DESIGN AND IMPLEMENTATION OF A GEOSCIENCE & TECHNOLOGY MARKETPLACE WEB PORTAL

A Thesis

by

BENKE QU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jyh-Charn (Steve) Liu
Committee Members,	Duncan M. (Hank) Walker
	Guofei Gu
	Jiang Hu
	Michael Bishop
Head of Department,	Dilma Da Silva

May 2017

Major Subject: Computer Engineering

ABSTRACT

The geoscience & technology marketplace aims to provide a common geoscientific data and publication sharing and transaction platform to improve the flexibility of carrying out high performance large scale scientific computing problems and facilitating the experts to carry out their research work. To achieve this goal, the geoscience & technology marketplace builds a web based information management pipeline to support end to end scientific data processing in an interactive fashion. So, as a resource management server, the marketplace provides the following major modules: gis data uploading and tiling; relational database for indexing and tracking of datasets; browsing and editing the stored gis data and their details; data processing of selected areas on google map and its result visualization and management; data transaction. The platform maintains the research credits for each register for data purchase and records each transaction produces when he buys some data; researchers' publication uploading and management; relational database for indexing and tracking of uploaded publications; browsing and editing of the stored publication data and their details and publication keywords extraction service for purchased publications, etc.

DEDICATION

To my parents,

my sister and brother in law,

and my little niece.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Jyh-Charn Liu, and my committee members, Dr. Walker, Dr. Gu, Dr. Bishop, and Dr. Hu, for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my mother and father for their love and support, and to my sister's family for their love and encouragement.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Jyh-Charn Liu, Professor Duncan M. "Hank" Walker, Professor Guofei Gu and Professor Jiang Hu of the Department of Electrical and Computer Engineering and Professor Michael Bishop of the Department of Geography.

The connection between the marketplace platform and petabyte scale file system was figured out by Ryan Vrecenar. He’s also responsible to process the created jobs and generate outputs for the marketplace platform to digest. Xing Wang provides the keyword weight list generation service. Jason Lin helped maintain the platform on the local server.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

GIS	Geographic Information System
HPC	High Performance Computing
API	Application Program Interface
CPU	Central Procesing Unit
GPU	Graphics Processing Unit
GDAL	Geospatial Data Abstraction Library
MVC	Model-View-Controller
MVT	Model-View-Template
Ajax	Asynchronous JavaScript and XML
SSH	Secure Shell
SRID	Spatial Reference System Identifier
C/S	Client/Server
B/S	Browser/Server
WSGI	Web Server Gateway Interface
URL	Uniform Resource Locator
SQL	Structured Query Language
ORM	Object-Relational Mapping
OO	Object-Oriented
ER	Entity-Relationship
DDL	Data Definition Language

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

LIST OF TABLES xi

1. INTRODUCTION 1

    1.1 System overview 1
    1.2 Related work 5
    1.3 System structure 7
    1.4 Preliminaries 9
        1.4.1 GDAL 9
        1.4.2 Web development: B/S, WSGI, MVC and MVT 11
    1.5 System development 15
    1.6 System deployment 19

2. DESIGN AND IMPLEMENTATION OF MODELS 23

    2.1 Introduction to Django models 23
    2.2 Entity-relationship diagram 26
    2.3 User 28
    2.4 GisFile 29
    2.5 Tile 30
    2.6 Publication 31
    2.7 Transaction and transaction_publication 33

3. DESIGN AND IMPLEMENTATION OF VIEWS AND TEMPLATES 35

Page

3.1 Introduction to Django view and template 35
3.2 Register 38
3.3 Login and logout 41
3.4 Retrieve 44
3.5 Upload 46
3.6 Search 52
3.7 Detail and save 56
3.8 Order and payment 60
3.9 Process, calculation and result 64
3.10 Upload_publication 71
3.11 Search_publication 74
3.12 Detail_publication and save_publication 77
3.13 Process_publication 80

4. CONCLUSIONS 82

REFERENCES 83

LIST OF FIGURES

FIGURE		Page
1	Tradition GIS software vs CyberGIS	6
2	CyberGIS as a service vendor	7
3	System structure of the marketplace platform	8
4	Apache, WSGI and MVC	13
5	MVC structure	14
6	Marketplace code structure	17
7	Relationships among objects, model, ORM and database	23
8	All the tables of the database of the marketplace	25
9	Entity-relationship diagram among Django models	27
10	From request to response, what does Django do?	37
11	The workflow of the register view	38
12	The register.html template	40
13	The workflow of the login view	42
14	The login.html template	43
15	The workflow of the retrieve view	45
16	The retrieve.html template	46
17	The workflow of the upload view	47
18	The upload.html template	50
19	The workflow of the search view	53

FIGURE Page

20 The search.html template 55

21 The workflow of the detail view 57

22 The detail.html template with the “Buy this data” button..... 58

23 The detail.html template with the “Edit this data” button..... 59

24 The detail.html template with the “Save this data” button..... 59

25 The order.html template with the “credit point” option expanded..... 61

26 The order.html template with the “credit card” option expanded 62

27 The workflow of the payment view 63

28 The process.html template..... 67

29 The possible relative positions of a tile and a selectArea query window... 68

30 The workflow of the calculate view..... 70

31 The workflow of the upload_publication view 72

32 The upload_publication.html template..... 73

33 The workflow of the search_publication view..... 75

34 The search_publication.html template 76

35 The workflow of the detail_publication view 77

36 The detail_publication.html template..... 79

37 The process_publication.html template..... 81

LIST OF TABLES

TABLE		Page
1	The marketplace application buildup commands	16
2	System development server Apache HTTP Server and its WSGI plugin	20
3	Httpd.conf configurations	21
4	Settings.py configurations	21
5	Django database settings	24
6	The marketplace application registration	25
7	Django ORM commands	26
8	The User model	28
9	DDL corresponding to the user model	28
10	The GisFile model	29
11	DDL corresponding to the GisFile model	30
12	The Tile model	31
13	DDL corresponding to the Tile model	31
14	The Publication model	32
15	DDL corresponding to the Publication model	33
16	The Transaction model	34
17	DDL corresponding to the Transaction model	34
18	The Transaction_publication model	34
19	DDL corresponding to the Transaction_publication model	34

TABLE Page

20 Urls.py 35

21 The code snippet of the register view 39

22 The message box in the register.html 40

23 The code snippet of the login view 41

24 Md5 digest calculation of the password input by a user 44

25 Configurations for email in settings.py 45

26 The code snippet of the upload view 48

27 The code snippet of the tileRaster() and generateTiles() functions 49

28 The base.html webpage 51

29 The frontend javascript function dealing with file uploading 51

30 The code snippet of the search view 54

31 The code snippet of the detail view 57

32 The code snippet of the payment view 64

33 The code snippet of the initMap() function that draws the bounding box of the data 65

34 The code snippet of the initMap() function that prepares the selectArea .. 66

35 The code snippet of the upload_publication view 72

36 The code snippet of the search_publication view 74

37 The code snippet of the detail_publication view 78

38 The code snippet of the process_publication view 80

1. INTRODUCTION

1.1 System overview

CyberGIS [1-6] created by Dr. Shaowen Wang proposed a large scale system architecture to support transformation and integration of advanced Geographic Information System (GIS) software pieces into networked services for end users. As the CyberGIS research and development ecosystem begins to take shape, the problems of interest to the community grow as well. Both the depth and breadth of professional expertise and large data accessibility required to solve those super scale problems will elevate. How to build a common data storage and management center to serve the core missions of leading research and education while meeting the stakeholders' daily missions require novel solutions to shape the future generation of geoscience and technology.

Inspired by CyberGIS, the proposed geoscience&technology Marketplace aims to provide a common data and expertise resource sharing and transaction platform to improve the flexibility of carrying out high performance large scale scientific computing problems and facilitating the registered experts to carry out their research work. In other words, it provides the data users with a common platform to directly interact with data and thus waives the users of the necessity to personally own the data and the related data manipulation software. To achieve this goal, the marketplace builds up a web based information management portal, which builds up the pipeline from data publication, transaction to processing, and thus supports end to end scientific data manipulation in an

interactive fashion. So, as a resource management server, the marketplace tries to provides the following major modules:

(a) GIS data uploading and tiling. The marketplace user can publish geoscientific data that may be interesting for other possible subscribers. Using this module, the user can upload his GIS Data via web browser. He can also attach other useful descriptive properties, e.g. data generation time, along with the uploaded GIS Data to let other potential subscribers to know more about the data as a merchandise. One property is, the user can choose if he would like the to-be-uploaded data public for view, purchase and process or private for personal usage. Along with the uploading behavior, the user can also choose to let the backend system generate tiles for his uploaded data per the specified tile size in the webpage.

(b) Centralized indexing and tracking of datasets. Instead of storing and managing his own data in his personal computer, the marketplace provides the ability so that the user doesn't have to allocate any physical hard drive space to host his own data. Instead, all the uploaded data is managed collectively in the petabyte scale file system powered by Terra, the high-performance computing (HPC) system, which is a resource for research and discovery technically sponsored by the Division of Research, Texas A&M University, College Station [7]. The user just uploads his data; the marketplace does the rest. The marketplace will communicate with the petabyte file system to store them in a place configurable in the program.

(c) Browsing and editing the stored GIS data and their details. The web frontend provides the ability for the marketplace user to browse the GIS data published

by himself or by others who selected to make his uploaded data public for sale. For his own uploaded data, the user can further check the details of the data and edit them if necessary. The user can choose to put his data on the shelf for public sale or to manage it himself for his own private usage before he decides to publish it. For data published by others for public sale, the user can only check their details without the ability to process it before he purchases it.

(d) Data processing of selected areas on google map and its result visualization. For his own data or those he purchased, the user can operate on them by specifying the area of interest among the data he plans to do calculation on. The marketplace provides an easy to use interface powered by Google Map API for the user to draw the selected area on the google map. Furthermore, the user can further control the backend calculation resources by specifying the number of CPUs, GPUs and amount of memories for his selected area.

(e) Data transaction. When the user registers for the marketplace, the marketplace by default assigns each registered user $1,000 virtual credit points for purchase of the published data and other resources by others. When the user checks the details of the published data by others, he can choose if he would like to purchase this data for further processing work. The marketplace provides three interfaces for the user to choose the payment method, that is, the credit points, paypal, and credit card. The marketplace records each transaction produces when he buys some data; Besides, the user can earn credit points if someone purchased his published data.

(f) Publication management. Besides the GIS data, the marketplace also works as a gateway for the marketplace user to publish his own publications in hand that may be interesting for other possible subscribers. Like the uploaded GIS data, publications are also hosted in a centralized way in the petabyte file system of Terra. The marketplace also provides the user with the ability to browse, search, view, edit and save the details, and purchase his desired publications.

(g) Publication processing. Different from the processing pipeline provided for the GIS data, the marketplace processes the publication by generating its keywords and corresponding weights, list them in a tabular form and visualize them using a bar chart.

The rest of the thesis is organized as follows. In Section 1.2, we will discuss CyberGIS as related work. Following the related work, Section 1.3 will show the System Structure and its interactions with other related systems, like Terra and keyword generation service module. In Section 1.4, we will discuss the technical preliminaries, the Geospatial Data Abstraction Library (GDAL), Model-View-Control (MVC) and its variety Model-View-Template (MVT). In Section 1.5, we will discuss the application of techniques used in the development of the marketplace. In Section 1.6, we will discuss the deployment requirement of the marketplace. In Section 2, we will touch the MVT model part of the marketplace. We will discuss in detail the models User, GisFile, Tile, Publication, Transaction, etc, and their relationships. In Section 3, we will discuss the MVT views and templates of the marketplace. The view part is the business logic realization center. Every view stands for a logic block. We will discuss in detail the design and implementation of each view, e.g., register, login, retrieve, upload, search,

detail, save, order, payment, etc. Nearly every view is attached with a template that will be filled with result data and generate the final html for the browser the digest. Section 4 concludes the thesis and provides the possible direction to expand the current marketplace platform.

1.2 Related work

The marketplace is inspired by CyberGIS, which, compared with the traditional GIS software, works as a platform where readymade software functions get extracted and wrapped as service for participants' employment. The simple comparison of the two types of GIS software is shown in Figure 1. Generally, we can treat it as a cloud system. Or more specifically, it is SaaS (Software as a Service) in the eyes of the service subscribers (customers) and PaaS (Platform as a Service) for the service providers (developers). It breeds an ecosystem where computing power, service, application, platform, libraries, etc. get produced and consumed by stakeholders of multidiscipline.

For the service subscriber, he can construct his workflow through a CyberGIS portal, and CyberGIS will do the rest which includes requirement analysis, data preparation, service selection, service monitoring and auditing, data collection and synchronization and intermediate/final result presentation, e.g. data visualization and reporting, etc. For the service publisher, he survives by contributing his software (programming ability as a service, ready service, algorithms (spatial analysis, GIS, etc), libraries, etc.) and hardware (CUDA-managed single or clustered GPU(s), MPI-managed single or clustered CPU(s), data storage). So CyberGIS can be seen as a service vendor,

both recruiting and promoting service. It can also advocate free market economy whereby service subscribers can directly negotiate with service providers.

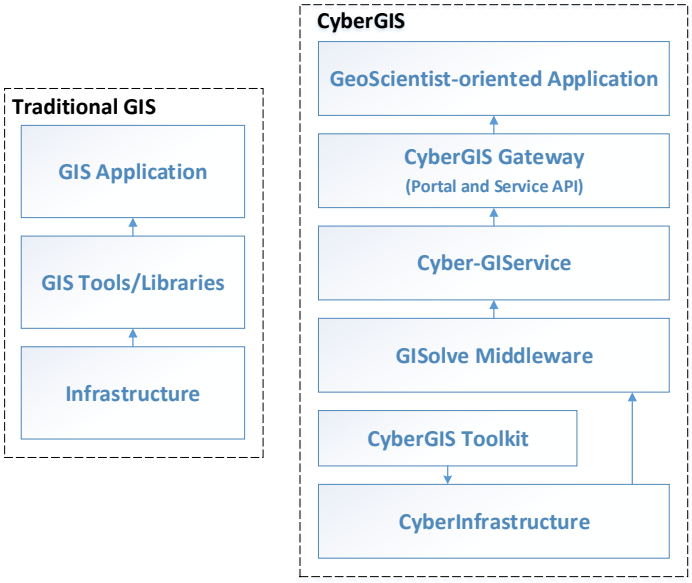

Figure 1. Tradition GIS software vs CyberGIS

CyberGIS provides the environment for a large-scale experiment for which local computing software/hardware can hardly meet its demand. It takes advantage of the web service standard to publish the local computational resource as a service for others' subscription. The service subscriber can take advantage of this servitization to carry out experiments that cost too much or are impossible when local computational resource can by no means satisfy. The CyberGIS overview is shown below in Figure 2.

Figure 2. CyberGIS as a service vendor

1.3 System structure

As a data management center, marketplace works as a middleware that interacts with remote web browser to accept user request, and to send corresponding response, and that connects the petabyte scale file system to store the uploaded user GIS data and publications, to send created jobs for Terra to digest, and to fetch generated results. The overall system structure is shown in Figure 3.

As Figure 3 shows, the whole system can be separated into three major portions, i.e., the web client portion, the local server portion, and the high performance computing portion. **The web client portion** is responsible to accept user requests to the local server in the normal http get or post way or in the Ajax way, to parse responses from the local server, and to deal with the user's interaction with the google map using the rich Google Maps Javascript API [8]. The local server portion is the central data control and communication channel. It facilitates data sharing by connecting the remote user with the remote data. For interaction with the remote web browser, starting from the user

registering or logging in the marketplace, the web browser begins to send a request to the local server.

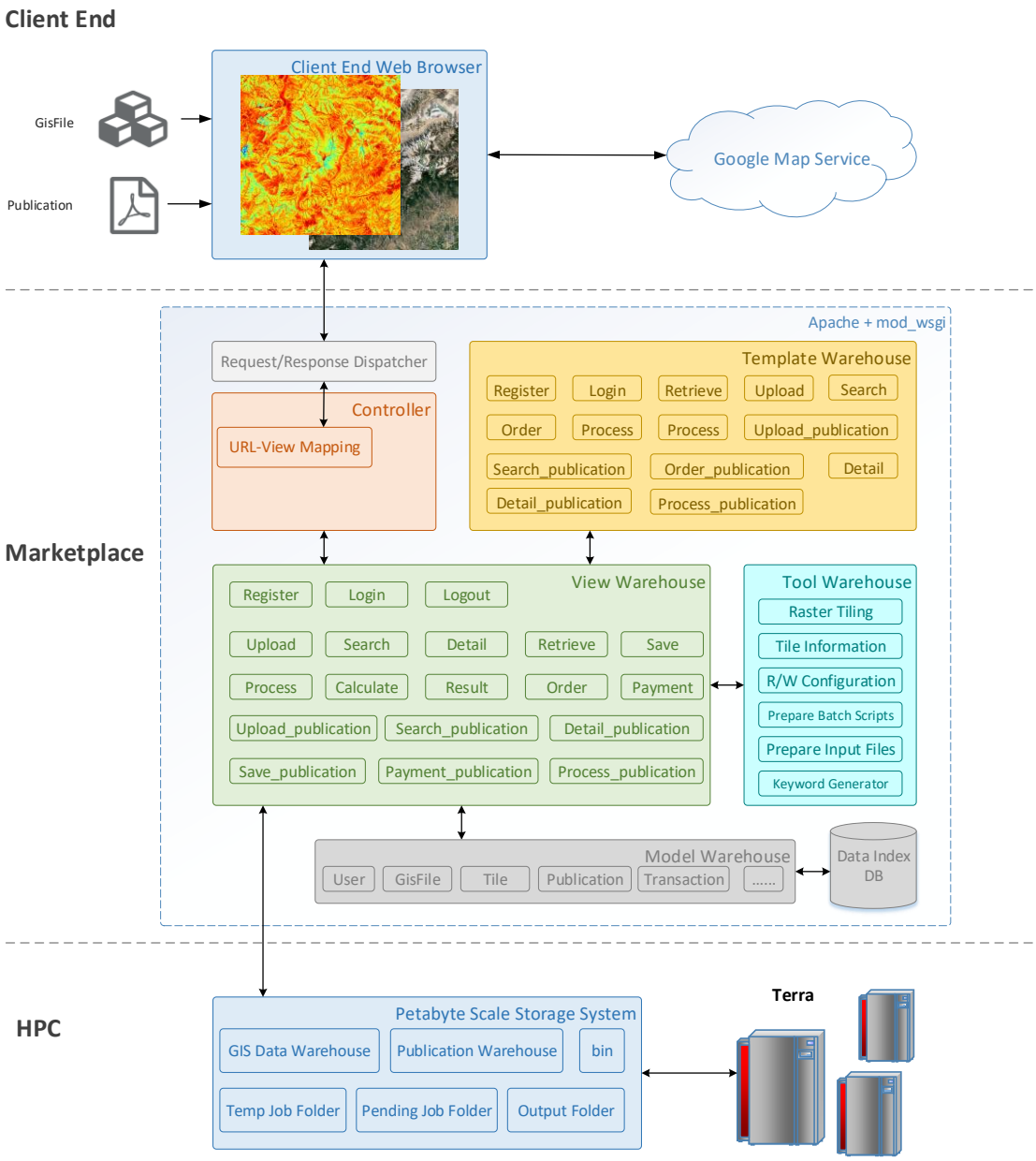

Figure 3. System structure of the marketplace platform

When the request/response dispatcher in the local server receives the request, it will redirect this request to the control module, which will refer to the routing table to look for the correct enrolled view logic which this request should be thrown at. If it finds the right view logic, the view logic gets activated. The view logic will handle the request data and prepare a response. If the request is related to data uploading and processing, the view logic needs to interact with the petabyte file system via SSH connection to store and fetch data and create jobs for calculation. The view logic may need help from some tool from the tool warehouse to handle heavy tasks, for example, raster tiling, job creation, and keyword-weight list generation, etc. As for the response, the view logic will select the right semi-product template, fill data in and generate the final html for the remote web browser to digest. The high-performance computing portion is responsible to handle data storage and job execution. Whenever there is a job created in the Pending_Jobs folder, the backend listener will read the parameter and job property file to prepare the right resources for calculation.

1.4 Preliminaries

1.4.1 GDAL

GDAL[9], or the Geospatial Data Abstraction Library, is a powerful library to deal with raster and vector data. It was released by the Open Source Geospatial Foundation[10] under the MIT license. As a library, it provides an abstract data model as a common data structure for all its user programs to deal with various data sources of different data types. It offers a convenient command line tool for the user to directly interact with the input data. For example, the command line tool named raster2pgsql can

be used to directly store the raster data into the PostgreSQL [11] database, *raster2pgsql -s 4326 -I -C -M -F -t auto testData.tif staging.testdata | psql -h localhost -U postgres -p 5432 -d postgis_in_action*.

The command example tries to load raster data named testData.tif to the table named staging.testdata. testData employs spatial reference system identifier (SRID) 4326. This command creates index on the raster column, add the standard set of constraints on the raster column, vacuum analyzes the result raster table, add filename column for the destination table. The table is created in database named postgis_in_action hosted in localhost server at port 5432. The username of postgis_in_action is postgres. With data stored in database, we can directly make use of the convenient raster data operation method provided by postgresql. In addition, using gdallocationinfo, we can check the band value of a pixel at specific location. For example, *gdallocationinfo testData.tif 50 50* can check the band value of pixel at (50, 50) in testData.tif.

Moreover, it provides API for various programming language, which facilitates the developer to manipulate the input data and generate the desired result. Fortunately, GDAL also provides an up-to-date python interface, which is extensively used for tile generation and raster information retrieval in the marketplace. The latest GDAL after version 2.0 provides even richer GDAL API in Python. In GDAL version 1, if we try to split a raster into tiles, we must call the external system function gdal_translate that can be used to extract an area from the original large raster. For example, *gdal_translate -of GTiff -srcwin 0 0 500 500 testData.tif testData_tile.tif*.

This command can be used to extract a 500x500 tile named testData_tile.tif from testData.tif at its upper left pixel location (0, 0). The generated tiff has metadata that stores the corner lat/lon, etc, which facilitates further process in our programs. Using two nested for loops to traverse the large raster, we successfully generated the standalone tiles. Now in GDAL version 2, the gdal python bindings provide us with the counterpart function for us to invoke, that is, **gdal.Translate()**. The usage of this function is shown in the following function call, *gdal.Translate(outputTilePath, inputData, srcWin = [rowPixelOffset, colPixelOffset, tileSize,tileSize])*.

This function call reads in raster data specified by inputData and generate a tile in the directory named outputTilePath. The generated tile is a subset of the original raster, that is formed using the parameters in the srcWin list. The rowPixelOffset and colPixelOffset designate the upper-left corner coordinates. The next two tileSizes specify the width and height of the tile to be generated.

1.4.2 Web development: BS, WSGI, MVC and MVT

The traditional software was developed following the C/S pattern, that is Client/Server, where the software mainly runs in the client side and the database is hosted in the server. As internet becomes more and more popular, the C/S framework find itself harder to catch up with the fast tempo. Web applications usually face up with frequent modification and update, while the nature of the C/S framework requires each client get updated one by one. This caused C/S fades away, while B/S, that is, the Browser/Server framework shoulders the responsibility to server the user. Within B/S, all the business logic and data of the application is hosted in the server side, the client

just needs a web browser to request service from the server, get response in the form of an html web page, and visualize it for the client to interact with. The highly interaction ability and representability of html waives the user of the task to deal with system deployment as well as its version update. This makes B/S win a rapid success for web development. To solve the problem of logic scripts blending with html, which makes projects rough to maintain, web development introduced the MVC [12] design pattern to simplify the development process.

Before we dive into MVC, let's first briefly talk about WSGI, namely Web Server Gateway Interface. The interaction of the web browser and the backend web server is as follows. First, the web browser send an http request to the backend web server. Second, the web server accepts the request, processes it and generates an html web page. The generate html file, as a response body, gets wrapped with a response header and sent back to the web browser. The waiting web browser receives the response, extract the response body and visualize it for the end users to digest. If the user just request a normal static html web page, then the current popular static web server, such as the Apache HTTP Server [13], Nginx [14], etc., can easily handle this, but if the user needs interaction with the web page, needs to send his personal user inputs for the backend server to parse and process, the static server alone is not enough to handle that. WSGI is the web container to accept and parse this kind of http request, host the necessary business logic to handle the request, and prepare the final response header and body for the web browser to digest. With the help of WSGI, the developer can get waived of dealing with these underlying communication tasks and concentrate on the

upper-level business logic. Based on WSGI, MVC makes the development work even easier. It works as a web framework that helps the developers deal with WSGI. The developer only needs to fill in his request process logic and provide response templates. Figure 4 shows the relationship of these modules.

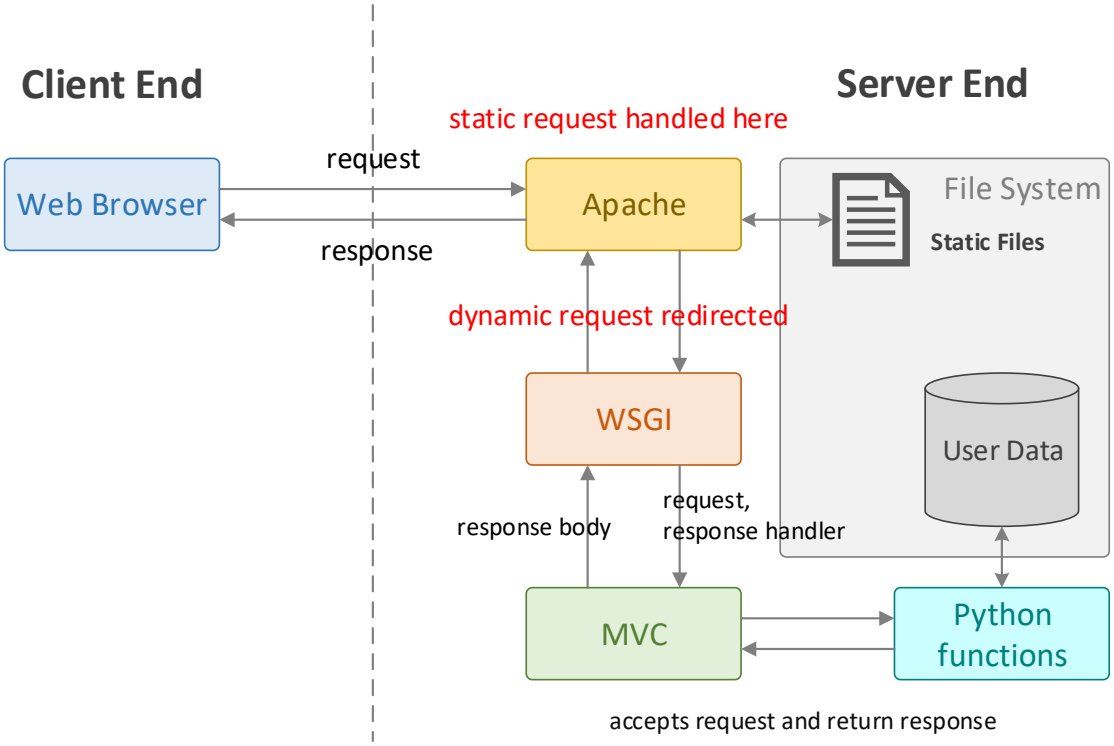

Figure 4. Apache, WSGI and MVC

MVC, an abbreviation for Model-View-Controller, is a software design paradigm. It organizes codes in a project per their different roles. This design forces the separation of program inputs, processing and program outputs. As Figure 5 shows, the codes that are used to handle users' requests and responses and maintain the work pipelines within a software are integrated and called Controller. The codes that are used

to deal with business logic and manage data are singled out and together form a module called model. Model is the core of the whole MVC structure. Next, the result of the business logic will be transferred to the View module, which, per the process result, selects the correct viewer to represent the result data and sends back the rendered result back to the web browser. The whole process ultimately forms a closed loop, where different modules take on different tasks without overlaps.

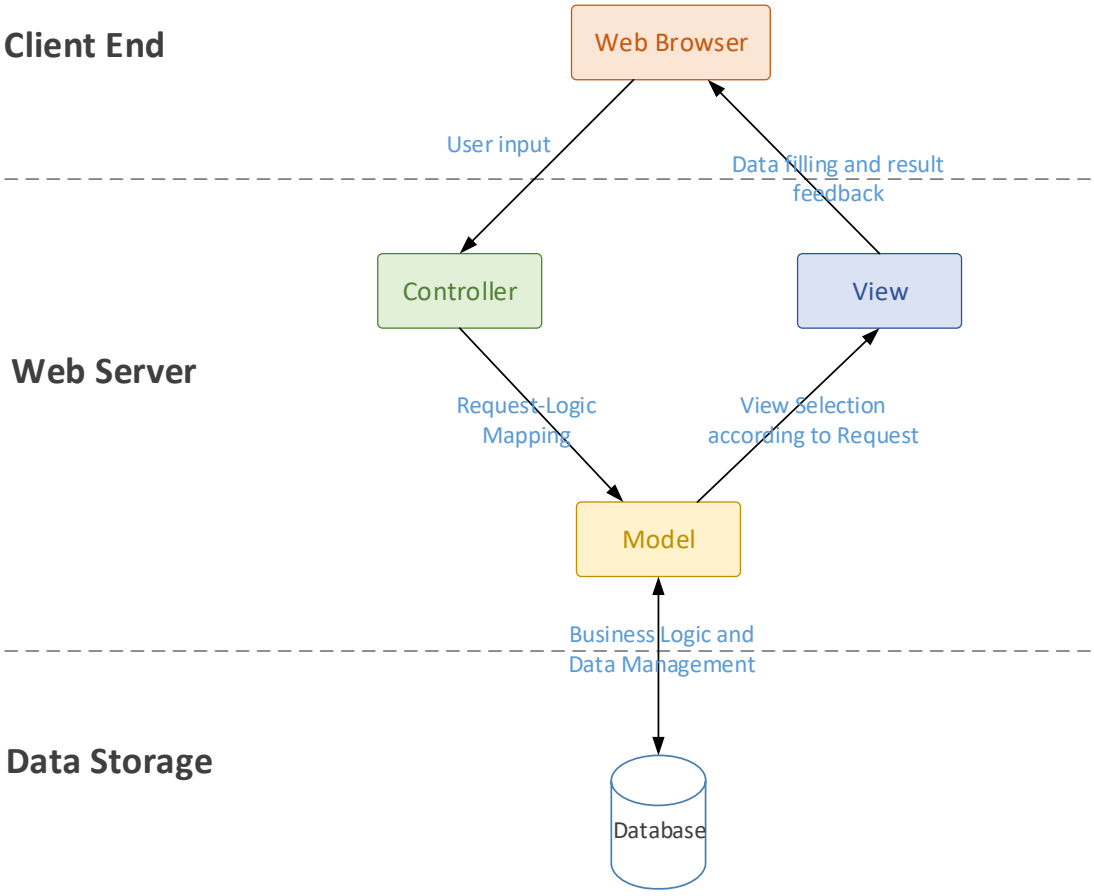

Figure 5. MVC structure

As a software design paradigm, MVC promotes loose couple of program codes. The separation of the View and Model, makes it convenient for the developers to plug in/out view modules without too much interference with the codes of Model and Controller. Likewise, the changes of business workflow or rules within a business logic or database switches will only touch Model without affecting View and Controller. All modules within MVC is self-contained. This characteristic makes each module highly reusable. With the development of technology, more and more ways are provided to access the backend applications. The View can select different views to display by telling if the request is from a mobile end or a normal PC end. Different views all share the same model. The View part can simply deal with the display part within its own module while dealing with the same business logic. So, this modular design pattern makes it easy for the development, deploy and management of an application.

1.5 System development

In this project, I selected Django [15] as a web framework to build up the whole website. Essentially, Django conforms to the MVC design pattern, but because of the design of Django, the web framework takes on the role of a controller, and leaves blank Model, View and Templates for the developers to fill in their codes, so Django is said to employ MTV, that is, Model-Template-View design pattern, in which, Model assume the work of object definition and object-relational mapping between program objects and database tables. Template is responsible to provide a html page for the developer to fill in result data, to render, and to send it back to the request user. View stands for the core business logic which invokes Model and Template at proper time. Django, as a

controller, exposes a URL dispatcher to the developer from them to fill in the rules of mapping between URLs, described in regular expressions and views, used to handle user request. The following commands are essential for use to build up the whole marketplace application, as are shown in Table 1.

#	Command	Brief Description
1	django-admin startproject webgis	Create a project
2	python manage.py startapp marketplace	Create an application within a project
3	python manage.py runserver	Start the testing server
4	python manage.py runserver 0.0.0.0:8000	Start the testing server for internet access

Table 1. The marketplace application buildup commands

Go to the specified place where the project named webgis will be created. Using Command 1, we will create the folder of the webgis project, that is the root directory of the whole project. It includes a series of auto generated directories and files, which take on different tasks. Django differentiates application and project. An application is a system created for some function. A project is the host of multiple application and their settings, which constitute the whole website. To make an app reusable and a project scalable, one project can host many applications and one application can be migrated to multiple projects without problem. Using Command 2, we create an application called marketplace in the root folder of the project. Taking consideration into the tool

warehouse created for the view logic. The whole marketplace code structure is shown in Figure 6.

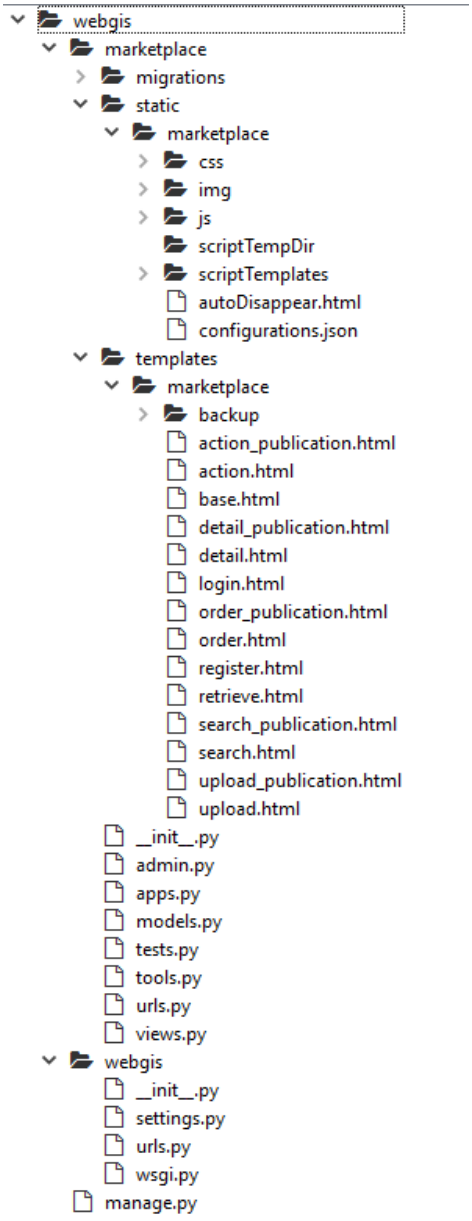

Figure 6. Marketplace code structure

Manage.py is a command line tool which is used to interact with Django for various requirement, such as application creation, model migrations, and test server startup. Settings.py in the project folder webgis is the core configuration file of the whole project. This file includes our created application configurations in INSTALLED_APPS, database configurations in DATABASES, the root path of the application static files, like javascripts and css, and mail server settings. Urls.py here does a preliminary routing work. Per the user request, it directs the user request to the corresponding application. Wsgi.py works a gateway to a WSGI web server. It abstracts the WSGI functions which deals with the underlying internet communication as an integrated application module.

In the application folder named marketplace, important folders include the static folder, which hosts all the static files, like javascript and css, the templates folder which hosts all the html templates for views to call and fill data in. Modesl.py defines all the objects used in the application. The defined objects will relate themselves to relational tables stored in the database. Tools.py was created by myself. It is used to contain all the utilities serving the business logic in views.py. These extracted functions can be safely reusable and simplify the project maintenance work. Urls.py works as a roster whereby the applications can find the entrance to a view logic module corresponding to a user's request, that is, it's a request-logic mapping reference book. Views.py contains all the functions used to accept user requests, deal with business logic, database read and write and prepare responses. It takes the most important role of a controller for the whole project.

Django provides a built-in lightweight web server for the ease of development, which waives the developers of the hardships to deal with the maintenance of an online web server. Using Command 3, we can easily start up this testing server. The ability of hot plug-in of the code updates makes is convenient for testing. By default, the built-in server listens on port 8000 for only localhost requests. If we need to let the application to accept requests from the internet, we can input IP address 0.0.0.0 to tell the test server to accept requests from different IP. Command 4 is just used to expose the whole application to the internet for public access.

But one thing that needs mention is the built-in test server can only guarantee one reliable service for one request at a time. Besides, the test server doesn't take security into consideration and may face up with security problems if the application gets deployed into the test server as a product. If the application needs deployment for public frequent and concurrent access, we need to find a reliable web server to host our project. In this project, I use the popular combination of Apache and mod_wsgi to take on this responsibility.

1.6 System deployment

The combination of Apache HTTP server and mod_wsgi [16] is a proven and popular way to host the Django project as a server for public access. The Apache HTTP server, or Apache for short, is an open web page hosting server sponsored by The Apache Software Foundation. It has a recognized ability of cross-platform and security. Apache itself can only hosts static resources, but it's open to 3rd-party plug-ins to expand its ability to handle dynamic user requests. Mod_wsgi is such a plug-in module,

which can host any Python WSGI applications, such as Django. In accordance with this requirement, Django also provides the ability to cooperate with mod_wsgi in Apache.

To accommodate the versions of python and operating system, Apache and mod_wsgi have to be carefully chosen to avoid possible compatibility problems. One thing is, Apache doesn't come with a version for windows 10 64bit. Fortunately, the Apache Haus [17] filled the vacancy. For Python version 2.7.x 64 bit, Mod_wsgi only supports Apache 2.4 built with the ASF [18] and OpenSSL sources in C, which needs the compiler provided by Visual Studio 2008 (VC9). The version of used Apache HTTP server and mod_wsgi is listed in Table 2 shown below. Go to LFD [19] to download the proper mod_wsgi named mod_wsgi-4.5.14+ap24vc9-cp27-cp27m-win_amd64.whl, decompress this file and move the file mod_wsgi.so to %APACHE_HOME%/modules.

Software	Version	Source
Apache HTTP server	2.4.25-x64-r1 VC9	Apache Haus
mod_wsgi	mod_wsgi-4.5.14+ap24vc9-cp27-cp27m-win_amd64	LFD

Table 2. System development server Apache HTTP Server and its WSGI plugin

The next deployment process mainly involves two configurations, Apache's httpd.conf and Django's wsgi.py and settings.py. First, open httpd.conf and append it with following configurations in Table 3.

```
1. LoadModule wsgi_module modules/mod_wsgi.so
2. WSGIScriptAlias / E:/workspace/workspace-python2/webgis/webgis/wsgi.py
3. WSGIPythonPath E:/workspace/workspace-python2/webgis
4. <Directory E:/workspace/workspace-python2/webgis/webgis>
5.     <Files wsgi.py>
6.             Require all granted
7.     </Files>
8. </Directory>

9. Alias /static E:/workspace/workspace-python2/webgis/static
10.<Directory E:/workspace/workspace-python2/webgis/static>
11.Require all granted
12.</Directory>
```

Table 3. Httpd.conf configurations

The command in Line 1 is used to tell Apache to include the 3rd-party module to arm itself with the ability to handle dynamic user requests. Line 2 shows the wsgi configuration file of the deployed project. Line 3 shows the wsgi application path. Line 4-8 assigns mod_wsgi the permission to access the wsgi configuration file of the project. Line 9-10 tells mod_wsgi where the static resources of the project are.

By default, Django version after 1.6.x will automatically generate the wsgi.py. There is no need for the developer to create this file. Go to settings.py and make the following changes,

```
1. DEBUG = False
2. ALLOWED_HOSTS = ['127.0.0.1', 'localhost']
3. STATIC_ROOT = os.path.join(BASE_DIR, "static")
```

Table 4. Settings.py configurations

Line 1 tells Django the deployment is not for debug and thus needs to be deployed in the mature web server. If in the future, you would like to switch back and use the built-in test server, just set DEBUG = True. Line 2 allows us to use localhost as a domain name to access the marketplace. Line 3 is used to tell where the static resources of the projects are.

Next go the root directory of the project and execute the following command, *python manage.py collectstatic*. This command will collect all the static resources, such as javascripts and css within the project and create a folder named static in the root directory to store all the static resources. This will allow the Apache HTTP Server to find where the static resources are, otherwise, the deployed project will fail to load javascript and css, etc.

2. DESIGN AND IMPLEMENTATION OF MODELS

2.1 Introduction to Django model

From this section on, we will start to touch the development details. The first part is related to interaction with database. Previously, when we need to insert/delete/update/select data stored in the database, we need to maintain our own database connection pool to handle multiple database access, compose our own SQL clause to reflect complex requirement, take care of atomic transaction to avoid dirty data, batch execution to boost execution, etc. All the nuts and bolts may severely affect the efficiency of a developer. Especially, the data access logic is scattered among the whole project codes, which makes it difficult to manage database connection. To solve the problem, ORM, that is Object-Relational Mapping was created, which aims to build mappings between business entities and the relational database tables, so that we can directly operate on the business entities. This OO design idea saves us from having to interact with databases by leaving the database management work to ORM. The basic relationship of these entities is shown in Figure 7.

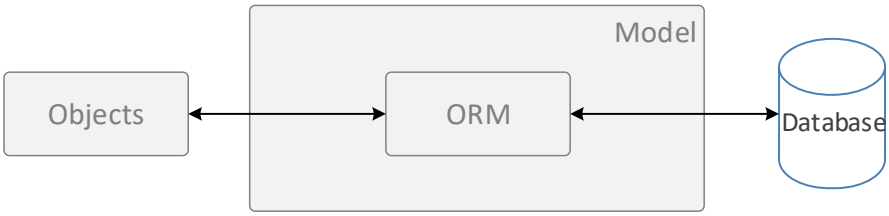

Figure 7. Relationships among objects, model, ORM and database

Fortunately, Django provides a powerful ORM ability for the developers to utilize. By providing the necessary database configuration work, the developer directly manipulate data using the API provided by ORM and thus totally ignore the interaction with database. In this project, PostgreSQL is chosen as the backend database. So, the database configuration in settings.py is shown below,

```
DATABASES = {
   "default": {
      "ENGINE": "django.db.backends.postgresql",
      "NAME": "postgis_in_action",
      "USER": "postgres",
      "PASSWORD": "admin",
      "HOST": "127.0.0.1",
      "PORT": "5432"
   }
}
```

Table 5. Django database settings

The ENGINE is the database driver name; NAME shows the database created by me in PostgreSQL. All the tables in postgis_in_action created by Django and me are shown in Figure 8.

As mentioned just now, Django makes use of ORM to manage databases. In Django, each model is represented as a Python class, and it stands for the corresponding table automatically created in the database by Django. Each object of the class stands for a row of data in the corresponding table, and each variable of the object corresponds to one column of the table. In the marketplace application, I created six models: User, GisFile, Tile, Publication, Transaction and Transaction_publication. Before we use

ORM to create corresponding tables in the database, first register the marketplace application to the project webgis in settings.py, shown below in Table 6.

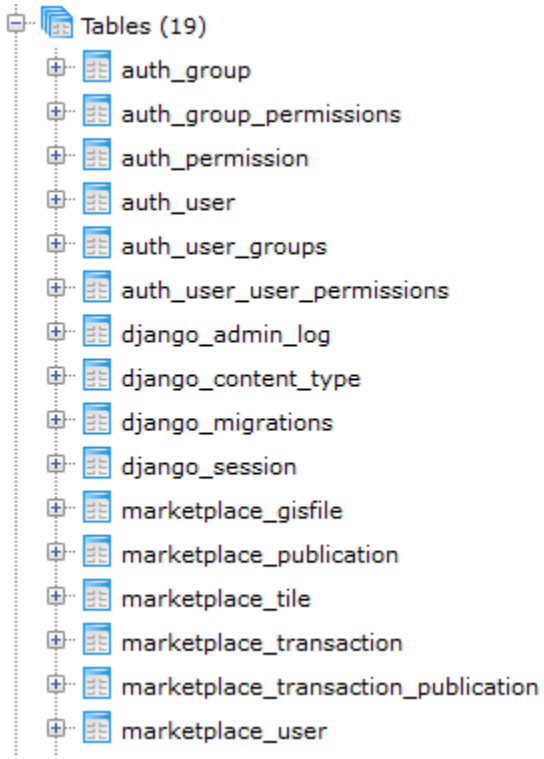

Figure 8. All the tables of the database of the marketplace

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    "marketplace.apps.MarketplaceConfig"
]
```

Table 6. The marketplace application registration

After that, we can execute the next commands shown in Table 7. In Command 3, the parameter 0001 is the number of migrations in the folder named migrations.

#	Command	Description
1	python manage.py makemigrations marketplace	create migrations to be reflected in database
2	python manage.py migrate	Migrate object changes to tables in database
3	python manage.py sqlmigrate marketplace 0001	check the sql clauses create by Command 1

Table 7. Django ORM commands

2.2 Entity-relationship diagram

As Figure 9 shows, the relationship of these models is expressed using ER diagram, that is, Entity-Relationship Diagram. In this diagram, each rectangle stands for a defined model, each ellipse stands for the properties of the model, which corresponds to the column of the corresponding table. One user gets involved with a gisFile via two kinds of relationship, that is, Transaction and Upload, depicted using diamonds. The transaction relationship is many-to-many mapping, which means one user can buy multiple gisFiles, and multiple users can buy one gisFile. Likewise, one user can upload multiple gisFiles, and one gisFile can be uploaded by multiple users. The relationship between the GisFile and Tile is a one-to-many relationship, which means one gisFile can be splitted into multiple tiles, while one tile only belongs to one gisFile. The relationship between User and Publication is the same as that between User and GisFile. In the following subsections, we will dive into the details of these models.

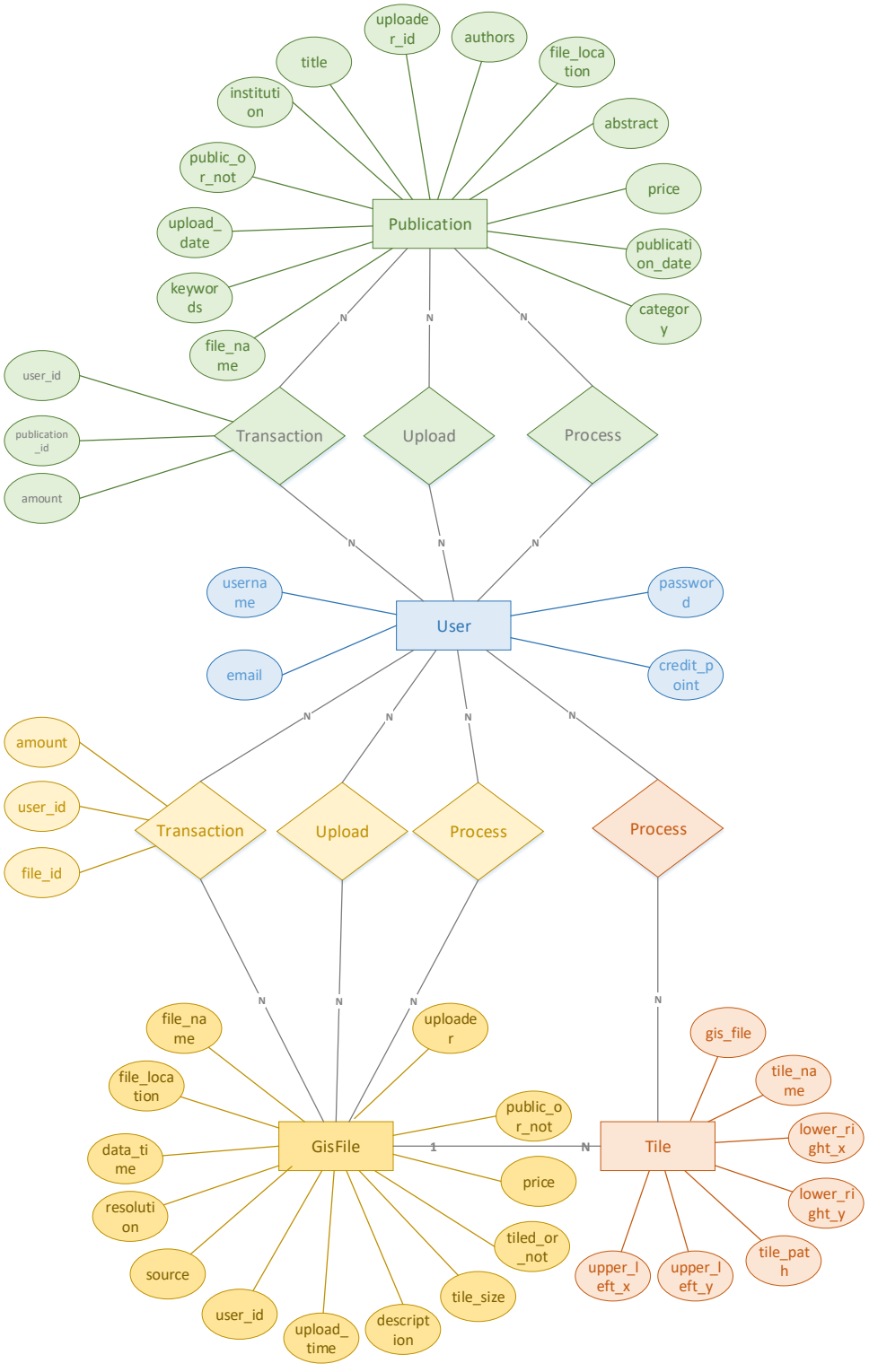

Figure 9. Entity-relationship diagram among Django models

2.3 User

Each model in Django has a superclass named models.Model, which provides rich API to manipulate the python object and reflect the manipulation to the underlying database tables. For the users of the marketplace, I created a model named User as is shown in below in Table 8.

```
from django.db import models
class User(models.Model):
    username = models.CharField(max_length = 200)
    email = models.CharField(max_length = 200)
    password = models.CharField(max_length = 200)
    credit_point = models.FloatField(default=1000)
```

Table 8. The User model

The model.Model has to be imported from django.db before use. Every time, one user registered for an account in the marketplace, my program will will create an object of this type, Django will help save information held by the created object into the table named marketplace_user. To create the corresponding table in the database, execute Command 1 and 2 in Figure. By executing Command 3, the generated DDL by Command 1 is shown below in Table 9.

```
CREATE TABLE "marketplace_user"
(
        "id" serial NOT NULL PRIMARY KEY,
        "username" varchar(200) NOT NULL,
        "email" varchar(200) NOT NULL,
        "password" varchar(200) NOT NULL
)
```

Table 9. DDL corresponding to the user model

By default, each table generated by Django is attached with a default field named id. This field is default primary key in that table, and can be used as a foreign key in other tables when necessary. These fields are self-explanatory.

2.4 GisFile

The GisFile model is created to hold information of the gis file uploaded by the user. And for one user, he can do the following operations on a gisFile, that is, Upload, Transaction and Process. Upload means the user can upload a gisFile, Transaction means the user can buy a data, and Process means the user can do calculations on the purchased data. The GisFile model is shown below in Table 10.

```
class GisFile(models.Model):
    file_name = models.CharField(max_length = 200)
    file_location = models.CharField(max_length = 1000)
    data_time = models.CharField(max_length = 200, blank=True, null=True)
    resolution = models.FloatField(blank=True, null=True)
    source = models.CharField(max_length = 200, blank=True, null=True)
    public_or_not = models.IntegerField()
    price = models.FloatField(blank=True, null=True)
    uploader = models.CharField(max_length = 200, blank=True, null=True)
    upload_time = models.CharField(max_length = 200, blank=True, null=True)
    tiled_or_not = models.IntegerField()
    tile_size = models.IntegerField(blank=True, null=True)
    user_id = models.IntegerField(default=1)
    description = models.CharField(max_length = 2000, blank=True, null=True)
```

Table 10. The GisFile model

The field named data_time stores the time when the data was collected. The resolution is the pixel width of the gis data, which means how many meters a single pixel stands for. Source means the sponsor of the data or who generated the data. Public_or_not is used to record if the uploader of this data would like to publish the data

for sale. If yes, other users of the marketplace can see this uploaded data in the search page, otherwise, the data is for private use only or not ready for sale yet. Price tells the other users how much they must pay if they would like it. Tiled_or_not is a field to record if the user selected to tile this uploaded file. If yes, the field tile_size becomes significant. The user_id is provided by the session when the user logs in. In addition, the field that receives blank=True, null=True will allow for blank input and the generated DDL will set that column NULL instead of NOT NULL. The DDL for GisFile is generated as follows in Table 11.

```
CREATE TABLE "marketplace_gisfile"
(
        "id" serial NOT NULL PRIMARY KEY,
        "file_name" varchar(200) NOT NULL,
        "file_location" varchar(1000) NOT NULL,
        "data_time" varchar(200) NULL,
        "resolution" double precision NULL,
        "source" varchar(200) NULL,
        "public_or_not" integer NOT NULL,
        "price" double precision NULL,
        "uploader" varchar(200) NULL,
        "upload_time" varchar(200) NULL,
        "tiled_or_not" integer NOT NULL,
        "tile_size" integer NULL,
        "user_id" integer NOT NULL,
        "description" varchar(2000) NULL
)
```

Table 11. DDL corresponding to the GisFile model

2.5 Tile

When the user uploads a gisFile, he can select to generate tiles for the uploaded file at the same time. If he chooses to do that, a Tile object will be created and stored into the database. The Tile object is shown below in Table 12.

```
class Tile(models.Model):
    gis_file = models.ForeignKey(GisFile, on_delete=models.CASCADE)
    tile_name = models.CharField(max_length = 200)
    upper_left_x = models.FloatField()
    upper_left_y = models.FloatField()
    lower_right_x = models.FloatField()
    lower_right_y = models.FloatField()
    tile_path = models.CharField(max_length = 1000)
```

Table 12. The Tile model

For a generated tile, it has a foreign reference to its parent gisFile, which is named gis_file. For ease of later calculation, the tile's corner coordinates are also recorded. Tile_path shows where the tile is stored in the petabyte scale file system. The corresponding DDL is shown in Table 13. In the generate DDL, we can see that the foreign reference is translated to gis_file_id. In addition, when in the process page, the user chooses to use tiles for calculation, the backend server will find the proper tiles for calculation.

```
CREATE TABLE "marketplace_tile" (
        "id" serial NOT NULL PRIMARY KEY,
        "tile_name" varchar(200) NOT NULL,
        "upper_left_x" double precision NOT NULL,
        "upper_left_y" double precision NOT NULL,
        "lower_right_x" double precision NOT NULL,
        "lower_right_y" double precision NOT NULL,
        "tile_path" varchar(1000) NOT NULL,
        "gis_file_id" integer NOT NULL)
```

Table 13. DDL corresponding to the Tile model

2.6 Publication

The Publication model is created to hold information of the publication uploaded by the user. And for one user, he can do the following operations on a publication, that

is, Upload, Transaction and Process. Upload means the user can upload a publication, Transaction means the user can buy a publication, and Process means the user can do calculations on the purchased publications. The Publication model is shown below in Table 14.

```
class Publication(models.Model):
    title = models.CharField(max_length=1000, blank=True, null=True)
    category = models.IntegerField(blank=True, null=True)
    authors = models.CharField(max_length=1000, blank=True, null=True)
    institution = models.CharField(max_length=200, blank=True, null=True)
    publication_date = models.CharField(max_length = 200, blank=True, null=True)
    keywords = models.CharField(max_length = 1000, blank=True, null=True)
    price = models.FloatField(blank=True, null=True)
    public_or_not = models.IntegerField(blank=True, null=True)
    abstract = models.CharField(max_length=10000, blank=True, null=True)

    file_name = models.CharField(max_length = 200, blank=True, null=True)
    file_location = models.CharField(max_length = 1000, blank=True, null=True)
    uploader_id = models.IntegerField()
    upload_date = models.CharField(max_length = 200, blank=True, null=True)
```

Table 14. The Publication model

The title, authors, institution, keywords, abstract and publciation_date are self-explanatory. The category field is used to record if the publication is a conference paper or a journal one. Public_or_not is used to record if the uploader of this publication would like to publish it for sale. If yes, other users of the marketplace can see this uploaded publication in the search page, otherwise, the publication is for private use only or not ready for sale yet. Price tells the other users how much they must pay if they would like it. The uploader_id is provided by the session when the user logs in. The DDL for Publication is generated as follows in Table 15.

```
CREATE TABLE "marketplace_publication"
(
        "id" serial NOT NULL PRIMARY KEY,
        "title" varchar(1000) NULL,
        "category" integer NULL,
        "authors" varchar(1000) NULL,
        "institution" varchar(200) NULL,
        "publication_date" varchar(200) NULL,
        "keywords" varchar(1000) NULL,
        "price" double precision NULL,
        "public_or_not" integer NULL,
        "abstract" varchar(10000) NULL,
        "file_name" varchar(200) NULL,
        "file_location" varchar(1000) NULL,
        "uploader_id" integer NOT NULL,
        "upload_date" varchar(200) NULL
)
```

Table 15. DDL corresponding to the Publication model

2.7 Transaction and Transaction_publication

A Transaction object is generated when one user purchased one gis file. A Transaction_publication object is generated when one user purchased one publication. Here the two kinds of transactions are separated, which needs consolidation in the future. When I designed the Transaction model, the requirement only involves GisFile, so the transaction model added a field named file_id. Later, Publication was required, but the file_id is only the id for GisFile. Treating file_id as a common id column for GisFile and Publication will cause problems, because both ids of GisFile and Publication are numbered from 1 in order, which will cause conflicts. In the next version of Transaction, it will include a field called category to record which item was bought, GisFile or Publication or other item that may be included in the future. The Transaction model is shown below in Table 16.

```
class Transaction(models.Model):
   user_id = models.IntegerField()
   file_id = models.IntegerField()
   amount = models.FloatField(blank=True, null=True)
```

Table 16. The Transaction model

Its DDL is shown in Table 17.

```
CREATE TABLE "marketplace_transaction"
(
                "id" serial NOT NULL PRIMARY KEY,
                "user_id" integer NOT NULL,
                "file_id" integer NOT NULL,
                "amount" double precision NULL
)
```

Table 17. DDL corresponding to the Transaction model

The Transaction_publication model is shown below in Table 18.

```
class Transaction_publication(models.Model):
   user_id = models.IntegerField()
   publication_id = models.IntegerField()
   amount = models.FloatField(blank=True, null=True)
```

Table 18. The Transaction_publication model

Its DDL is shown in Table 19.

```
CREATE TABLE "marketplace_transaction_publication"
(
                "id" serial NOT NULL PRIMARY KEY,
                "user_id" integer NOT NULL,
                "publication_id" integer NOT NULL,
                "amount" double precision NULL
)
```

Table 19. DDL corresponding to the Transaction_publication model

3. DESIGN AND IMPLEMENTATION OF VIEWS AND TEMPLATES

3.1 Introduction to Django view and template

In this section, we will talk about the core of the marketplace, business logic and result visualization. Each subsection is an integrated description of the implementation of both the view and template. Before we dive into each subsection, I will first briefly talk about Django's control part. Django takes on most of the work on the control part, only exposing a url.py file for the developer to fill in the mapping rule between the request url and view logic. Urls.py is just a standard Python file and supports dynamic configuration. It works as a route map to connect one url pattern and a view function logic. The url pattern is composed using regular expression [20], which defines the way how user's request data is attached conforming to an elegant url design. For example, r'^detail/(?P<gisFileId>[0-9]+)/$' simply replaces the normal request url, detail?gisFileId=21 with a much simpler version, detail/21. The view function will receives a request parameter named gisFileid valued at 21. The overall urlpatterns in urls.py is shown below in Table 20.

```
app_name = "marketplace"
urlpatterns = [
    url(r'^$', views.login, name="index"),
    url(r'^upload/$', views.upload, name = "upload"),
    url(r'^search/$', views.search, name = "search"),
    url(r'^action/(?P<gisFileId>[0-9]+)/$', views.action, name = "action"),
    url(r'^calculate/$', views.calculate, name = "calculate"),
    url(r'^result/$', views.result, name="result"),
    url(r'^login/$', views.login, name="login"),
```

Table 20. Urls.py

```
    url(r'^register/$', views.register, name="register"),
    url(r'^retrieve/$', views.retrieve, name="retrieve"),
    url(r'^logout/$', views.logout, name="logout"),
    url(r'^detail/(?P<gisFileId>[0-9]+)/$', views.detail, name="detail"),
    url(r'^save/$', views.save, name="save"),
    url(r'^order/$', views.order, name="order"),
    url(r'^payment/$', views.payment, name="payment"),
    url(r'^upload_publication/$', views.upload_publication, name="upload_publication"),
    url(r'^search_publication/$', views.search_publication, name = "search_publication"),
    url(r'^detail_publication/(?P<publication_id>[0-9]+)/$', views.detail_publication,
        name = "detail_publication"),
    url(r'^save_publication/$', views.save_publication, name="save_publication"),
    url(r'^order_publication/$', views.order_publication, name="order_publication"),
    url(r'^payment_publication/$', views.payment_publication, name="payment_publication"),
    url(r'^process_publication/(?P<publication_id>[0-9]+)/$', views.process_publication,
        name="process_publication")
]
```

Table 20. Continued

In each url() function, the first parameter defines the rule to match coming request urls. each url pattern begins with ^ and ends with $, which mean matching start and end, respectively. P<parameterName> defines the name of the group captured by its following regular expression. The second parameter designates a view callback function name, which will be activated when the first url parameter get matched. the third parameter named name is used for the template to designate which url pattern he’s going to match and thus which view he’s going to send his data to.

A view in Django is used to accept a user request, carry out business logic to deal with models and prepare response body. Each view accepts an HttpRequest object as its first parameter, and the parameter matched by the url pattern is attached with following parameters. Each response prepared by a view is an HttpResponse object. Both HttpRequest and HttpResponse are defined in the django.http module.

A web application not only has to deal with business logic, but also take great consideration into the web pages to be visualized in front of the users. Simply wrapping an html string into the HttpResponse will make the development work boring and prone to error. So, Django provides a template mechanism to prepare a semi-finished product which incorporates a series of Django directives programmed using the Django template language. The Django template functions will inflate a template with data produced by a view. A simple example is shown below in Figure 10. This figure also gives us a brief overview of the whole process template of the following view+template modules.

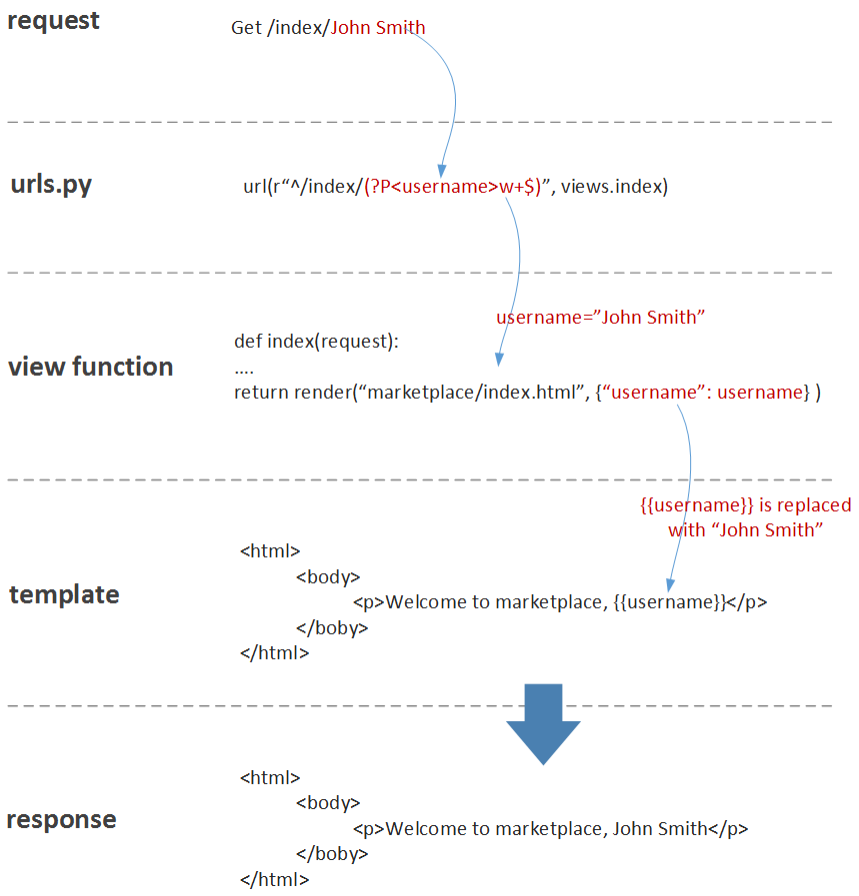

Figure 10. From request to response, what does Django do?

3.2 Register

The register view and its attached template register.html are used to realize the ability for the user to register for an account in the marketplace platform. Actually each view is a specific function in views.py. In views.py the function register(request) is the corresponding view function. It accepts a HttpRequest object and returns HttpResponse. The main workflow of the view function is shown in Figure 11.

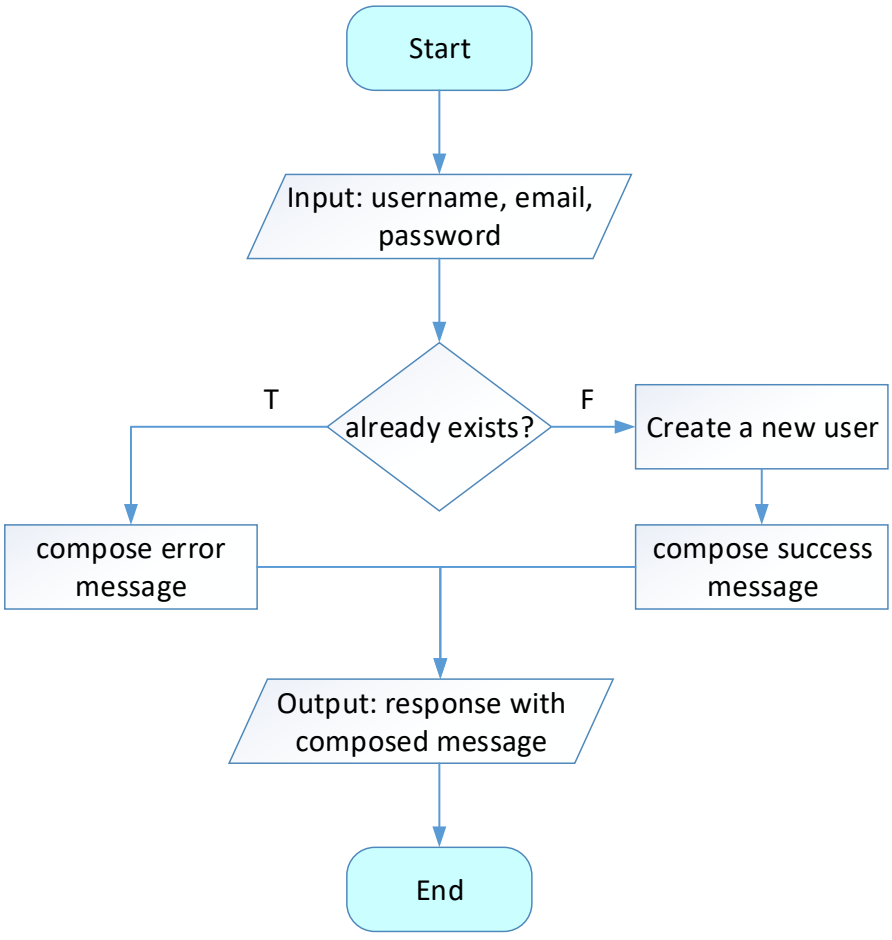

Figure 11. The workflow of the register view

When a user tries to register using an email, the register view function first asks the database if the email already exists. If so, an error message will be composed to tell the user why it fails the registration. If not, the input user information will be used to create a new user in the database, then a success message will be composed. The key part of the view function is shown in Table 21. Line 1-3 is used to get the user input from the request object, Line 4 means the User model first test if there exists such row in the table marketplace_user. If yes, that row will be selected and get returned as a User object, otherwise, the model will use the input info to insert one row in the marketplace_user table. Line 5-9 composes messages per the result of Line 4. Line 10 returns the response with the composed message.

```
1. username = request.POST.get("username")
2. email = request.POST.get("email", "")
3. password = request.POST.get("password")
4. obj, created = User.objects.get_or_create(username=username, email=email.lower(),
                                                password=password)
5. if created:
6.     message = "You successfully registered! Now login!"
7. else:
8.     message = "This email already exists. Please change it."
9.     status = "error"
10. return render(request, "marketplace/register.html", {"message" : message, "status": status})
```

Table 21. The code snippet of the register view

The register.html is shown in Figure 12. In the template, codes in Table 22 is used to show the returned message and visualize the message box per the status, error or success. One thing that needs special mention is the frontend framework makes extensive use of inspinia.js [21]. Inspinia.js is an integrated frontend responsive

framework. It's built based on HTML 5 [22] and Bootstrap [23]. And it integrates various useful javascript and css toolkits, such as Dropzone [24], ChartJS [25], Datapicker [26], and DataTables [27] etc.

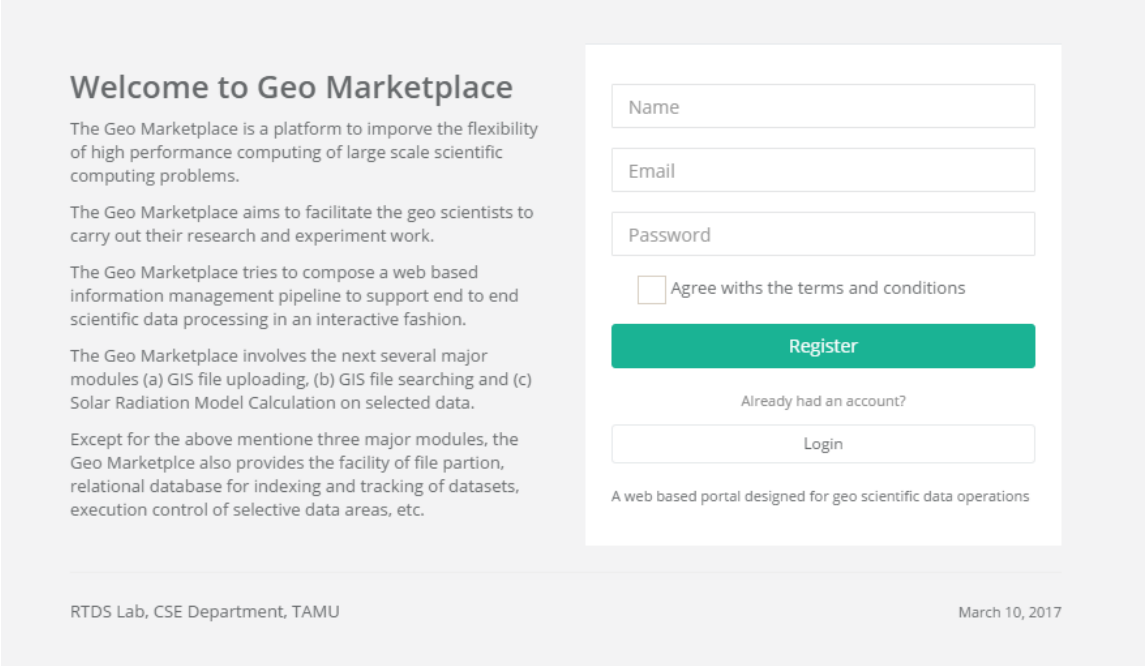

Figure 12. The register.html template

```
{%if message%}
      <script>
         swal({
               title:"",
               text: "{{message}}",
               type: "{{status}}",
               confirmButtonClass: 'btn btn-primary m-b buttonWidth',
               buttonsStyling: false
            });
      </script>
{%endif%}
```

Table 22. The message box in the register.html

3.3 Login and logout

The login view and its attached template login.html are used to let the user log in to the marketplace platform. The key code snippet of the login view function is shown in Table 23, and its main workflow is shown in Figure 13. As the figure shows, After the user's input gets sent to the login view function, the login view first test if the user exists in the database, if not, then directly send back an error message. If yes, then it gets the password from the database where the email is what the user's input. Next, we get the md5 digest of the user's password and compare it with the md5 value sent from the user's input. It they are the same, then the login view prepares its success template upload.html attached with a success message. Otherwise, an error message is sent back indicating login failure. One thing that needs mention is, after the password comparison succeeds, the user info is save into the session which records information for the whole user interaction with the website before he logs out.

```
email = request.POST.get("email", "")
password = request.POST.get("password", "")
try:
      user = User.objects.get(email=email.lower())
       password_instore = user.password
       encryption = hashlib.md5()
       encryption.update(password_instore)
       password_md5 = encryption.hexdigest()
       if password == password_md5:
            request.session["userId"] = user.id
            request.session["username"] = user.username
            request.session["balance"] = user.credit_point
            return redirect(reverse("marketplace:upload"),
                        {"success": "success", "username" : user.username, "balance": user.credit_point})
```

Table 23. The code snippet of the login view

```
        else:
            message = "Wrong email or password."
            status = "error"
    except User.DoesNotExist:
        message = "Wrong email or password."
        status = "error"
    return render(request, "marketplace/login.html", {"message":message, "status": status})
```

Table 23. Continued

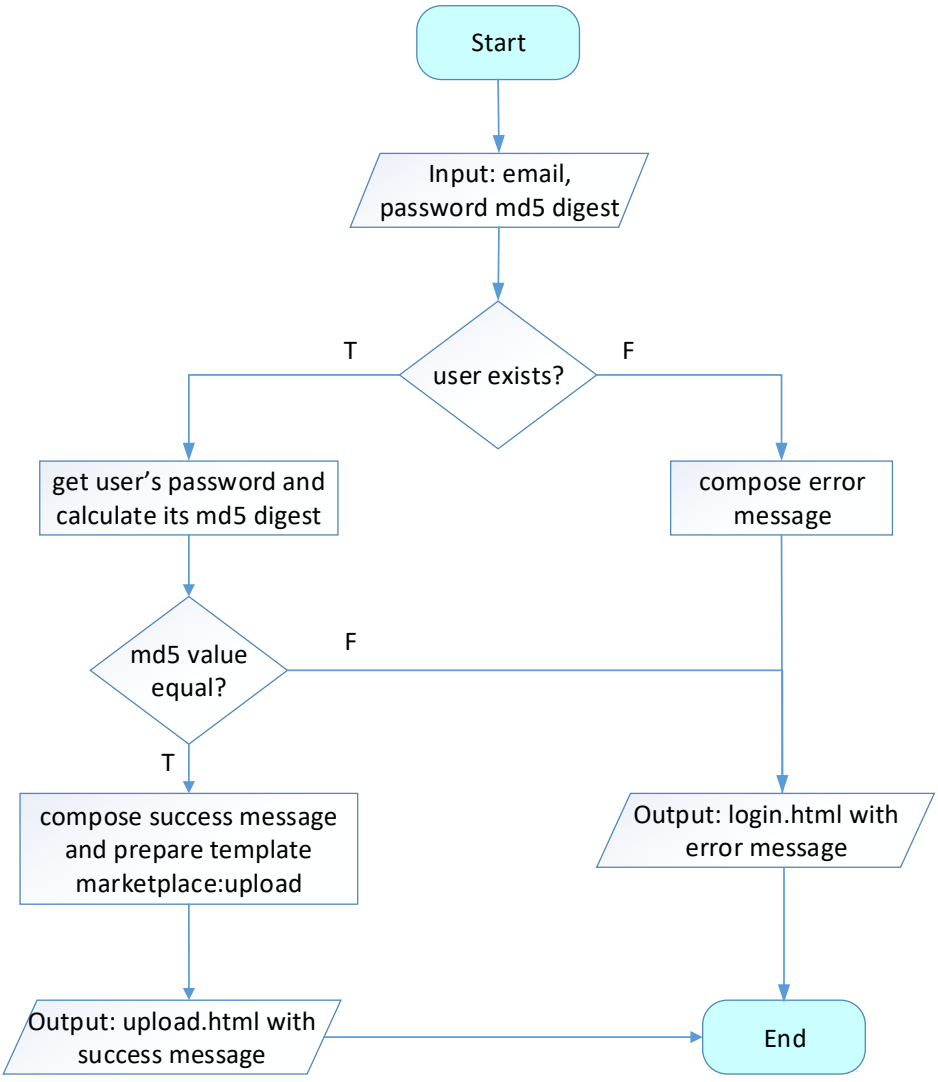

Figure 13. The workflow of the login view

The login.html template is shown in Figure 14. As the figure shows, the template also provides entrances for the user to retrieve his forgotten password and register for an account.

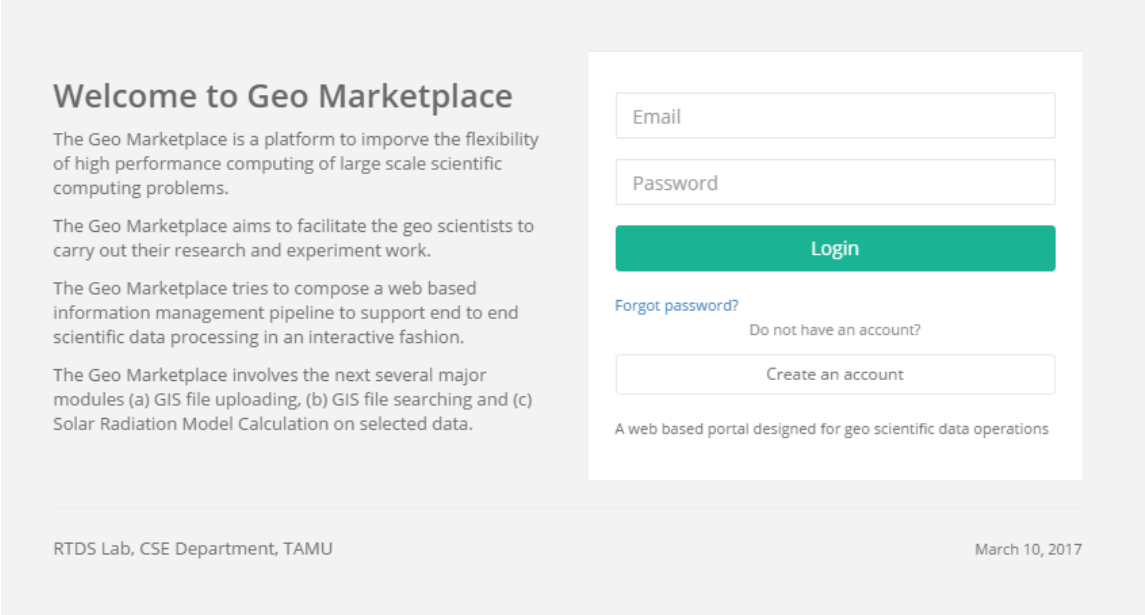

Figure 14. The login.html template

Before the user click the login button to submit the login form, the template use the javascript function mask() to calculate the md5 digest of the user's input in the password textbox, assign the md5 digest to a hidden password textbox. Then the value of the hidden password textbox, together with email, gets send the hidden password field as a request parameter to the backend server to validate. The key code snippet is shown in Table 24.

```
…...
<input type="password" class="form-control" id="password_input" placeholder="Password"
required="">
 <input type="hidden" id="password_md5" name="password">
…...
<script>
      function mask(){
         var password_input = $("#password_input");
         var password_md5 = $("#password_md5");
         password_md5.val(hex_md5(password_input.val()));
         return true;
      }
</script>
…...
```

Table 24. Md5 digest calculation of the password input by a user

As for the logout view, it first empty session using request.session.flush(), then redirect to login.html.

3.4 Retrieve

The retrieve view, along with its template retrieve.html is used for the user to get back his password if he forgets it. Its main workflow is shown in Figure 15. As the figure shows, after the backend retrieve view function receives the user request, it first asks the database if the email exists. If no, it will send back an error message indicating the email problem. Otherwise, it will get the password per the provided email and send it to that email. Fortunately, Django provides an easy email API for use to invoke. First, we need some configurations in settings.py, as is shown in Table 25. I temporarily chose to use the public email server to build this module. The EMAIL_HOST and EMAIL_PORT specifies the web server name and listening port. EMAIL_HOST_USER and EMAIL_HOST_PASSWORD is the email account I registered in that email server, which is used to send forgotten email to the user. EMAIL_USE_TLS tells Django the

specified email server needs TLS, that is Transport Layer Security, to protect email communication.

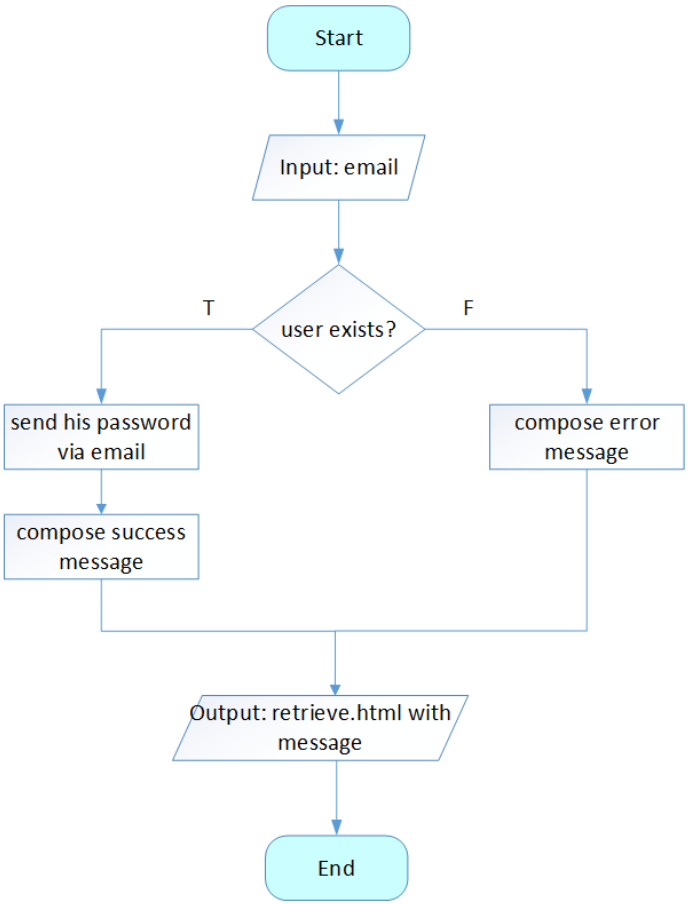

Figure 15. The workflow of the retrieve view

```
EMAIL_HOST = "smtp-mail.outlook.com"
EMAIL_PORT = 587
EMAIL_HOST_USER = "geomarketplace@outlook.com"
EMAIL_HOST_PASSWORD = "geomarketplace2017"
EMAIL_USE_TLS = True
```

Table 25. Configurations for email in settings.py

The retrieve.html template is shown in Figure 16.

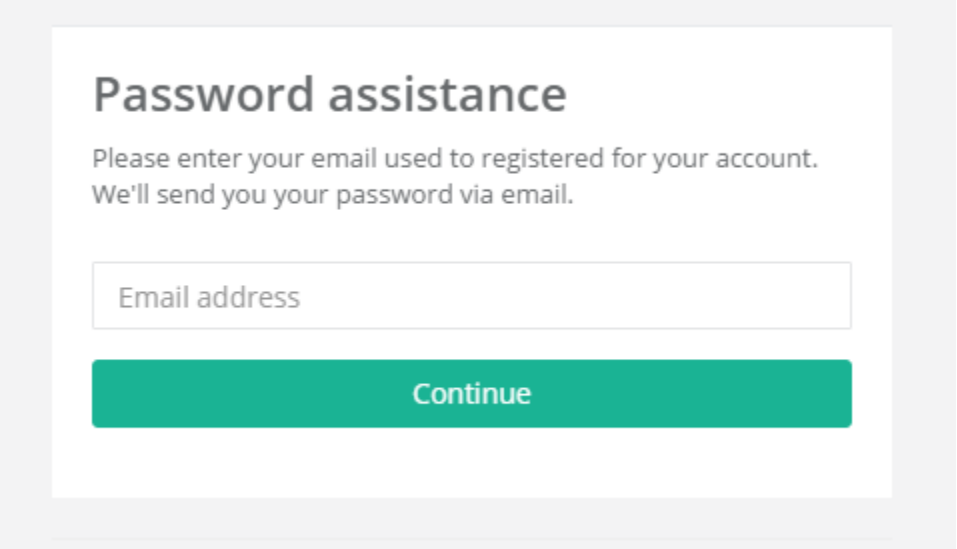

Figure 16. The retrieve.html template

3.5 Upload

The upload view and its upload.html template are used to help a user upload their gis data into the backend petabyte scale file system. The upload view function mainly involves four steps. First, write the uploaded file to the petabyte scale file system. Second, store the information attached with the upload file into database hosted on the local server. Next, if the user chose to generate tiles for the uploaded file, the third steps will be started. At last, after the tiles are all generated, the view function will store all the tile information to database hosted on the local server. The overall workflow of the upload view function is shown in Figure 17.

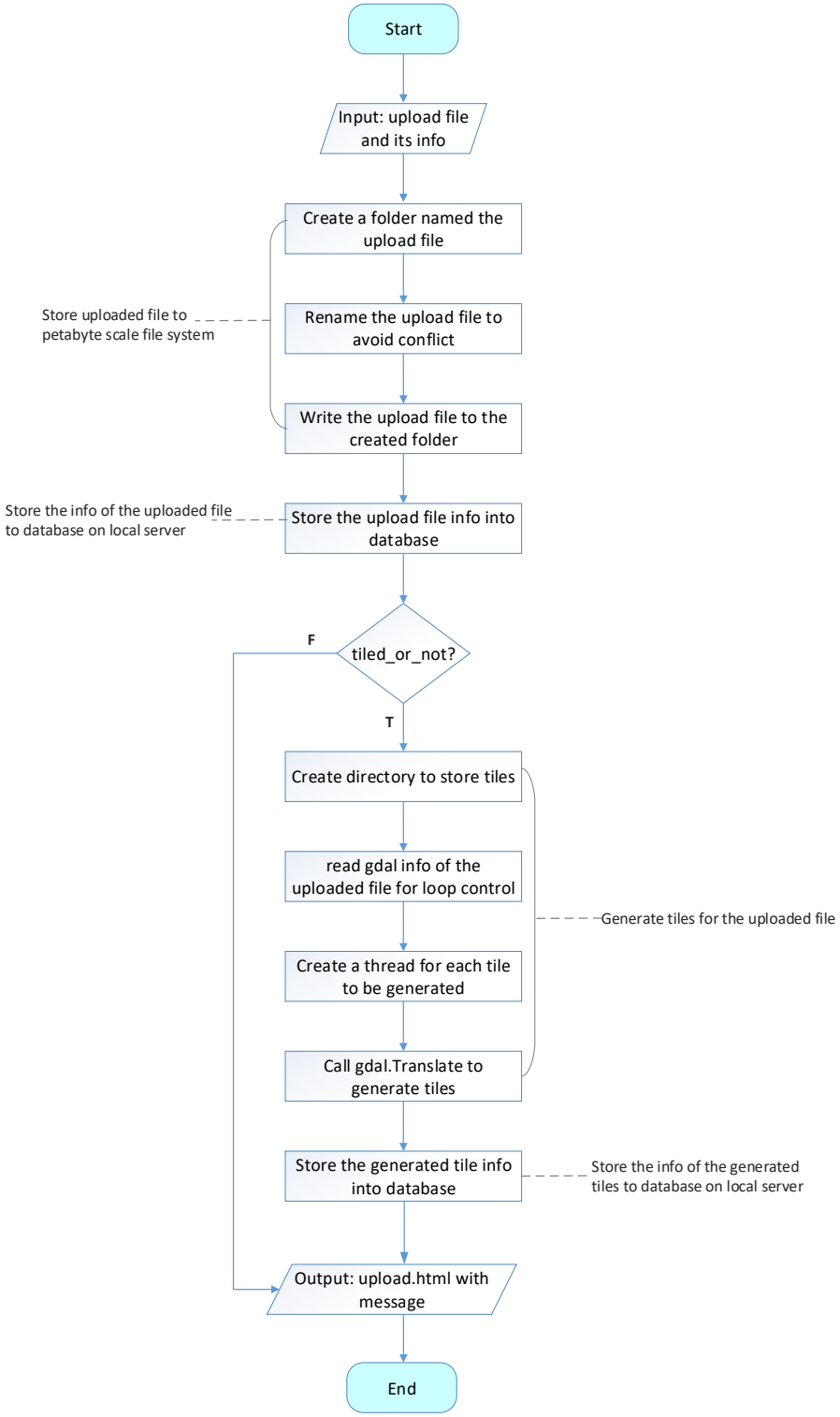

Figure 17. The workflow of the upload view

As Figure 17 shows, in the first step, when the view function receives the request, it first create a folder named after the uploaded file in the Uploads folder to store the uploaded file. The file name to be written to the petabyte file system is a combination of the filename and a timestamp to avoid the conflict that users may upload files of the same file names. Line 1-5 in the code snippet in Table 26 shows the key logic. The second step is Line 6-7 which uses the uploaded file information to create a GisFile object and save it into the database. Line 8-9 will launch the tileRaster function if the user chooses to generate tiles for the uploaded file.

```
……
1. file_name_new = ".".join([strftime("%H_%M_%S_%m_%d_%Y"), file_name.split(".")[1]])
2. file_location = os.path.join(file_path, file_name_new)
3. with open(file_location, 'wb+') as destination:
4.             for chunk in fileData.chunks():
5.                 destination.write(chunk)
……
6. uploadFile = GisFile(file_name = file_name, file_location = file_location,
                        data_time = data_time, resolution = resolution, source = source,
                        public_or_not = public_or_not, tiled_or_not = tiled_or_not,
                        price = price, tile_size = tile_size, uploader = uploader, upload_time = upload_time,
description = description, user_id = user_id)
7. uploadFile.save()
8. if tiled_or_not == 1:
9.         tileRaster(uploadFile)
……
```

Table 26. The code snippet of the upload view

Line 1-5 in the tileRaster() function, shown in Table 27, iterates the uploaded file with an iteration step valued at tileSize specified by the user when he uploads the file. As Line 3 shows, each tile generation job is wrapped with a Python thread for multithreading. Line 5 starts the created thread which in turn calls Line 9 in the

generateTiles() function. Line 8 is responsible to store the bulk of tile infos into the database using the batch execution mode.

```
......
for rowPixelOffset in range(0, width, tileSize):
        for colPixelOffset in range(0, height, tileSize):
                ......
                oneThread = threading.Thread(target=generateTiles,
                                                args=(uploadFile, outputTileName,outputTilePath,
                                                inputData, rowPixelOffset, colPixelOffset, tileSize, tiles))
                threads.append(oneThread)
                oneThread.start()
......
for thread in threads:
        thread.join()
Tile.objects.bulk_create(tiles, batch_size=500)
......
gdal.Translate(outputTilePath, inputData, srcWin = [rowPixelOffset,colPixelOffset,tileSize,tileSize])
......
```

Table 27. The code snippet of the tileRaster() and generateTiles() functions

The related upload template is shown in Figure 18. This template is assembled by two html files, base.html and upload.html. Given that the web pages of the marketplace application share common header and left navigation bar, I extracted the common html elements and put them into base.html. Base.html leave blank the cssPart, divPart, and jsPart blocks for the user to fill in with the help of Django's template language, as is shown in Table 28.

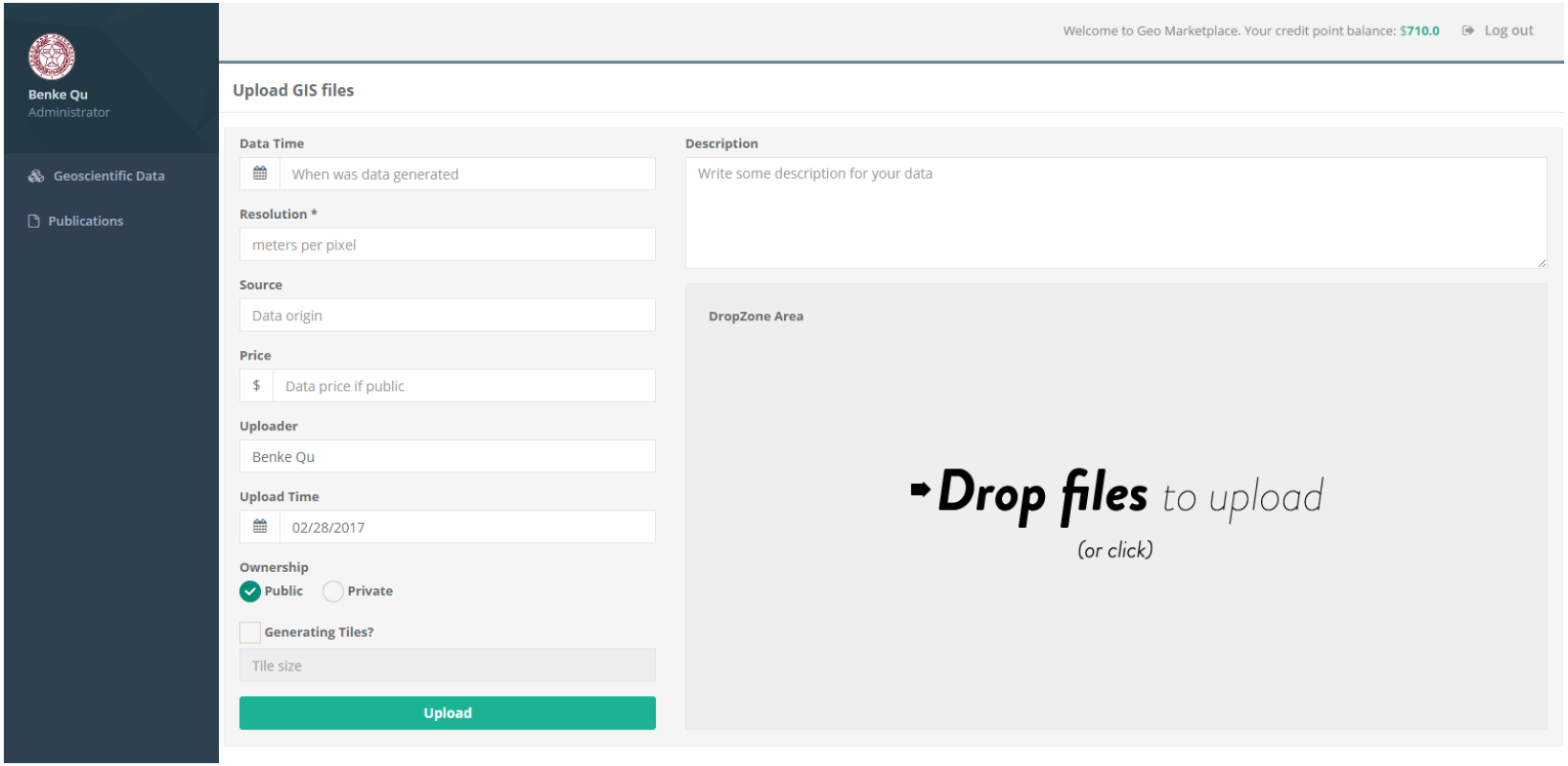

Figure 18. The upload.html template

```
<!DOCTYPE html>
<html>
    <head>
        ……
        {%block cssPart%}{%endblock%}
    </head>
    <body>
        <div>
            <nav>.....</nav>
            <div>{%block divPart%}{%endblock%}</div>
        </div>
    </body>
    {%block jsPart%}{%endblock%}
</html>
```

Table 28. The base.html webpage

As Figure 18 shows, the dropzone takes on the responsibility to deal with file uploading work. The key javascript related to the dropzone is shown in Table 29.

```
1. $(document).ready(function() {
        .....
2.      new Dropzone("#dropzoneDiv", {
3.          url: "{%url 'marketplace:upload'%}",
            .....
4.          init: function() {
                .....
5.              uploadButton.addEventListener("click", function(e) {
                        .....
6.                      myDropzone.processQueue();
7.              });
                .....
8.              this.on("sending", function(data, xhr, formData) {
9.                      formData.append("data_time", $("form input[name=data_time]").val());
                        .....
10.                     formData.append("description", $("textarea#description").val());
11.             });
```

Table 29. The frontend javascript function dealing with file uploading

```
12.                 this.on("success", function(file, response){
13.                   swal({......
14.                         text: "File got uploaded successfully!",
15.                         type: "success",
                            ......
16.                         })
                            ......
17.                 });
18.                 this.on("error", function(file, response){
19.                   swal({......
20.                         text: response,
21.                         type: "warning",
                            ......
22.                         });
23.                 });
24.             }
25.     });
        ......
26.});
```

Table 29. Continued

Line 3 specifies which view function the uploaded file is going to send to. Line 5-7 processes the uploaded file and does the actual html form sending work. Line 8-11 attaches other user inputs along with the sending process. Line 12-17 will alert the user that the file was successfully uploaded, otherwise, Line 18-22 alert the user something went wrong.

3.6 Search

The search view and its template search.html is used to help the user the browse the published gisFiles and do possible operations on them. One important thing is, we must filter out those files that are private and not belong to the login user and only leaves those that are owned by the login user, public or private and public files that are exposed by the other users. Moreover, when the user browses the data, he needs to know if he can process them. All data uploaded by the user himself, public or private, can be processed.

Data published by others and purchased by the user can also be processed. One thing is, the data that can be process must be of type geotiff right now. The system will expand later based on requirements. The main workflow of the search view function is shown in Figure 19.

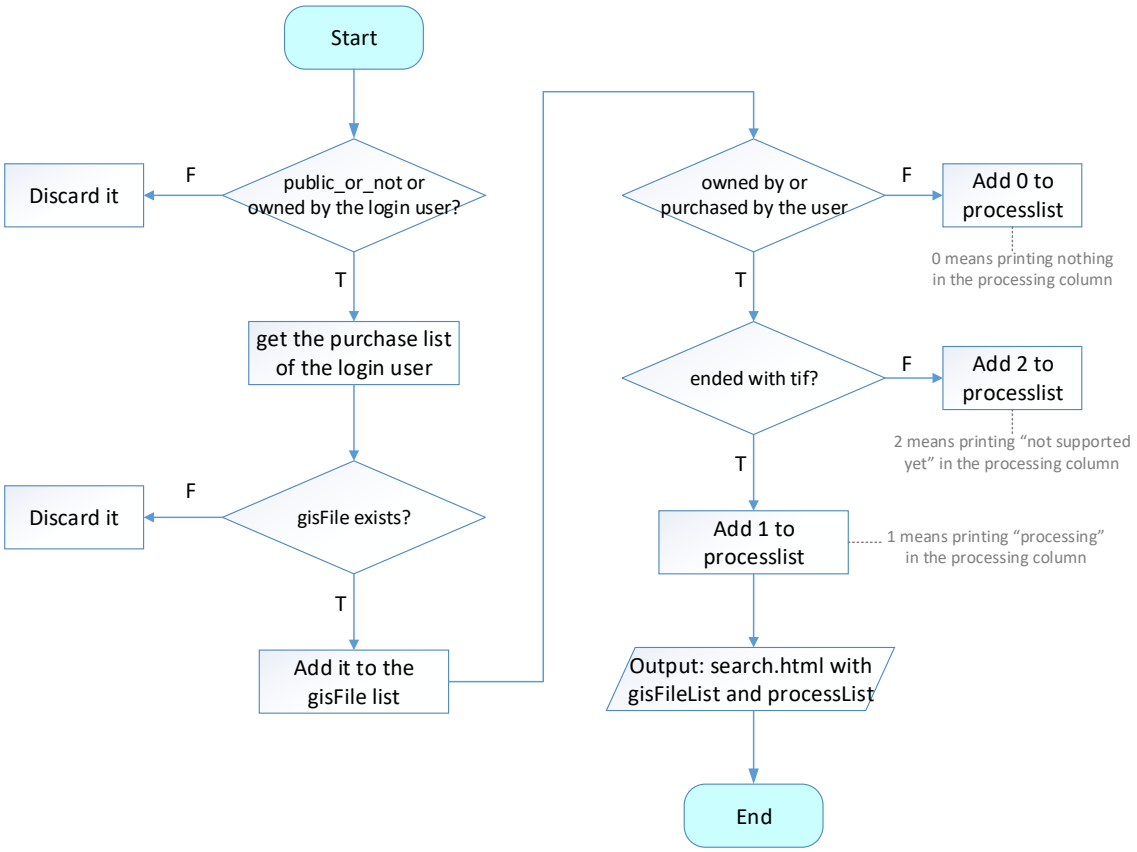

Figure 19. The workflow of the search view

One thing that needs mention is in search.html, there is a column called processing. One row that is marked 0 in the processlist will show nothing, marked 2 will

show “not supported yet” and marked 1 will show a “processing” like for further processing. The key part of the search view function is shown in Table 30.

```
1. gisFileSet = GisFile.objects.filter(Q(public_or_not=1) | Q(user_id=user_id))
2. purchaseList = Transaction.objects.filter(user_id = user_id)
3. gisFileList = []
4. actionList = []
5. for gisFile in gisFileSet:
6.         if os.path.exists(gisFile.file_location):
7.                 gisFileList.append(gisFile)
8.                 if (user_id==gisFile.user_id or purchaseList.filter(file_id=gisFile.id).exists()):
9.                         if gisFile.file_name.lower().endswith("tif"):
10.                                 actionList.append(1)
11.                         else:
12.                                 actionList.append(2)
13.                 else:
14.                         actionList.append(0)
```

Table 30. The code snippet of the search view

The template search.html is shown in Figure 20. With the help of the front end js named DataTables.js, we could elegantly show the result in pages, build up the powerful search box and store the list into clipboard or as a csv, excel or pdf file or simply print them out. See the processing column for different types of list results.

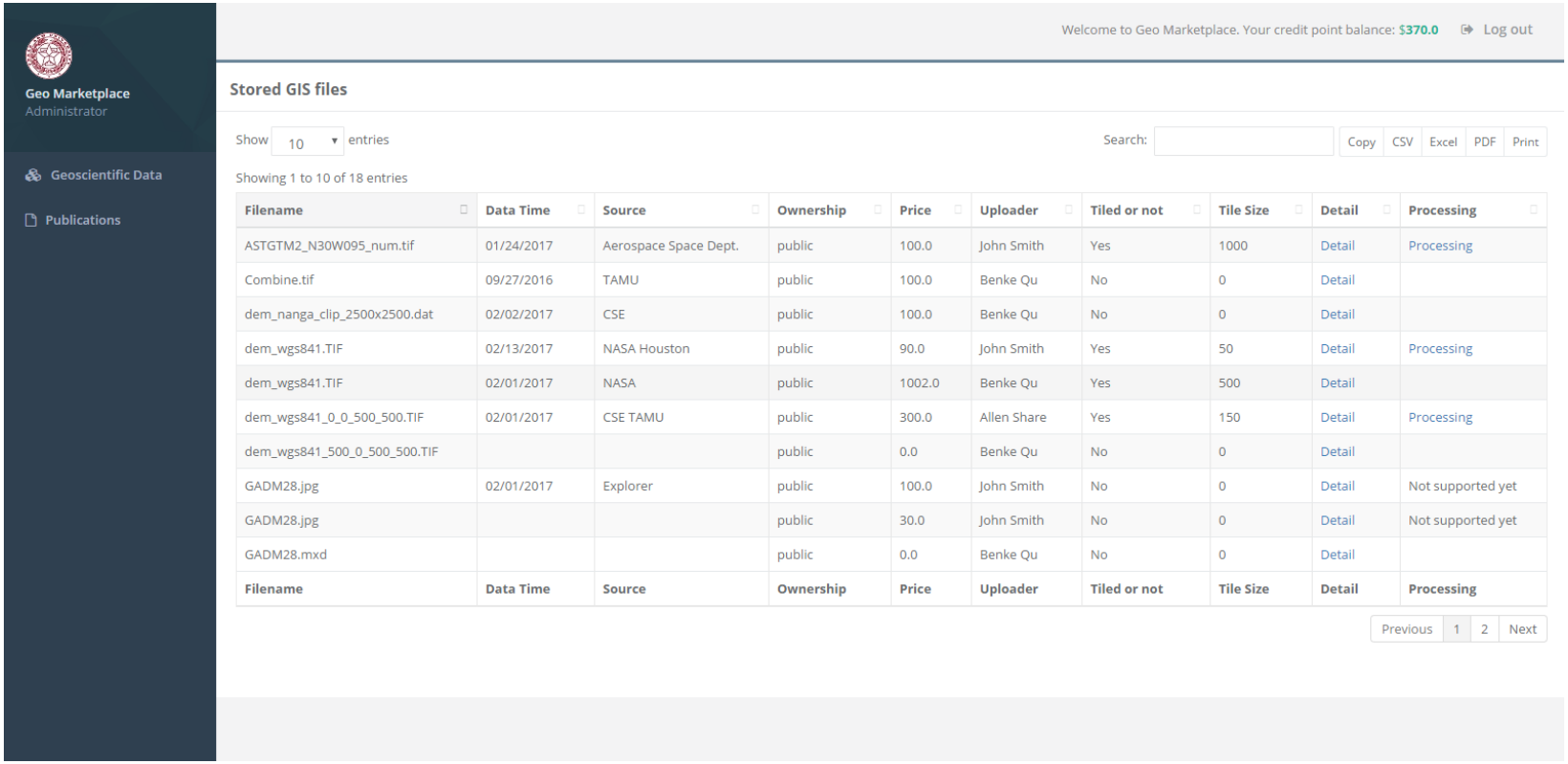

Figure 20. The search.html template

3.7 Detail and save

From Figure 20, we can see there is one column called Detail for each row of result. This is used to check the details of listed gisFile. This function is fulfilled by the detail view function and its template called detail.html. The simple workflow of the detail view is shown in Figure 21. As the figure shows, having the gisFileId in hand, the detail function needs to ask the database if the gisFile that needs checking details is owned by the login user. If yes, the detail.html template will show the data editable, otherwise, if the data was not purchased the login user previously, the detail.html template will show the data purchasable, if it was already purchased by the user, the user can do nothing more to the data. He can only check the details of the data. The key part of the detail view function is shown in Table 31. It implements the logic expressed in the workflow.

The detail.html template is shown in Figure 22. If the data is purchasable for the login user, a green “Buy this data” button will show up in the bottom right corner and the data details cannot be edited by the user. If the data is owned by the user, then he can edit the data details and save the update. In this case, the bottom right corner first shows a “Edit this data” button. as is shown in Figure 23. If the user clicks the button, it will change to “Save this data”, all the columns in this page will change to an editable mode, as is shown in Figure 24. One thing that needs attention is the “Go back to data list” button in bottom left corner. Its function is implemented in the javascript function called gotoSearch(), which helps go back the previous search page. For this function, we cannot simply write “window.history.back();” to use the browser cache of the previous

page. In this way, the update of the data cannot be brought to the already cached old data. One way is to simply ask for the backend server to send back the search.html again using “$(location).prop("href", "{%url 'marketplace:search'%}");”.

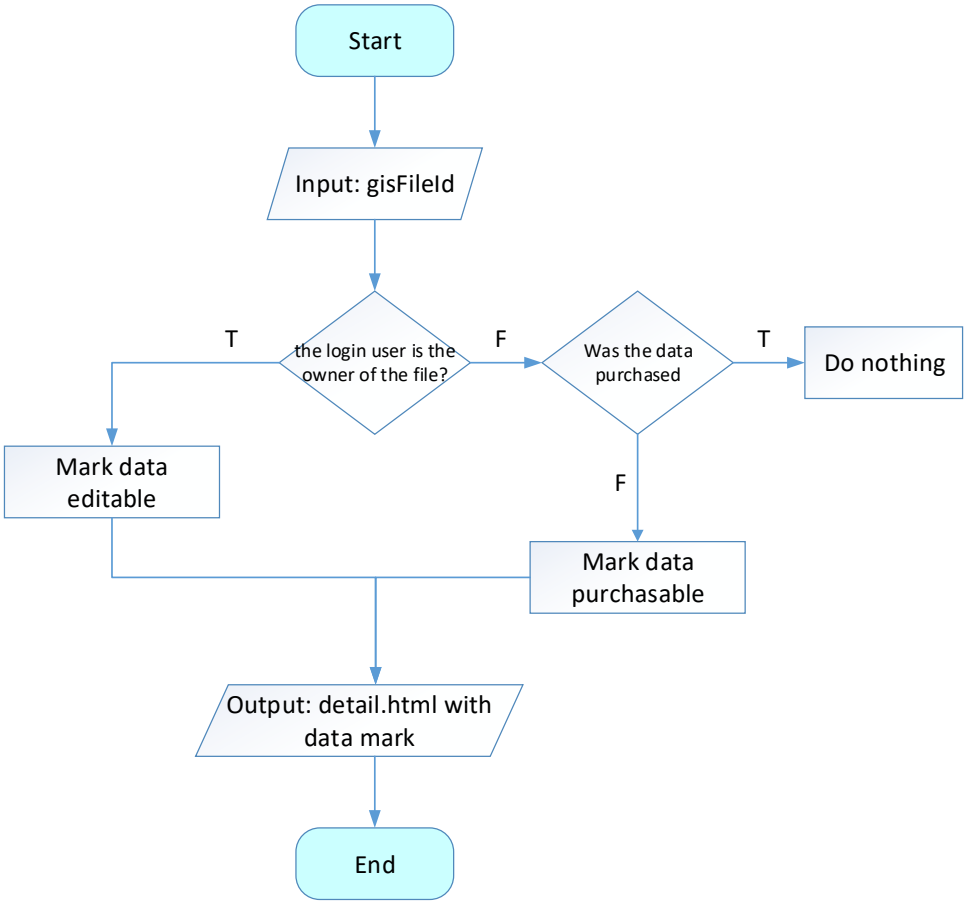

Figure 21. The workflow of the detail view

```
gisFile = GisFile.objects.get(pk=gisFileId)
    purchased = False
    if user_id != gisFile.user_id:
        if Transaction.objects.filter(user_id=user_id, file_id=gisFile.id).exists():
            purchased = True
```

Table 31. The code snippet of the detail view

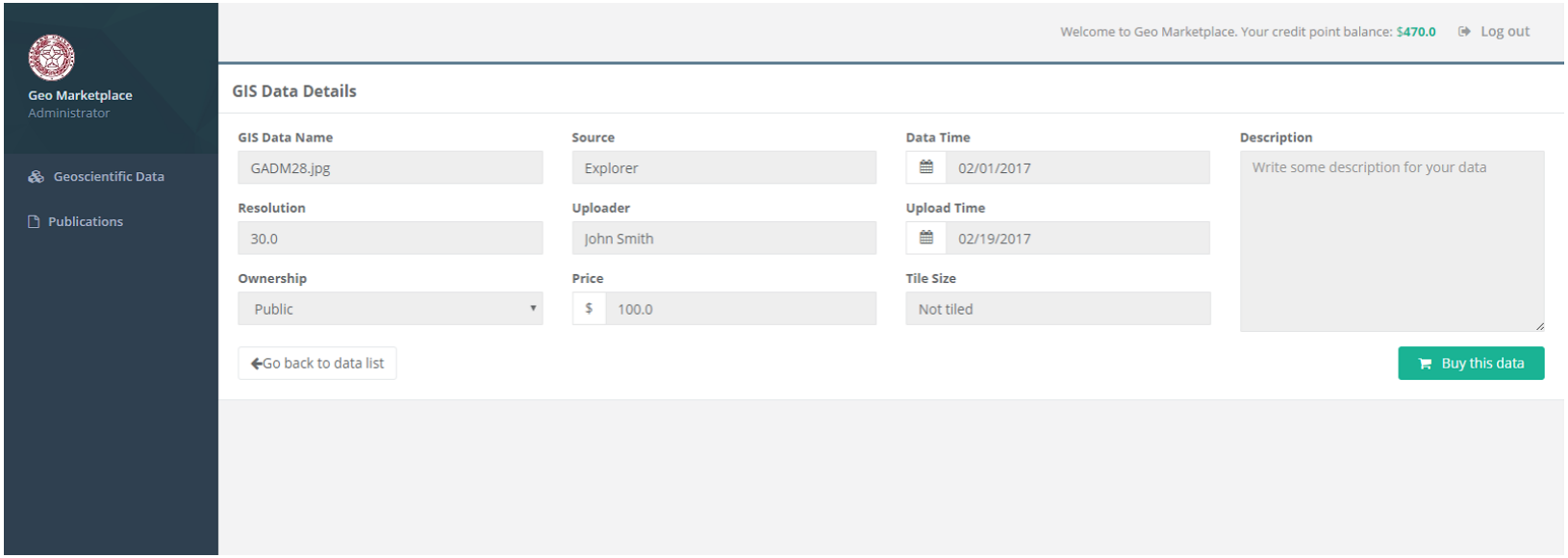

Figure 22. The detail.html template with the “Buy this data” button

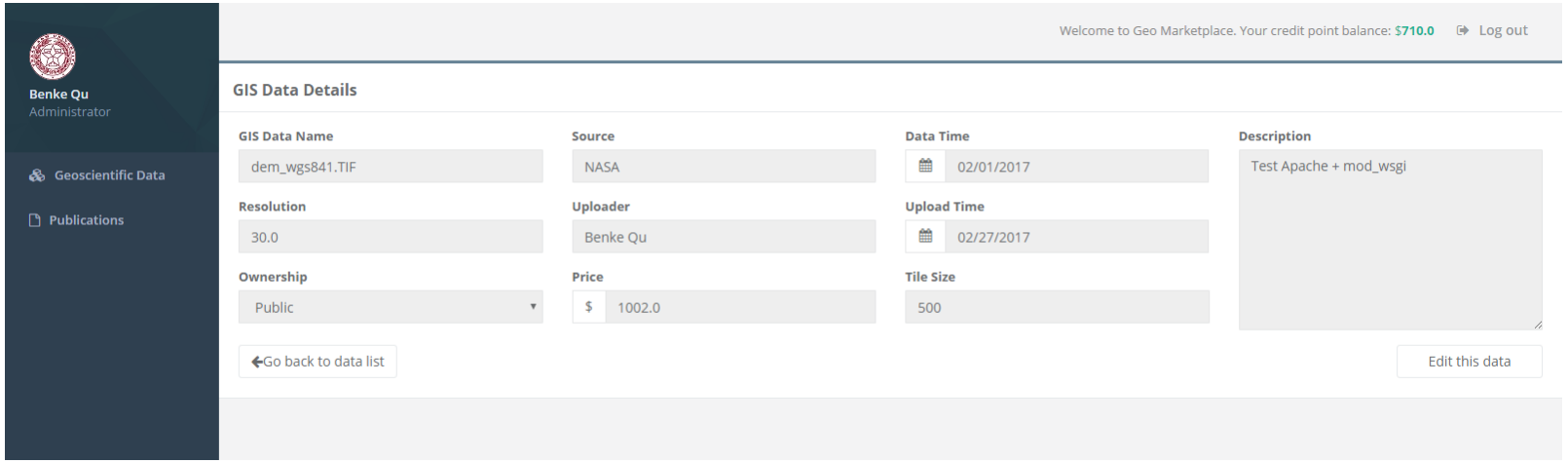

Figure 23. The detail.html template with the “Edit this data” button

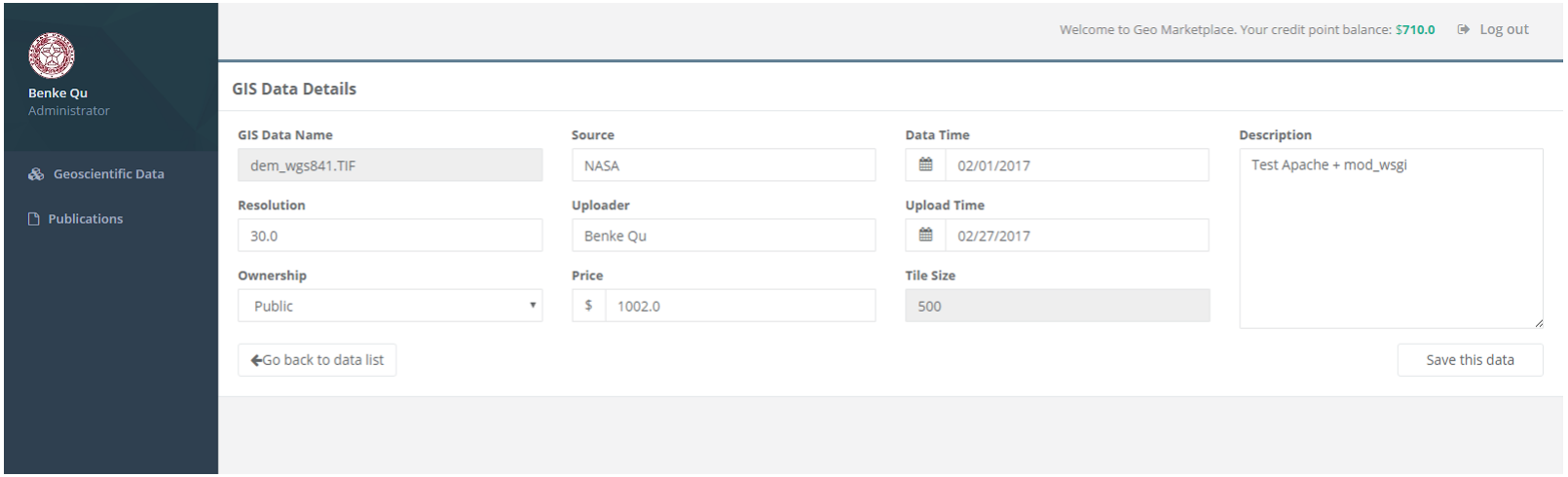

Figure 24. The detail.html template with the “Save this data” button

When we click the “Save this data” button in Figure 24, the save request is sent via Ajax to the backend save view function to process. It saves the updates into the database.

3.8 Order and payment

When the user clicks the “Buy this data” button in Figure 22, he will be redirected to the order.html page as is shown in Figure 25. The marketplace provides the user three options to purchase data. One is the credit point maintained by the marketplace platform, one is paypal and the other is to use credit card for payment. Right now, the paypal and credit card options are yet to be inflated with real payment processes. The payment method credit point was implemented for the marketplace. The credit point option is shown in Figure 25, and the credit card option is shown in Figure 26.

The order.html page let the user check the details of data again before he decides to buy it. When he clicks the “Buy with Credit Point” button, the transaction request is sent to the backend payment view function for processing. The workflow of the payment view function is shown in Figure 27.

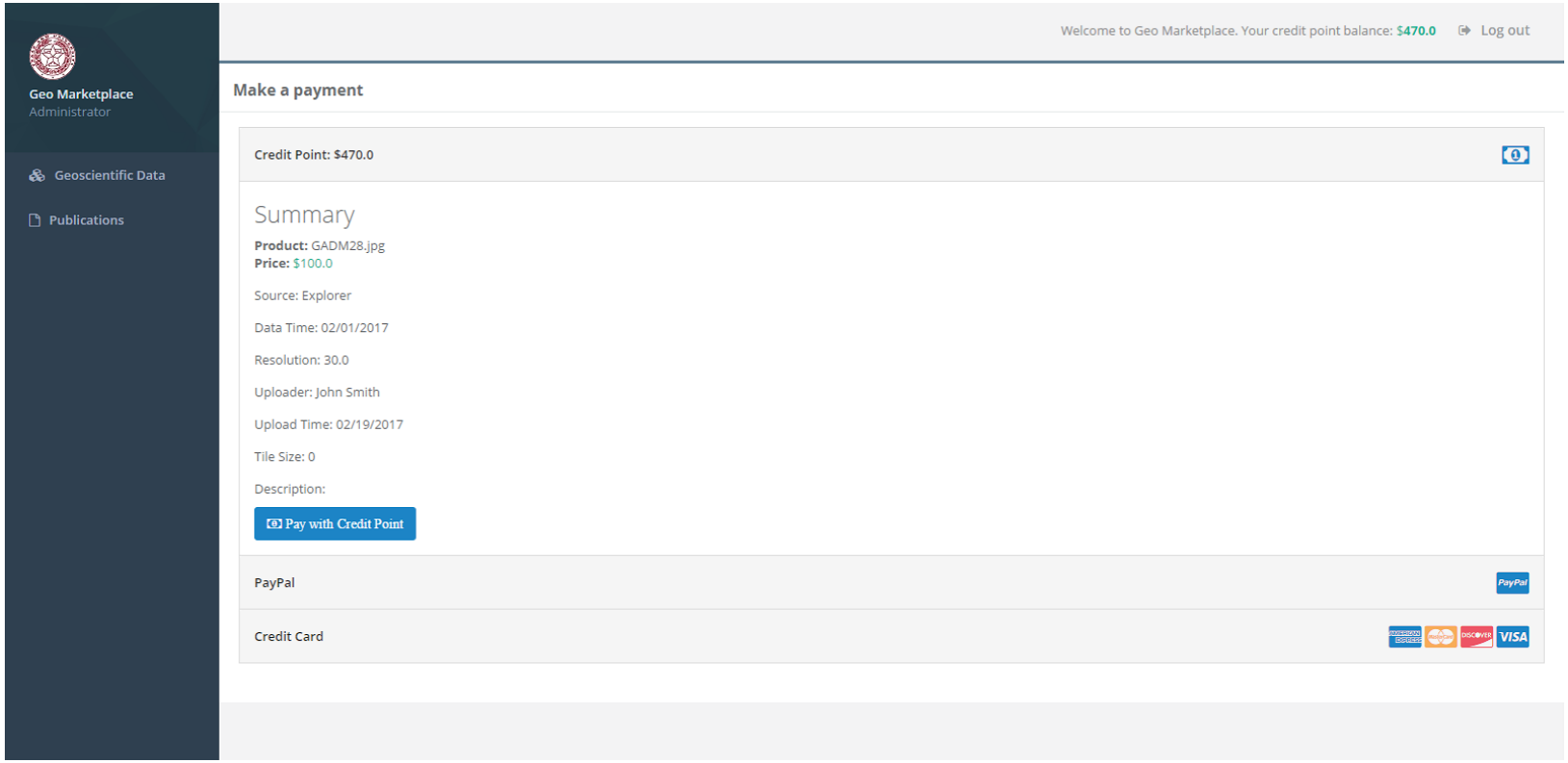

Figure 25. The order.html template with the “credit point” option expanded

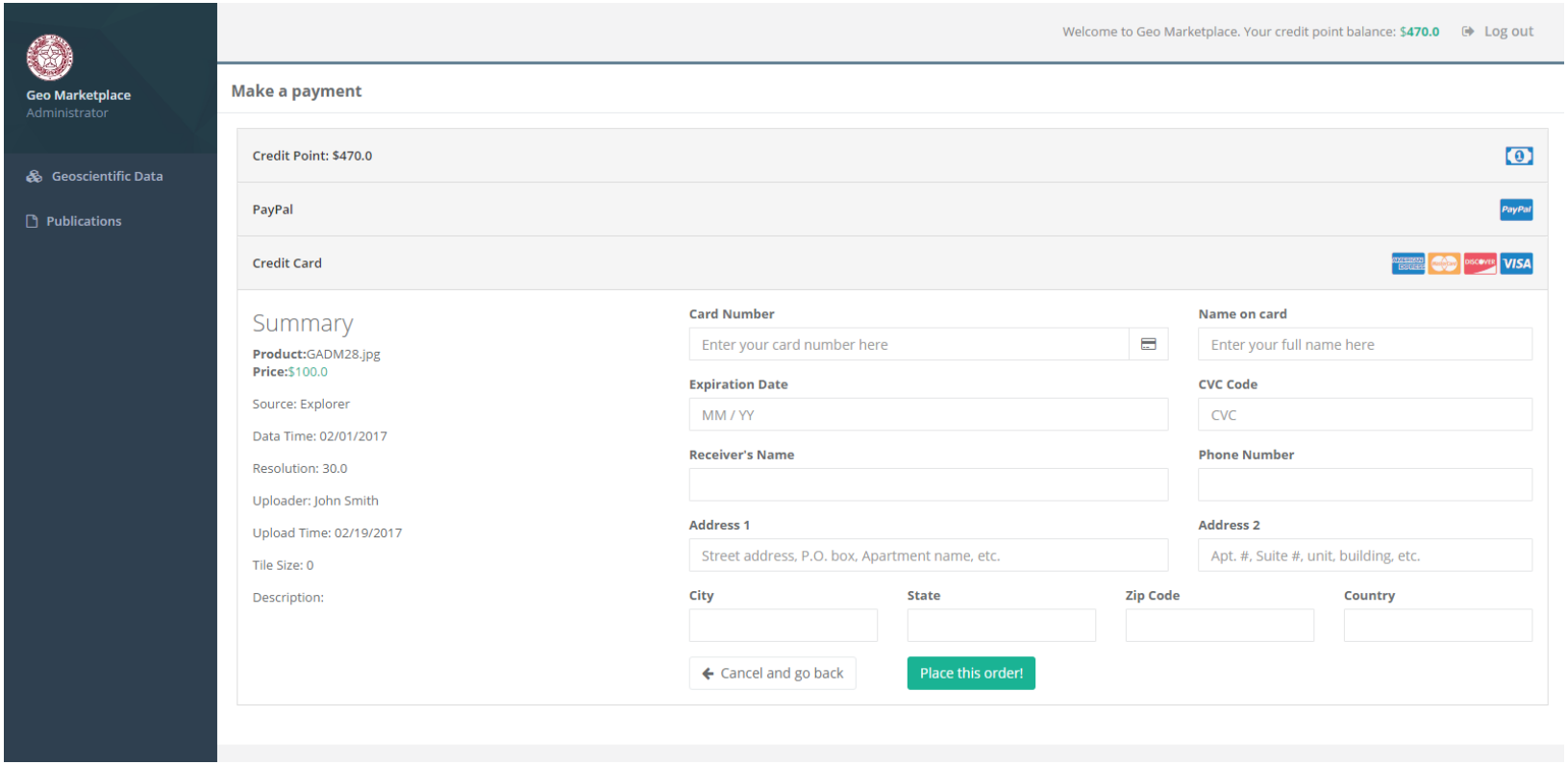

Figure 26. The order.html template with the “credit card” option expanded

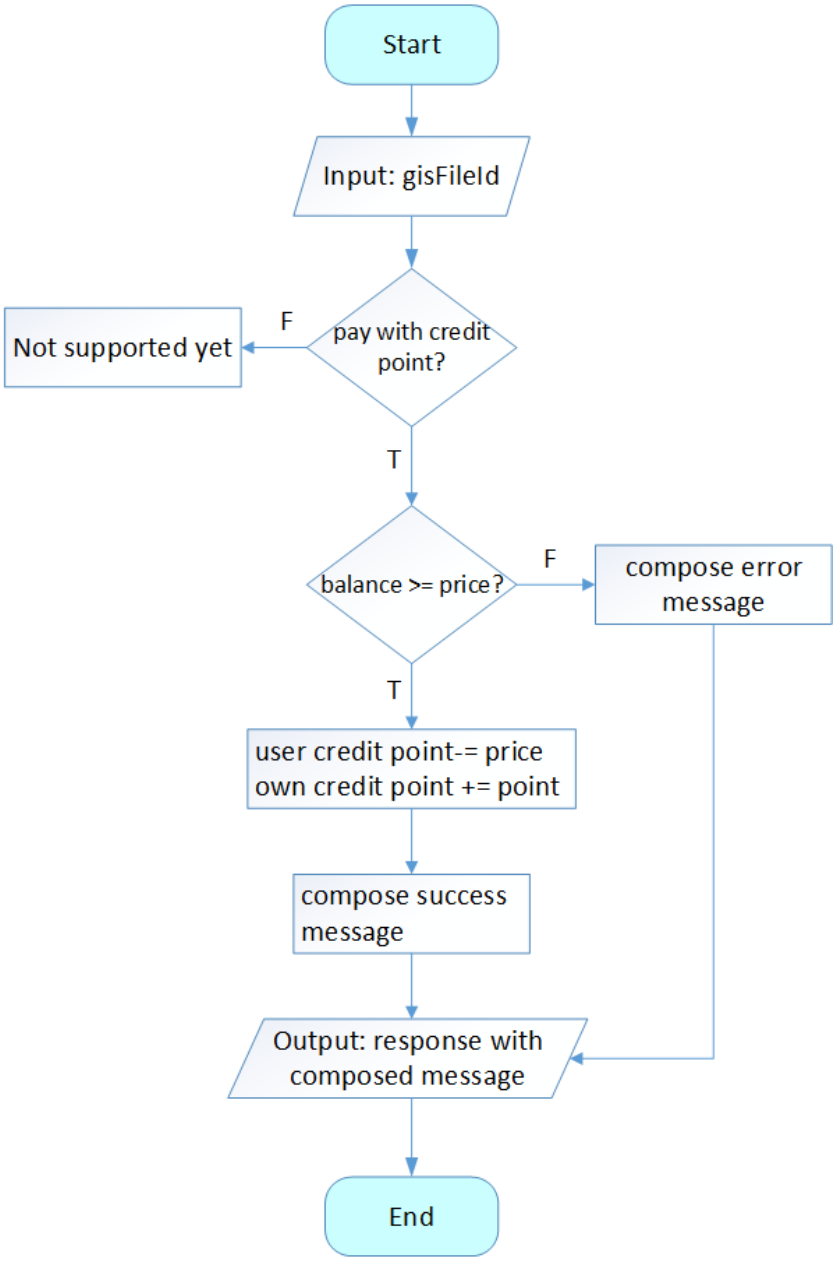

Figure 27. The workflow of the payment view

The key logic of the payment view function is shown in Table 32. The transaction should an atomic transaction, meaning that, the action of buyer's credit point getting deducted and that of the seller's credit point getting increased should both or neither happen. To use Line 1, we need to import transaction and IntegrityError from django.db. It's a Python decorator that guarantees the transaction atomicity. Line 7-18 updates the credit point balances of the buyer's and the seller's and save the transaction.

```
1. @transaction.atomic
2. def payment(request):
   ......
3.    if paymentMethod == "creditPoint":
4.       origin = User.objects.get(pk=gisFile.user_id)
5.       user = User.objects.get(pk=user_id)
6.       if user.credit_point >= price:
7.          try:
8.             with transaction.atomic():
9.                user.credit_point -= price
10.                origin.credit_point += price
11.               user.save()
12.                origin.save()
13.                oneTransaction = Transaction(user_id = user_id, file_id = file_id, amount=price)
14.                oneTransaction.save()
15.                request.session["balance"] = user.credit_point
16.             message = "Success@Go to the search page to find your purchased data!"
17.          except IntegrityError:
18.             message = "Error@Transaction failed due to internal error."
19.       else:
20.          message = "Warning@Your credit point balance is not enough to buy this data! Change"
                        + "another way to buy it."
   ......
```

Table 32. The code snippet of the payment view

3.9 Process, calculation and result

The process view function is responsible to prepare the process.html template, call the remote Google Maps Javascript API to render the supporting google map. The

google map provides a rich API to interact with the underlying map. When loading the process.html page, the initMap() function makes use of the response tile_info to set the center of the map and to render the covering area of the data. The code snippet in Table 33 shows the key logic.

```
function initMap() {
        map = new google.maps.Map(document.getElementById('map'), {
                                        zoom: 9,
                                        center: {
                                                lat: {{tile_info.cornerCoordinates.center.1}},
                                                lng: {{tile_info.cornerCoordinates.center.0}}
                                        },

                                        mapTypeId: "terrain"
                });
        var areaBounds = {
                north: {{tile_info.cornerCoordinates.upperLeft.1}},
                south: {{tile_info.cornerCoordinates.lowerRight.1}},
                east: {{tile_info.cornerCoordinates.lowerRight.0}},
                west: {{tile_info.cornerCoordinates.upperLeft.0}}
        };
        geoArea = new google.maps.Rectangle({
                bounds: areaBounds,
                strokeColor: '#FF0000',
                strokeOpacity: 0.9,
                strokeWeight: 1,
                fillColor: '#FF0000',
                fillOpacity: 0.1
        });
        geoArea.setMap(map);
        ......
}
```

Table 33. The code snippet of the initMap() function that draws the bounding box of the data

Then I bind a click listener to the created geoArea bounding box to monitor any click behavior of the mouse. When the mouse clicked twice on the geoArea bounding box, initMap() function will update the corresponding textboxes for the corner coordinates and create a new selectArea indicating the user's calculation area of interest.

I also bind three listeners to the new selectArea monitoring the “drag”, “bounds_changed” and “click” events on the selectArea, which updates the textboxes for the corner coordinates when these events happen on the selectArea. One thing is, for the click event on the selectArea, it removes the created the created selectArea bounding box and reset the corner coordinates to those of the geoArea bounding box. The code snippet in Table 34 shows the logic.

```
.....
google.maps.event.addListener(geoArea, 'click', function(e) {
                if(markerList.length < 2){
                        markerList.push(e.latLng);
                        if(markerList.length === 2){
                        …...
                        $("#upper_left_x").val(left_x);  $("#upper_left_y").val(upper_y);
                        $("#lower_right_x").val(right_x);  $("#lower_right_y").val(lower_y);
                        var areaBound = { north: upper_y, south: lower_y, east: right_x, west: left_x};
                        selectArea = new google.maps.Rectangle({……});
                        selectArea.setMap(map);
                        google.maps.event.addListener(selectArea, "drag", function(e){
                                var ne = selectArea.getBounds().getNorthEast();
                                var sw = selectArea.getBounds().getSouthWest();
                                $("#upper_left_x").val(sw.lng());  $("#upper_left_y").val(ne.lat());
                                $("#lower_right_x").val(ne.lng());  $("#lower_right_y").val(sw.lat());
                        });
                        google.maps.event.addListener(selectArea, "bounds_changed", function(e){
                                //the same to the “drag” event
                        });
                        google.maps.event.addListener(selectArea, "click", function(e){
                                selectArea.setMap(null);
                                markerList = []
                                var ne = geoArea.getBounds().getNorthEast();
                                var sw = geoArea.getBounds().getSouthWest();
                                $("#upper_left_x").val(sw.lng());  $("#upper_left_y").val(ne.lat());
                                $("#lower_right_x").val(ne.lng());  $("#lower_right_y").val(sw.lat());
                        });
……
```

Table 34. The code snippet of the initMap() function that prepares the selectArea

The process.html template is shown in Figure 28.

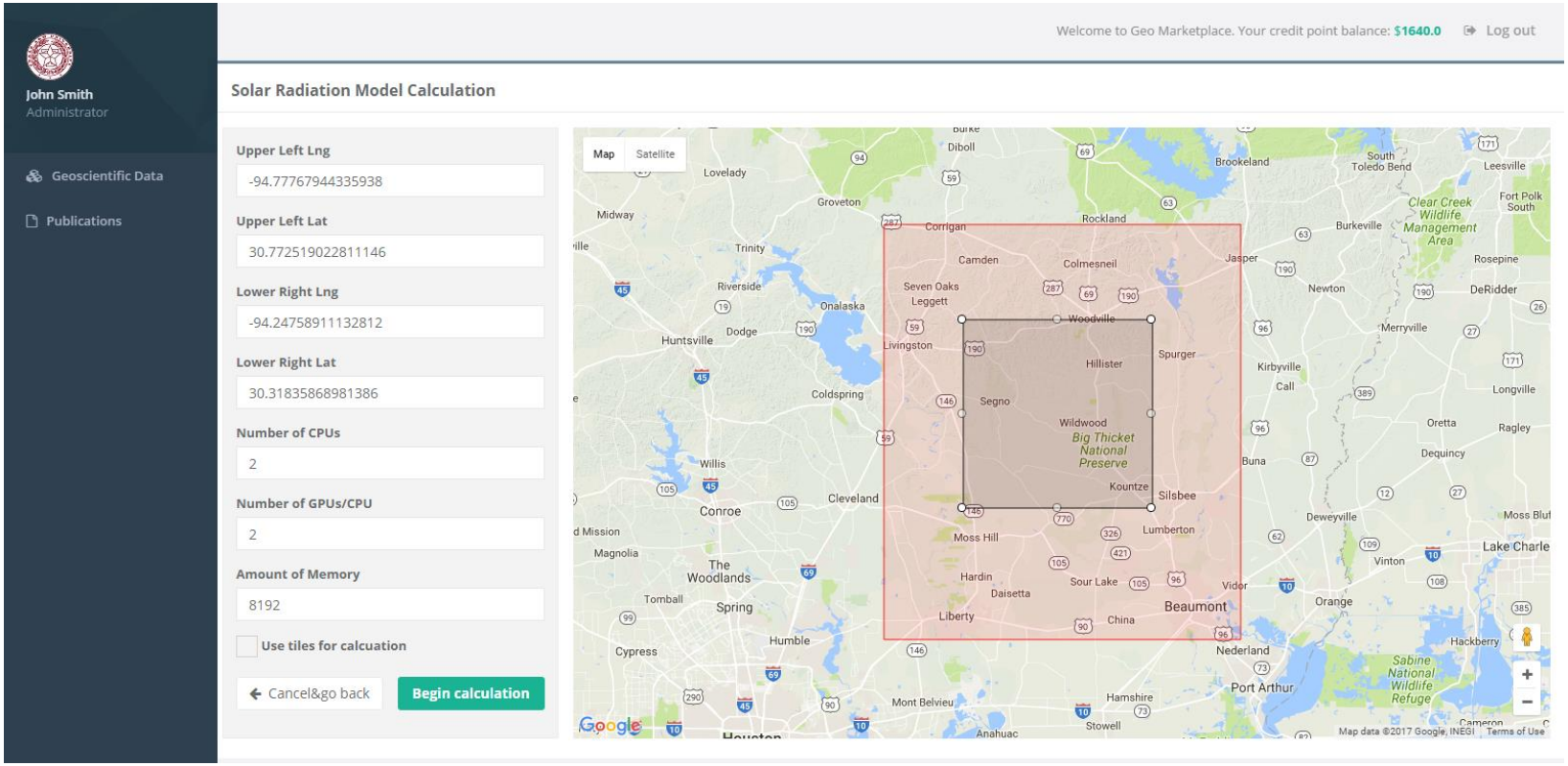

Figure 28. The process.html template

For data owned or purchased by the login user, he can do calculations on the data. As we can see in Figure 28, above the “Begin calculation” button, there is a checkbox

. If the select data for calculation was tiled when it was uploaded, the user can select if he would like to use tiles for calculation this time. If the select data is not tiled, this option is disabled, and the user can only clip the backend whole data to get the data covered by the selectArea and put the clipped data into calculation.

The following Figure 29 shows the possible relative positions of a tile (marked as T) and a selectArea query window (marked as Q).

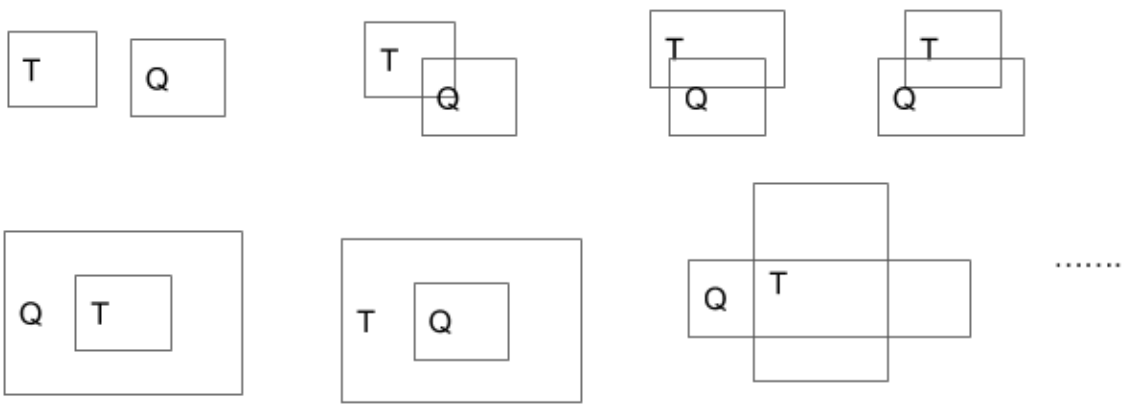

Figure 29. The possible relative positions of a tile and a selectArea query window

As the figure shows, we must take into considerations all the possible cases to test if they intersect. Here, I come up with a much simpler way to do the verification. Assume the upper left and lower right corners of a tile are T_1 (T_1.x, T_1.y) and T_2 (T_2.x, T_2.y), respectively, and the upper left and lower right corners of a query window are Q_1 (Q_1.x, Q_1.y) and Q_2 (Q_2.x, Q_2.y), respectively. If a tile and a query window don’t

intersect, it must be completely on the left/right of the query window or above/below the query window, so

if (!(T_1.x>=Q_2.x || T_2.x <= Q_1.x || T_1.y <= Q_2.y || T_2.y >= Q_1.y)){

collect T; //T and Q intersect

}

And its corresponding implementation in Django is *tiles = Tile.objects.filter (gis_file_id = file_id, upper_left_x__lt = lower_right_x, lower_right_x__gt = upper_left_x, upper_left_y__gt = lower_right_y, lower_right_y__lt = upper_left_y).* Right now, we only support Solar Radiation Model [28] calculation on geotiff data. After the user decides the selectArea for calculation and specifies the number of cpus, gpus/cpus and the amount of memory that are being used to create a calculation job on Terra, by clicking the "Begin calculation" button, these parameters get sent to the backend calculate view function. The workflow of the calculate view function is shown in Figure 30. Upon receiving the "job creation finished" message, the process.html template will constantly send ajax request to the backend result view function for the status of the calculation of the created jobs. The result view function constantly monitor the bin folder on the petabyte scale file system for the complete.txt file. One job has a corresponding complete.txt file indicating the job finished. Once the view function finds one complete.txt for some job, it will go to the Output folder to find the corresponding result in the folder named after that finished job, and send it back to the frontend. The frontend will overlay this result onto the underlying google map.

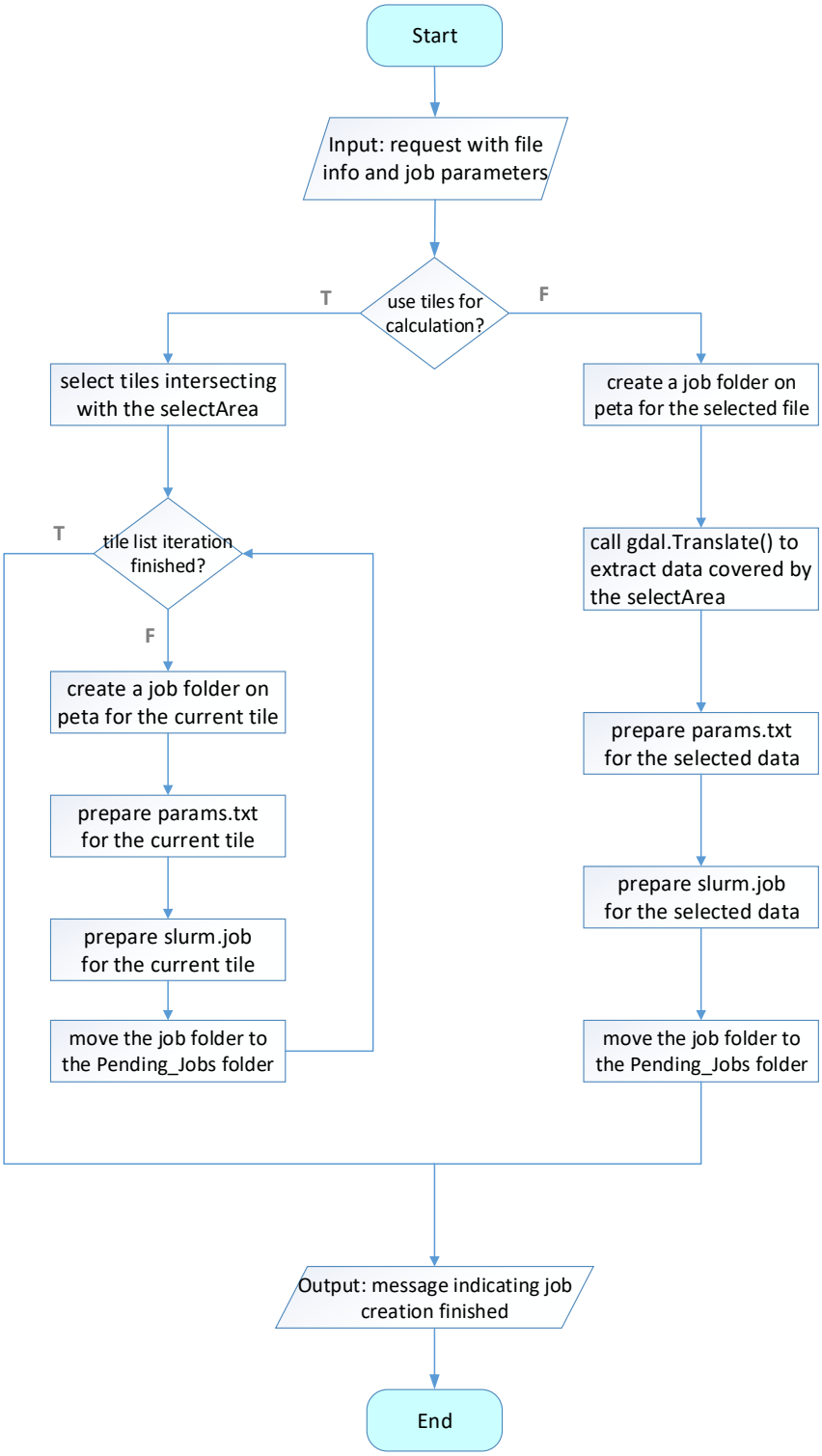

Figure 30. The workflow of the calculate view

One thing that needs mention about the marketplace is, it builds a common application directory access module to handle all the file paths used in the application to avoid hard codes in the development. All the file paths are stored in configuration.json, and I created readConfiguration() and writeConfiguration() for me to manipulate some directory stored in configuration.json.

3.10 Upload_publication

Like the upload view function, the upload_publication view function and its template upload_publication.html are used to let a login user upload publications he wants to share or manage for personal use only. The workflow of the upload_publication view function is shown in Figure 31. As this figures shows, the upload_publication view function mainly involves two major stemps. First, store uploaded publication to petabyte scale file system. Next, store the info of the uploaded publication to database on the local server. In the first step, when the view function receives the request, it first create a folder named after the uploaded publication in the Uploads folder to store the uploaded publication. The file name to be written to the petabyte file system is a combination of the filename of the publication and a timestamp to avoid possible conflicts because different users may upload files of the same file names. Line 1-5 in the code snippet in Table 35 shows the key logic. The second step is Line 6-7 which use the uploaded publication information to create a Publication object and save it into the database. The upload_publication.html is shown in Figure 32. This template is like the template

upload.html. It also deals with the dropzone, which accepts user's file input and process it when the user clicks the upload button.

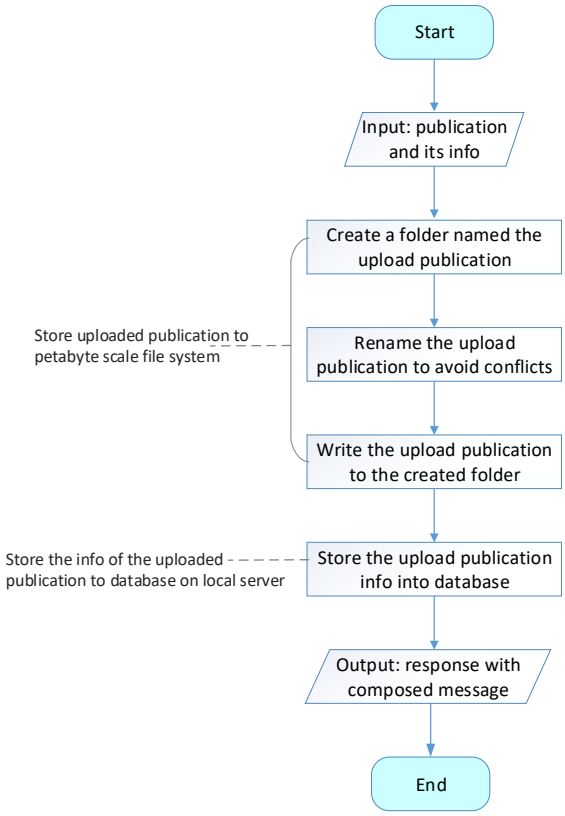

Figure 31. The workflow of the upload_publication view

```
......
1. file_name_new = ".".join([strftime("%H_%M_%S_%m_%d_%Y"), file_name.split(".")[1]])
2. file_location = os.path.join(file_path, file_name_new)

3. with open(file_location, 'wb+') as destination:
4.         for chunk in fileData.chunks():
5.                 destination.write(chunk)
......
6. publication = Publication(title=title, category=category, authors=authors,
            institution=institution, publication_date=publication_date, price=price,
            keywords=keywords, public_or_not=public_or_not, abstract=abstract,
            file_name=file_name, file_location=file_location,
            uploader_id=user_id, upload_date=upload_date)
7. publication.save()
```

Table 35. The code snippet of the upload_publication view

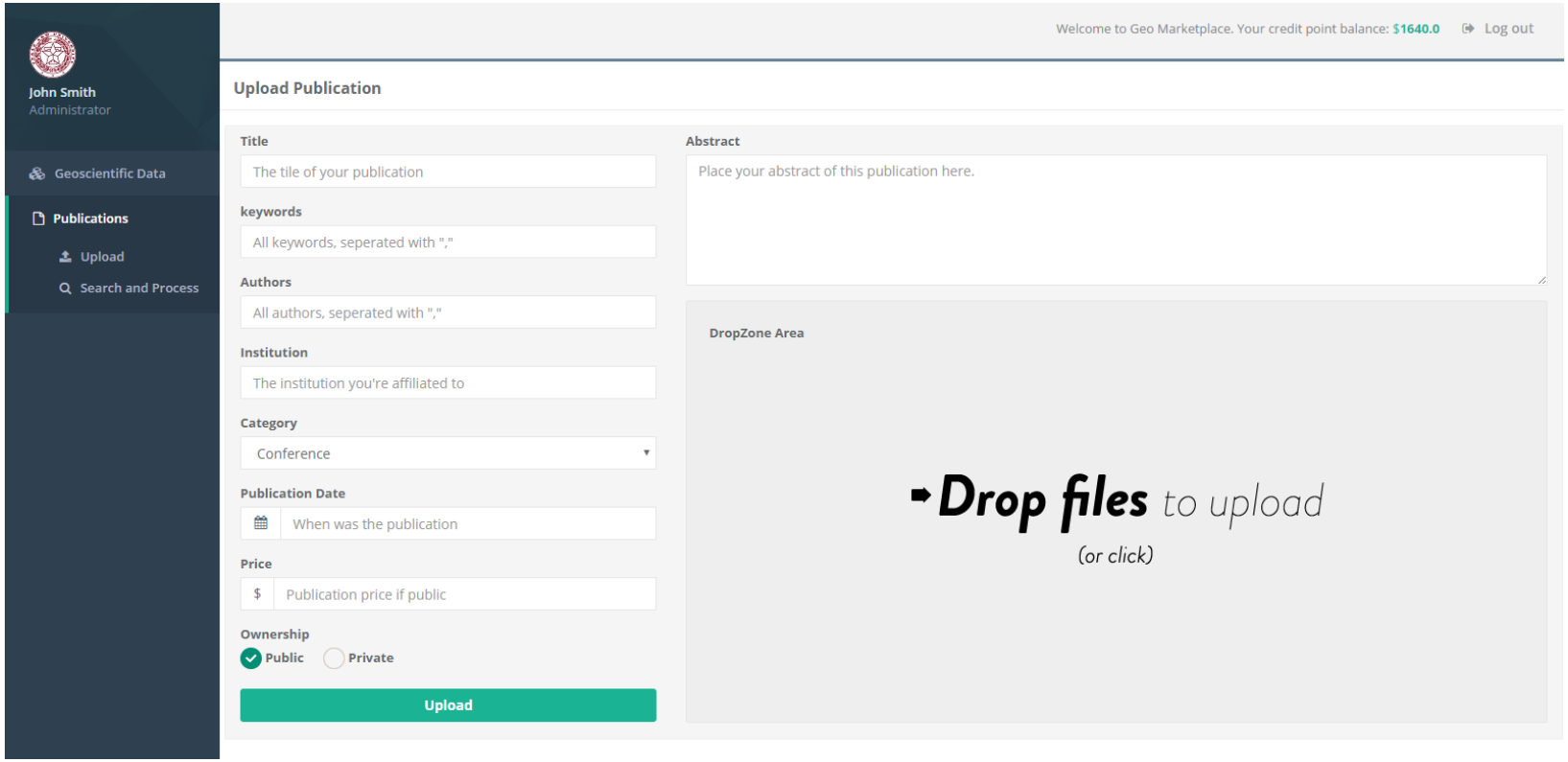

Figure 32. The upload_publication.html template

3.11 Search_publication

Like the search view function, the search_publication view and its template search_publication.html is used to help the user browse the published publications and do possible operations on them. One important thing is, we must filter out those publications that are private and not belong to the login user and only keeps those that are owned by the login user, public or private and public files that are exposed by the other users. Moreover, when the user browses the publications, he needs to know if he can process them. All data uploaded by the user himself, public or private, can be processed. Publications published by others and purchased by the user can also be processed. The main workflow of the search view function is shown in Figure 33.

One thing that needs mention is in search_publication.html, there is a column called processing. One row that is marked with 0 in the processlist will show nothing, and one that is marked with 1 will show a “processing” like for further processing. The key part of the search view function is shown in Table 36.

```
1. publicationSet = Publication.objects.filter(Q(public_or_not=1) | Q(uploader_id=user_id))
2. purchaseList = Transaction_publication.objects.filter(user_id = user_id)
3. publicationList = []
4. processList = []
5. uploaderList = []
6. for publication in publicationSet:
7.         if os.path.exists(publication.file_location):
8.                 publicationList.append(publication)
9.                 uploaderList.append(User.objects.get(pk=publication.uploader_id).username)
10.               if user_id==publication.uploader_id or purchaseList.filter(publication_id=publication.id).exists():
11.                       processList.append(1)
12.               else:
13.                       processList.append(0)
```

Table 36. The code snippet of the search_publication view

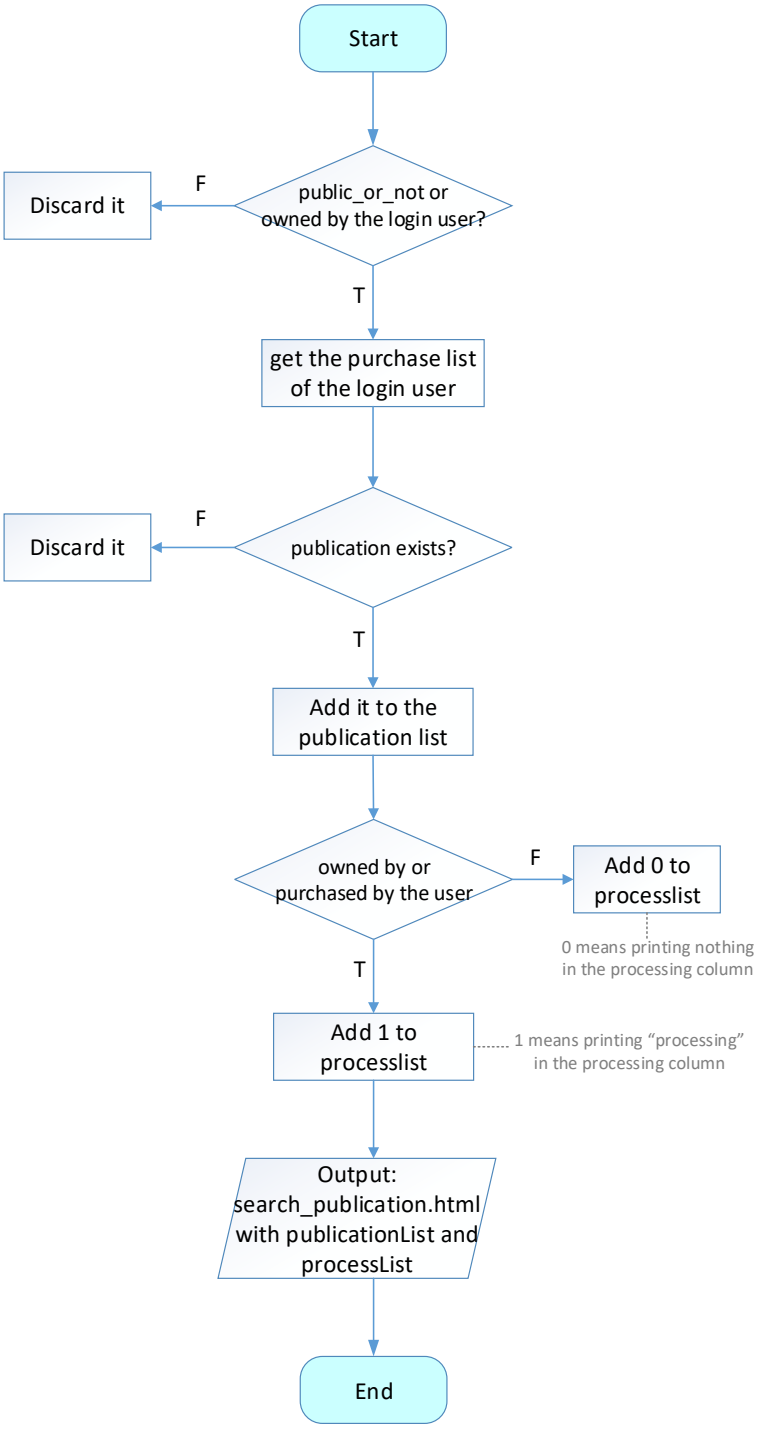

Figure 33. The workflow of the search_publication view

The template search_publication.html is shown in Figure 34.

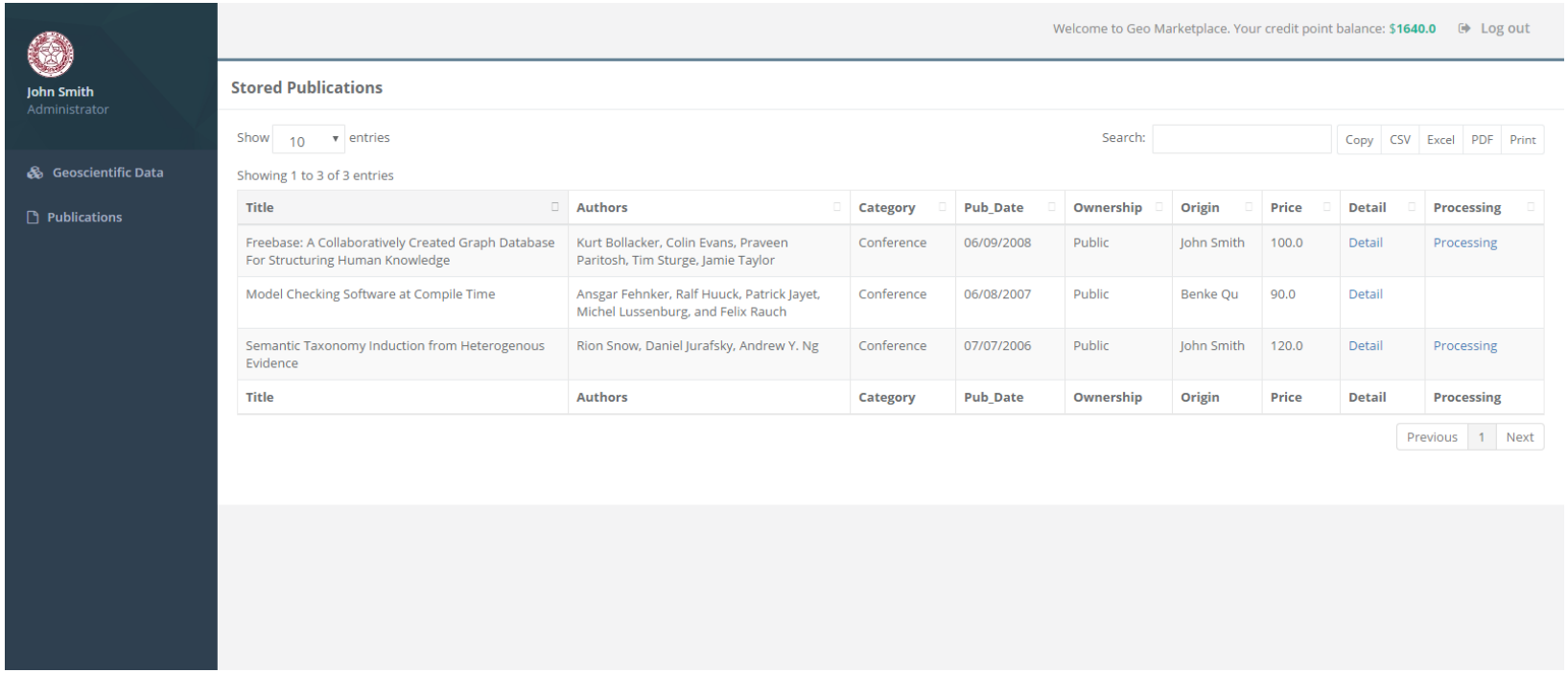

Figure 34. The search_publication.html template

3.12 Detail_publication and save_publication

From Figure 34, we can see there is one column called Detail for each row of result. This is used to check the details of listed publications. This function is fulfilled by the detail_publication view function and its template called detail_publication.html. The simple workflow of the detail_publication view is shown in Figure 35.

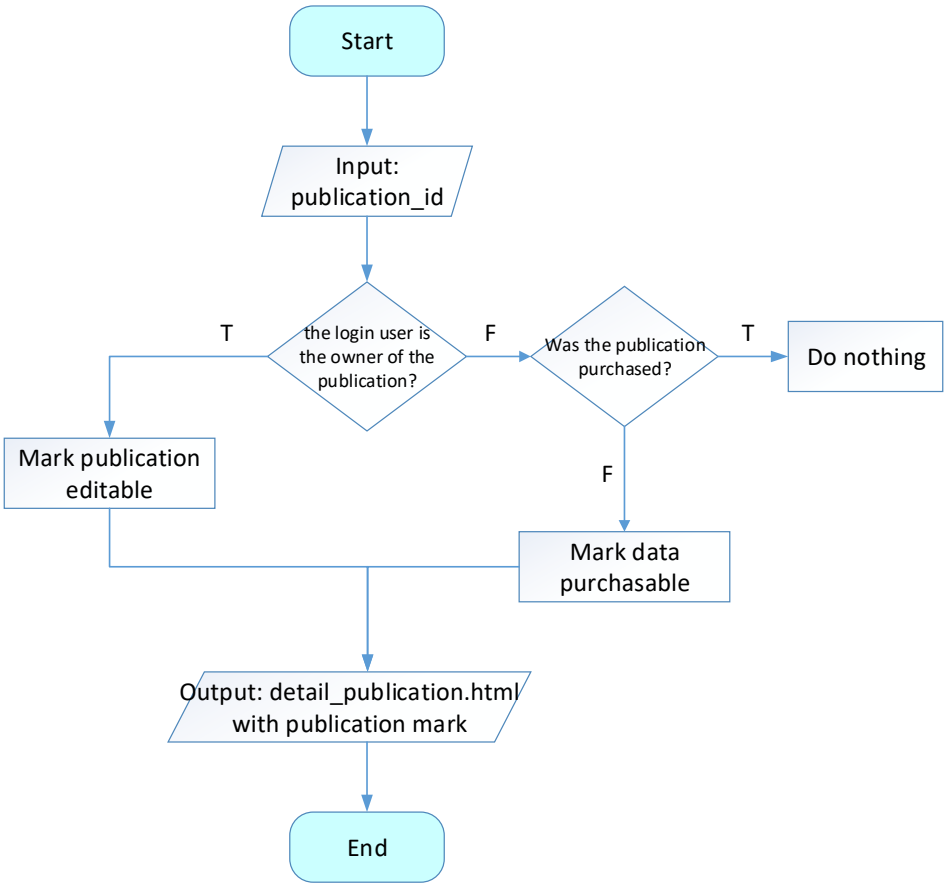

Figure 35. The workflow of the detail_publication view

As Figure 35 shows, having the publication in hand, the detail_publication function needs to ask the database if the publication that needs checking details is owned

by the login user. If yes, the detail_publication.html template will show the data editable, otherwise, if the publication was not purchased the login user previously, the detail_publication.html template will show the publication purchasable. If it was already purchased by the user, the user can do nothing more to the publication. He can only check the details of the publication. The key part of the detail view function is shown in Table 37. It implements the logic expressed in the workflow.

```
publication = Publication.objects.get(pk=publication_id)
    purchased = False
    if user_id != publication.uploader_id:
        if Transaction_publication.objects.filter(user_id=user_id, publication_id=publication.id).exists():
            purchased = True
```

Table 37. The code snippet of the detail_publication view

The detail_publication.html template is shown in Figure 36. If the publication is purchasable for the login user, a green button named “Buy this publication”will show up in the bottom right corner and the publication details cannot be edited by the user. If the publication is owned by the user, then he can edit the publication details and save the update. In this case, the bottom right corner first shows a button named “Edit this publication”. If the user clicks the button, it will change to “Save this publication”, all the columns in this page will change to the editable mode. The “Go back to publication list” button will take the user back the search_publication page. The view functions order_publication and payment_publication are similar to the order and payment view functions, respectively.

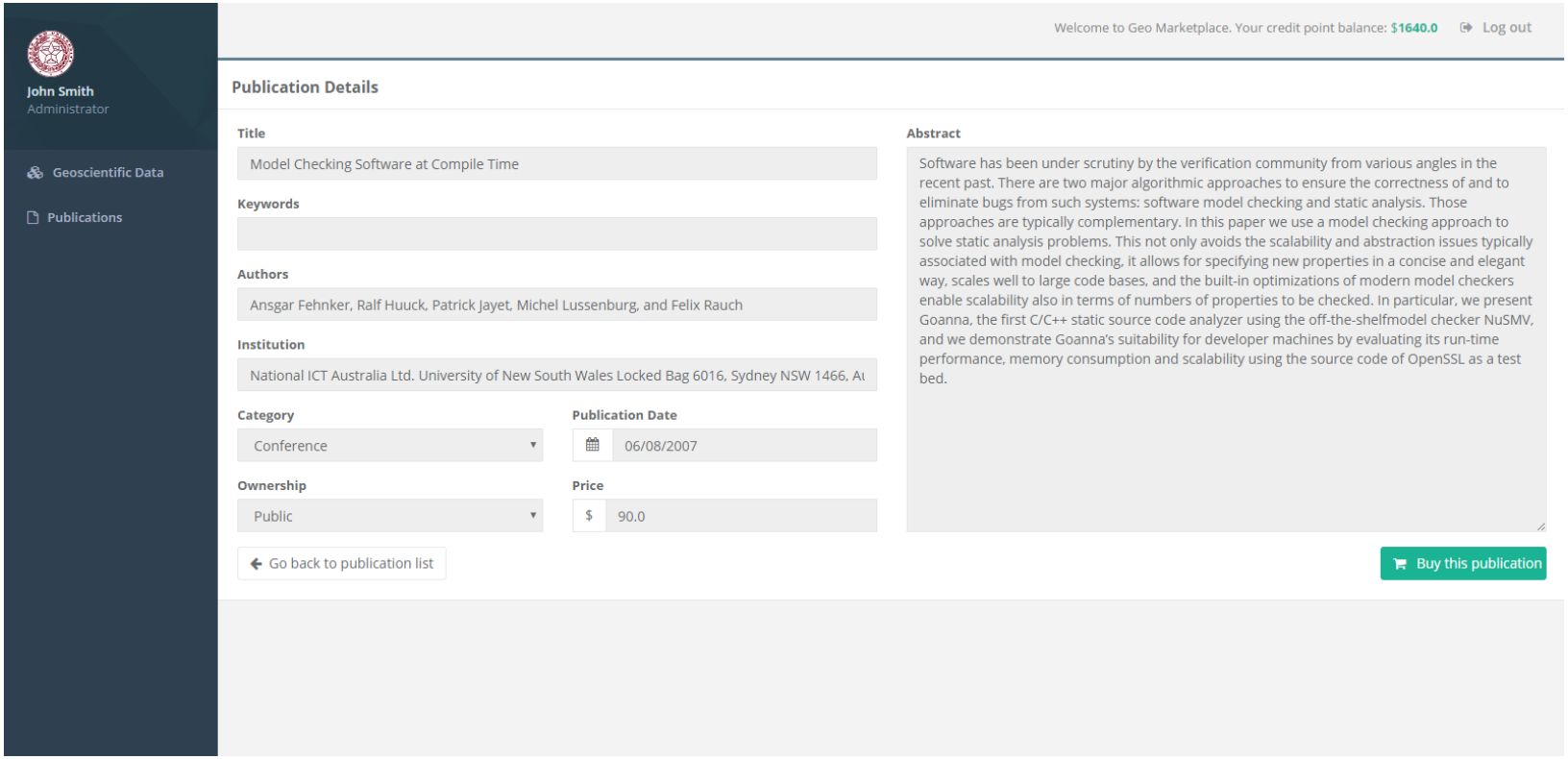

Figure 36. The detail_publication.html template

3.13 Process_publication

The process_publiation view function is by no means like the process view function. It generates a keyword-weight list for a publication of pdf version. The keyword-weight list stores all the keywords, produced by the keyword generation service which is hosted in the other server in the lab, and their weights in that publication. The code snippet is shown in Table 38, and the process_publication.html is shown in Figure 37. The process_publication.html uses a table and a bar chart to visualize the keyword-weight list.

```
fname = publication_location[publication_location.rindex(os.path.sep)+1:]
    keyword_weightList = None
    with open(publication_location, 'rb') as f:
        keyword_weightList = requests.post(
            'http://128.194.140.230:8889/get_signature',
            data={'fname':fname},
            files={fname: f})
    return json.loads(keyword_weightList.text)
```

Table 38. The code snippet of the process_publication view

The process_publication view function doesn't realize its own keyword-weight list generation function. Instead, it simply sends a service request to the keyword-weight generation service provider named get_signature, which is hosted in the server listening on 128.194.140.230:8889. The service get_signature is supported by Xing, who used a series of machine learning and natural language processing stuff to realize these service.

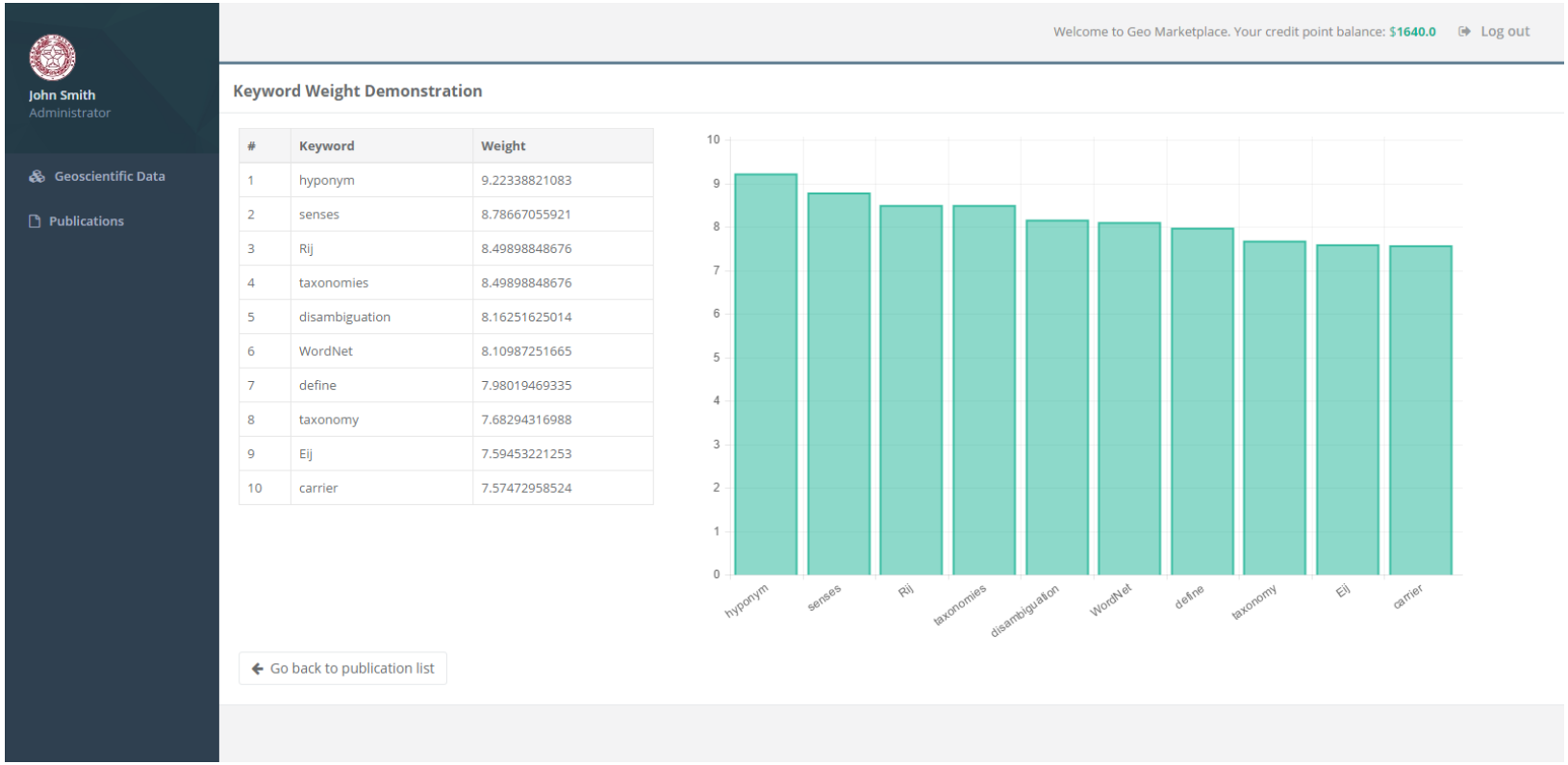

Figure 37. The process_publication.html template

4. CONCLUSIONS

The geoscience & technology marketplace provides the following functions to ease users' data management and operations: (a) GIS data uploading and tiling, (b) Centralized indexing and tracking of datasets, (c) Browsing and editing the stored GIS data and their details, (d) Data processing of selected areas on google map and its result visualization, (e) Data transaction, (f) Publication management, and (g) Publication processing.

Right now, the data processing only involves the calculation of solar radiation model. In the future, the marketplace will open a portal for the user to publish his calculation service. The other users can purchase a proper tool service among a list of published tools for sale. Once the user purchases a tool, he can make use of this tool to manipulate some data of his own or purchased by him.

The gis file transaction and the publication transaction service provided by the marketplace are now separated. They should be integrated as a common item transaction service. This is because the publication related service was added later into the marketplace, the first design of the marketplace only took consideration of the transaction of gis files, which caused some gis file related variables bound to the order and payment view functions and their templates. Next step, we will build a more general and common transaction service to accommodate possible future expansion of the marketplace. Besides, paypal and credit card payment methods are yet to be supported in the marketplace.

REFERENCES

[1] Wang, Shaowen. "A CyberGIS framework for the synthesis of cyberinfrastructure, GIS, and spatial analysis." Annals of the Association of American Geographers 100.3 (2010): 535-557.

[2] Wang, Shaowen, et al. "CyberGIS software: a synthetic review and integration roadmap." International Journal of Geographical Information Science 27.11 (2013): 2122-2145.

[3] Anselin, Luc. "From SpaceStat to CyberGIS: Twenty years of spatial data analysis software." International Regional Science Review 35.2 (2012): 131-157.

[4] Padmanabhan, Anand, et al. "FluMapper: an interactive CyberGIS environment for massive location-based social media data analysis." Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery. ACM, 2013.

[5] Liu, Yan, Anand Padmanabhan, and Shaowen Wang. "CyberGIS Gateway for enabling data‐rich geospatial research and education." Concurrency and Computation: Practice and Experience 27.2 (2015): 395-407.

[6] Wang, Shaowen, Nancy R. Wilkins-Diehr, and Timothy L. Nyerges. "CyberGIS-Toward synergistic advancement of cyberinfrastructure and GIScience: A workshop summary." Journal of Spatial Information Science 2012.4 (2012): 125-148.

[7] High Performance Research Computing. https://sc.tamu.edu/wiki/index.php/HPRC:About

[8] Google Maps API. https://developers.google.com/maps/documentation/javascript/tutorial

[9] GDAL. http://www.gdal.org/

[10] Open Source Geospatial Foundation. http://www.osgeo.org/

[11] PostgreSQL. https://www.postgresql.org/about/

[12] MVC. https://en.wikipedia.org/wiki/Model-view-controller

[13] Apache HTTP Server. https://httpd.apache.org/ABOUT_APACHE.html

[14] Nginx. https://www.nginx.com/resources/wiki/

[15] Django. https://www.djangoproject.com/start/overview/

[16] Mod_wsgi. https://pypi.python.org/pypi/mod_wsgi

[17] Apache Haus. https://www.apachehaus.com/

[18] ASF. https://en.wikipedia.org/wiki/Advanced_Systems_Format

[19] LFD. http://www.lfd.uci.edu/~gohlke/pythonlibs/

[20] Regular expression. https://en.wikipedia.org/wiki/Regular_expression

[21] Inspinia.js. https://wrapbootstrap.com/theme/inspinia-responsive-admin-theme-WB0R5L90S

[22] HTML5. https://en.wikipedia.org/wiki/HTML5

[23] Bootstrap. https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

[24] Dropzone. http://www.dropzonejs.com/

[25] ChartJS. http://www.chartjs.org/

[26] Datepicker. https://bootstrap-datepicker.readthedocs.io/en/latest/

[27] DataTables. https://datatables.net/

[28] Dobreva, I. D., et al. "Development, Evaluation and Parallelization of a Spatio-Temporal, Topographic, and Spectral GIS-based Solar Radiation Model."