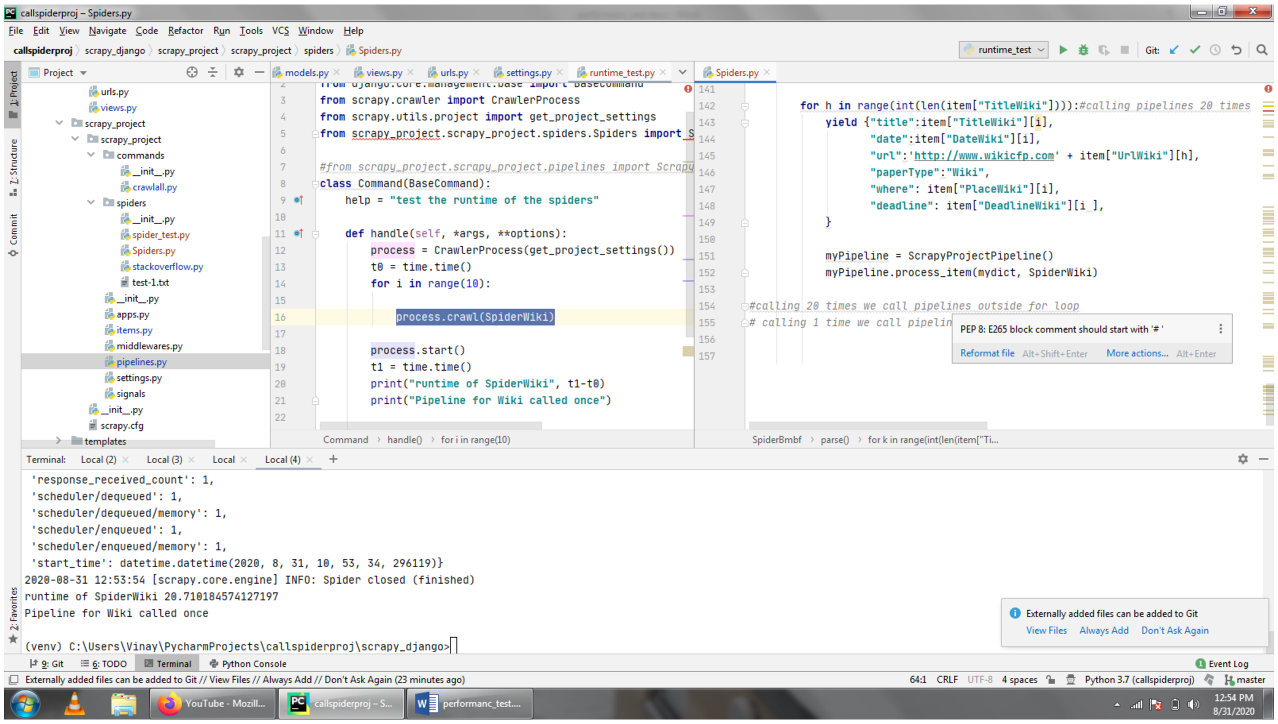
All the functionality can be found in guide of restframework.

Performance test:

In the app questions a directory commands we have runtime_test.py we can use this to run the required spider for 10 or more times to check the performance of the spider and below in the pictures I have shown the Spiders.py file and also testing file, the testing file will run the spider in a for loop for how many times we need.

If we want to call the pipeline once to send all the data check in scrapy_project directory then spider directory a file called spider_callonce.py is present this will call pipeline once as all the scraped data is first stored in mydict and then the yield command is there outside the for loop so everything is yielded once and the file Spideryield_manytimes will have the yield in the for loop so how many times the for loop will run that many times it will be yielded to the pipeline so it will call pipeline many times.

Run pipeline multiple times:


callspiderproj – Spiders.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
callspiderproj scrapy_django scrapy_project scrapy_project spiders Spiders.py
runtime_test
Git:
models.py views.py urls.py settings.py runtime_test.py Spiders.py
Project
urls.py
views.py
scrapy_project
scrapy_project
commands
__init__.py
crawlall.py
spiders
__init__.py
spider_test.py
Spiders.py
stackoverflow.py
test-1.txt
__init__.py
apps.py
items.py
middlewares.py
pipelines.py
settings.py
signals
__init__.py
scrapy.cfg
templates
Command › handle() › for i in range(10)
SpiderBmbf › parse() › for k in range(int(len(item["Ti...
PEP 8: E265 block comment should start with '# '
Reformat file Alt+Shift+Enter More actions... Alt+Enter
Terminal: Local (2) Local (3) Local Local (4)
Externally added files can be added to Git
View Files Always Add Don't Ask Again
9: Git 6: TODO Terminal Python Console Event Log
Externally added files can be added to Git // View Files // Always Add // Don't Ask Again (23 minutes ago)
64:1 CRLF UTF-8 4 spaces Python 3.7 (callspiderproj) master
YouTube - Mozill... callspiderproj – S... performanc_test....
12:54 PM 8/31/2020

```python
from django.core.management.base import BaseCommand
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings
from scrapy_project.scrapy_project.spiders.Spiders import S

#from scrapy_project.scrapy_project.pipelines import Scrapy
class Command(BaseCommand):
    help = "test the runtime of the spiders"

    def handle(self, *args, **options):
        process = CrawlerProcess(get_project_settings())
        t0 = time.time()
        for i in range(10):

            process.crawl(SpiderWiki)

        process.start()
        t1 = time.time()
        print("runtime of SpiderWiki", t1-t0)
        print("Pipeline for Wiki called once")
```

```python

        for h in range(int(len(item["TitleWiki"]))):#calling pipelines 20 times
            yield {"title":item["TitleWiki"][i],
                   "date":item["DateWiki"][i],
                   "url":'http://www.wikicfp.com' + item["UrlWiki"][h],
                   "paperType":"Wiki",
                   "where": item["PlaceWiki"][i],
                   "deadline": item["DeadlineWiki"][i ],
            }

            myPipeline = ScrapyProjectPipeline()
            myPipeline.process_item(mydict, SpiderWiki)

#calling 20 times we call pipelines outside for loop
# calling 1 time we call pipelin
```

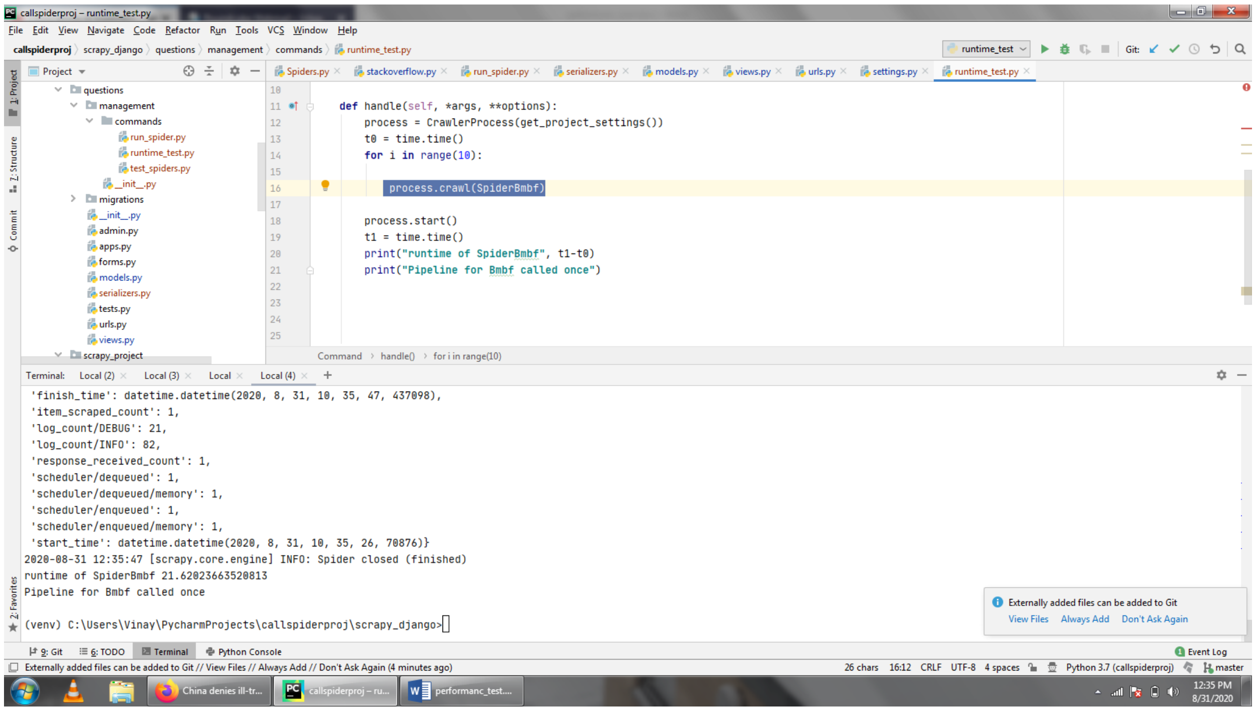
```
 'response_received_count': 1,
 'scheduler/dequeued': 1,
 'scheduler/dequeued/memory': 1,
 'scheduler/enqueued': 1,
 'scheduler/enqueued/memory': 1,
 'start_time': datetime.datetime(2020, 8, 31, 10, 53, 34, 296119)}
2020-08-31 12:53:54 [scrapy.core.engine] INFO: Spider closed (finished)
runtime of SpiderWiki 20.710184574127197
Pipeline for Wiki called once

(venv) C:\Users\Vinay\PycharmProjects\callspiderproj\scrapy_django>
```

Here all the data is sent to the pipeline one by one and is saved in the pipeline. Spider is called 10 times

Spiderbmbf will take around **20.7**-21.9s

SpiderWiki will take 20.2 – **20.7s**

Below pipeline is called once:

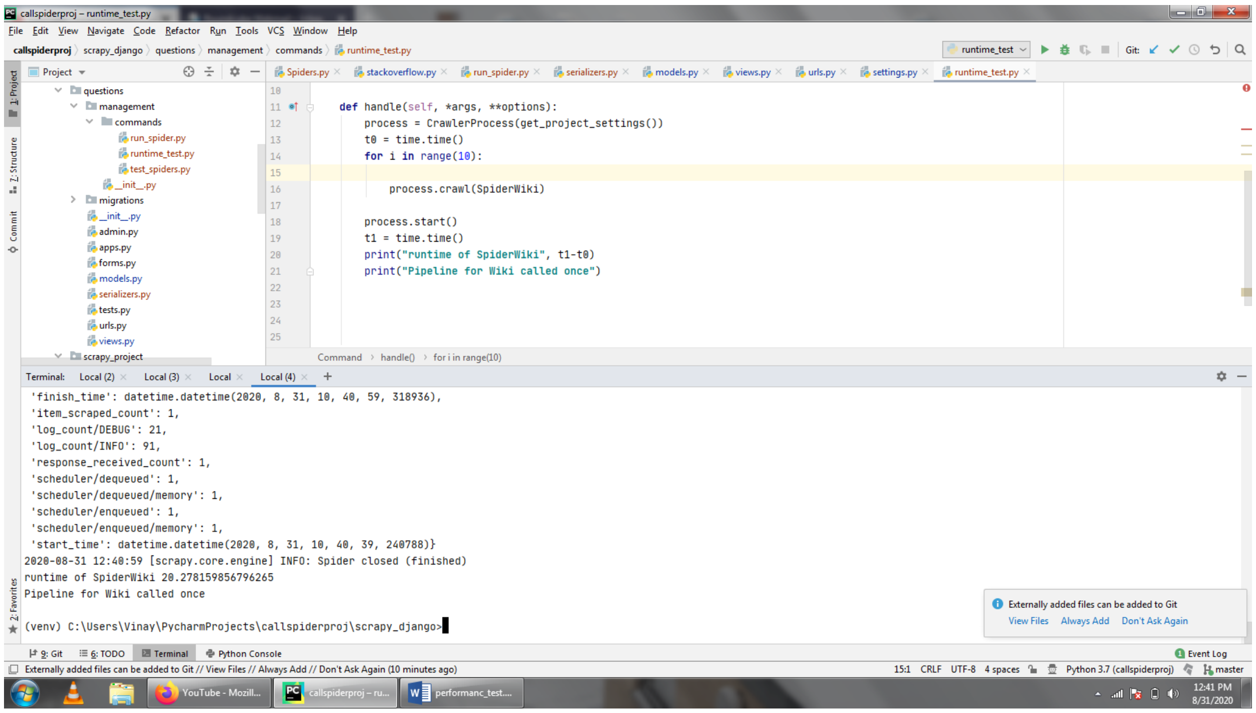


```
    def handle(self, *args, **options):
        process = CrawlerProcess(get_project_settings())
        t0 = time.time()
        for i in range(10):

            process.crawl(SpiderBmbf)

        process.start()
        t1 = time.time()
        print("runtime of SpiderBmbf", t1-t0)
        print("Pipeline for Bmbf called once")
```

```
 'finish_time': datetime.datetime(2020, 8, 31, 10, 35, 47, 437098),
 'item_scraped_count': 1,
 'log_count/DEBUG': 21,
 'log_count/INFO': 82,
 'response_received_count': 1,
 'scheduler/dequeued': 1,
 'scheduler/dequeued/memory': 1,
 'scheduler/enqueued': 1,
 'scheduler/enqueued/memory': 1,
 'start_time': datetime.datetime(2020, 8, 31, 10, 35, 26, 70876)}
2020-08-31 12:35:47 [scrapy.core.engine] INFO: Spider closed (finished)
runtime of SpiderBmbf 21.62023663520813
Pipeline for Bmbf called once

(venv) C:\Users\Vinay\PycharmProjects\callspiderproj\scrapy_django>
```

Externally added files can be added to Git
View Files Always Add Don't Ask Again

Externally added files can be added to Git // View Files // Always Add // Don't Ask Again (4 minutes ago)



```python
    def handle(self, *args, **options):
        process = CrawlerProcess(get_project_settings())
        t0 = time.time()
        for i in range(10):

            process.crawl(SpiderWiki)

        process.start()
        t1 = time.time()
        print("runtime of SpiderWiki", t1-t0)
        print("Pipeline for Wiki called once")
```

```
 'finish_time': datetime.datetime(2020, 8, 31, 10, 40, 59, 318936),
 'item_scraped_count': 1,
 'log_count/DEBUG': 21,
 'log_count/INFO': 91,
 'response_received_count': 1,
 'scheduler/dequeued': 1,
 'scheduler/dequeued/memory': 1,
 'scheduler/enqueued': 1,
 'scheduler/enqueued/memory': 1,
 'start_time': datetime.datetime(2020, 8, 31, 10, 40, 39, 240788)}
2020-08-31 12:40:59 [scrapy.core.engine] INFO: Spider closed (finished)
runtime of SpiderWiki 20.278159856796265
Pipeline for Wiki called once

(venv) C:\Users\Vinay\PycharmProjects\callspiderproj\scrapy_django>
```

Here the all the data is appended to the mydict dictionary and then it is sent to the pipeline all at once. Spider is called 10 times

Spiderbmbf will take around **21.6s**

SpiderWiki will take 20.27 - **21.6s**

Here as we can estimate different spiders will run at certain different time and approximately sending data all together and then calling pipeline once takes more time than calling the pipeline every time and saving the data.

Running the Spider by restarting the server

A form will submit the get request to the views.py from dashboard.html to the get_Call_for_Proposals(and get_Call_for_Papers) function in views.py this is done by checking the value of the form if value is Restart or Restartagain in the respected functions. Then the views.py file is first read and a random comment is generated letters will be in lower case and for loop will produce 10 random letters. Then this is written to the 1000th line of the file because it will not be a disturbance to other lines. Then all the lines read then this lines are written down back in the same format to the views.py with reading_file.writelines(list_of_lines). Then the opened file is then closed(reading_file.close()).

In dashboard.html (and Call_for_Papers)we have the form which will have the action to dashboard and this will send the get request as mentioned above and it will call the Alert_Function()which will receive the value of the button and if this is pressed then it will send a alert message on the page and then after pressing ok the server will restart as the comments will then be changed.

So basically in Pycharm the server will restart if we change the main file which is being used, so views.py is used as it will always have a function running and when a comment is changed the server will restart.

The Base command in django. This is used for command line operation and calling the spider through command line. In this class the feed paramaters need to be passed to the crawler process and not to the spider. Pass them as parameters to crawler process. Then read the current project settings and then override it for each crawler process . process = CrawlerProcess(get_project_settings())

The name of the spider should be passed as the first argument as a string, like this: process.crawl(SpiderBmbf) and of course MySpider should be the value given to the name attribute in your spider class. Process.start() will start the spider and process.stop() will stop it. So every time the server restarts the Command class is the first to run and then the spiders are crawled. Also we can run the spiders manually by python manage.py run_spider (run_spider is the file containing the Command class), here we have the same process in views.py and Command class will run automatically on restarting the server.

So when Run_Spiders button is pressed then the form will receive the request and will be received in the views.py with from the path in url.py, then the function(get_Call_for_Papers and get_Call_for_Proposals) will run to change the comment in the views.py and then so the server is restarted and Command class will run to start the process.

Project Tree

In the scrapy_django directory we have a django_project.

Questions is the django app.

Commands is directory created to run the spider from the command line.

Migrations are created when models are to be migrated to the database.

Filters.py is where the filters and we can use this in the functions in views.py

Models.py is the file which contains the models for migrating to database.

Serializers.py is the file used in restframework to convert to json format etc used by the class bmbfjson and Wikijson
Urls.py is the url which connects templates with the views.py to make and send http response methods.

Views.py this is the main controller of the app, which contains functions to perform all the tasks like receiving request and will returning the response.

Scrapy_project is the folder inside scrapy_django and it will contain all the files related to scrapy.

Spiders.py In spiders directory will contain the main spider file which is used for crawling.

Items.py is the file which has the containers to store the crawled data.

Pipelines.py will contain process item function to save te data into data base with help of the models created in django.

Templates directory will contain all the html files.

Dashboard.html contains the items to display the Call for Proposals.

Call_for_Proposals_Search_Result.html will display all the searched results of the call for proposals of bmbf.

Call_for_Papers.html contains all the items to display the Call for Papers.

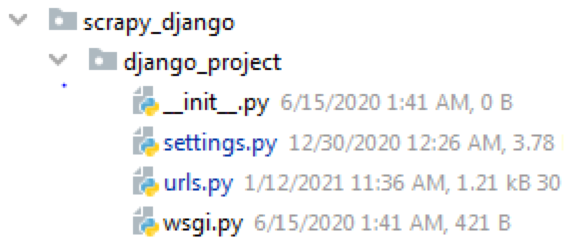
Call_for_Papers_Search_Result.html will display all the searched results of the call for papers of Wikicfp.

BmbfJson and WikiJson are for displaying the Json results.

```
├───scrapy_django
│   ├───django_project
│   │   └───__pycache__
│   ├───questions
│   │   ├───management
│   │   │   ├───commands
│   │   │   │   └───__pycache__
│   │   │   └───__pycache__
│   │   ├───migrations
│   │   │   └───__pycache__
│   │   ├───static
│   │   └───__pycache__
│   ├───scrapy_project
│   │   ├───scrapy_project
```

```
|   |   |   ├───commands
|   |   |   |   └───__pycache__
|   |   |   ├───spiders
|   |   |   |   └───__pycache__
|   |   |   └───__pycache__
|   |   └───__pycache__
|   ├───templates
|   |   └───registration
|   └───__pycache__
```

In the tree we can see that the scrapy_django is the main directory which contains the django project, app and scrapy project.



```
scrapy_django
  django_project
    __init__.py 6/15/2020 1:41 AM, 0 B
    settings.py 12/30/2020 12:26 AM, 3.78
    urls.py 1/12/2021 11:36 AM, 1.21 kB 30
    wsgi.py 6/15/2020 1:41 AM, 421 B
```

Django_project is the django project and it will contain mainly setting file with the url file. Setting file will have all the setting and configuration like:

Setting file:

Installed apps: contains the app which we have started in django and others which we will use like the restframework etc.

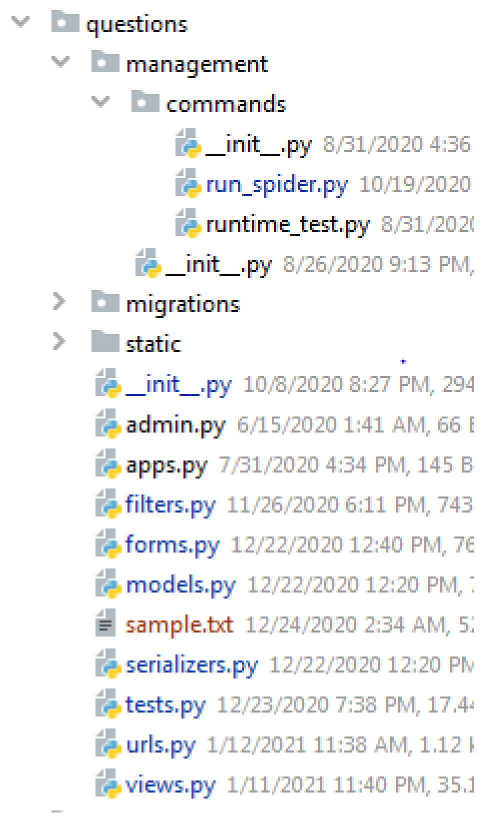
root url configuration: here specify the url file of the django , database configurations, template path, login and its redirecting path, rest framework details for using the django filters.

Database: Specify the path to the sqlite (default configuration)

LOGIN_REDIRECT_URL: is used to direct it to the page after login
REST_FRAMEWORK: uses the django filters to filter and give the json outputs.

Url file:

In the Url file just the url paths are specified here we will tell the setting file to go to the django app through this url and also restframework path is also specified here.
Admin page url is given as admin/ once we go to this page we can access the admin page of django.



```
questions
    management
        commands
            __init__.py 8/31/2020 4:36
            run_spider.py 10/19/2020
            runtime_test.py 8/31/202
        __init__.py 8/26/2020 9:13 PM,
    migrations
    static
    __init__.py 10/8/2020 8:27 PM, 294
    admin.py 6/15/2020 1:41 AM, 66 I
    apps.py 7/31/2020 4:34 PM, 145 B
    filters.py 11/26/2020 6:11 PM, 743
    forms.py 12/22/2020 12:40 PM, 76
    models.py 12/22/2020 12:20 PM,
    sample.txt 12/24/2020 2:34 AM, 5
    serializers.py 12/22/2020 12:20 PM
    tests.py 12/23/2020 7:38 PM, 17.4
    urls.py 1/12/2021 11:38 AM, 1.12 I
    views.py 1/11/2021 11:40 PM, 35.1
```

Django App(questions**)**:In the django_project a questions app is created in this we have the main files like:

In the commands folder we have run_spider.py which contains the command line functions to run the spiders.

Migrations directory contains all the previous migrations of the models to the database.

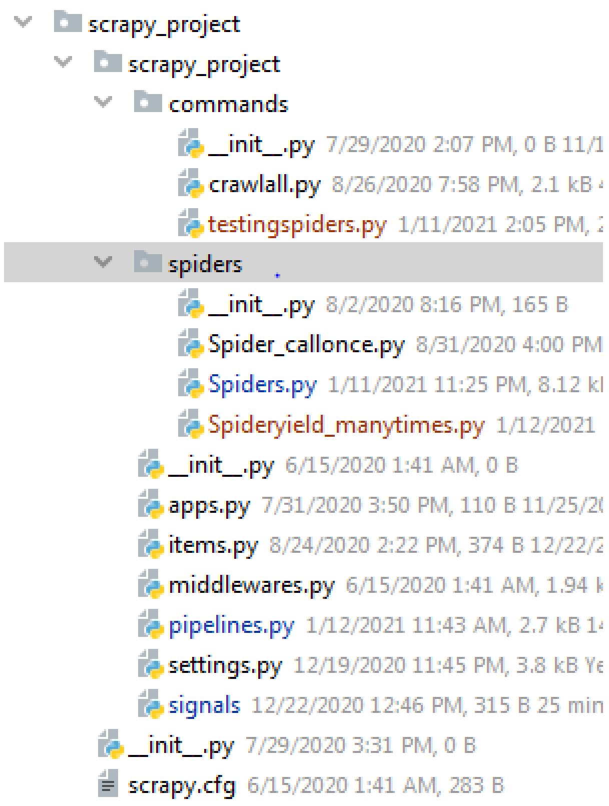
In the filters just the fileds which have to be filtered by using the models are present.

Models.py contains the 2 models which are represented as classes and the fields which are to be used.

Serializers.py this contains the classes of the models and and then their fieldsa are mentioned which is used for the filtering and searciing and get it in the json format.

Urls.py will contain the paths which are the connections between the views and the tempaltes.

Views.py will have all the functions and datas which has to be manipulated and then displayed on the templates.



```
scrapy_project
  scrapy_project
    commands
      __init__.py 7/29/2020 2:07 PM, 0 B 11/1
      crawlall.py 8/26/2020 7:58 PM, 2.1 kB
      testingspiders.py 1/11/2021 2:05 PM,
    spiders
      __init__.py 8/2/2020 8:16 PM, 165 B
      Spider_callonce.py 8/31/2020 4:00 PM
      Spiders.py 1/11/2021 11:25 PM, 8.12 k
      Spideryield_manytimes.py 1/12/2021
    __init__.py 6/15/2020 1:41 AM, 0 B
    apps.py 7/31/2020 3:50 PM, 110 B 11/25/2
    items.py 8/24/2020 2:22 PM, 374 B 12/22/
    middlewares.py 6/15/2020 1:41 AM, 1.94
    pipelines.py 1/12/2021 11:43 AM, 2.7 kB 1
    settings.py 12/19/2020 11:45 PM, 3.8 kB Y
    signals 12/22/2020 12:46 PM, 315 B 25 min
  __init__.py 7/29/2020 3:31 PM, 0 B
  scrapy.cfg 6/15/2020 1:41 AM, 283 B
```

Scrapy project: Scrapy_project will come under the scrapy_django directory.

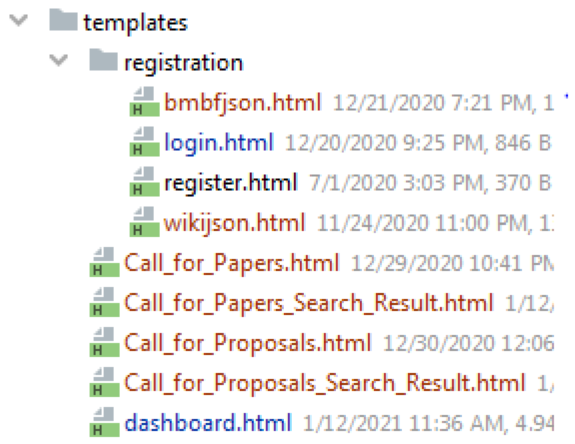
Spiders.py will contain the main spider classes which scrapes the data and send it to pipeline.

Spider_callonce.py will call the pipeline only once.

Spideryield_manytimes.py will call the pipeline or yield the data as many times the for loop runs.

Pipelines.py will contain the class to save the data into database.

Settings.py will contain the main configuration like downloader middleware, bot name and some modules of the scrapy_project.



```
templates
  registration
    bmbfjson.html 12/21/2020 7:21 PM, 1
    login.html 12/20/2020 9:25 PM, 846 B
    register.html 7/1/2020 3:03 PM, 370 B
    wikijson.html 11/24/2020 11:00 PM, 1
  Call_for_Papers.html 12/29/2020 10:41 PM
  Call_for_Papers_Search_Result.html 1/12,
  Call_for_Proposals.html 12/30/2020 12:06
  Call_for_Proposals_Search_Result.html 1,
  dashboard.html 1/12/2021 11:36 AM, 4.94
```

Templates This contains all the html files which is mainly used to display the data on the webpages.

Bmbfjson and wikijson will display json formats of the bmbf and wiki respectively.

Login.html is for the main login page.

Register.html is for the registration page.

Call_for_Papers displays the dashboard of the call for papers i.e, data of call for papers.

Call_for_Papers_Search_Result.html will display the search result of papers.

Call_for_Proposals_Search_Result.html will display the search result of proposals.

Dashboard,html will display the bmbf data as dashboard.

Future work:

1> Run spider without command keyword here I on pressing the button server will restart because in pycharm if anything is changed in the code it will restart the server so next check a command to restart the server and then connect it to RunSpiders button and replace the Base command keyword.

2> If the Runspiders button is pressed then alert window shows wait for 15 seconds but once this is done a form is submitted and the whole page is refreshed so we can't make a timer for it so by using AJAX we can submit form without refreshing the whole page and we can implement the countdown timer.

3> Here the xpath change and its items are tested but we can extend to write the integration test for the whole spiders.

References:

Figure 1: https://docs.scrapy.org/en/latest/topics/architecture.html

Figure 2 and 3: https://djangobook.com/mdj2-django-structure/

Scrapy is from its standard documentation https://docs.scrapy.org/en/latest/

Django is from its standard documentation https://docs.djangoproject.com/en/3.1/

BaseCommand:

1> https://www.webforefront.com/django/managementcommands.html

2> https://stackoverflow.com/questions/59305153/custom-django-management-command

Django is not for production: https://vsupalov.com/django-runserver-in-production/

JSON API Restframework: https://www.django-rest-framework.org/api-guide/filtering/#api-guide