



## Postgraduate Certificate in Software Design with Artificial Intelligence

### Applied Scripting Languages

#### Assignment 2: Research Assignment

Josh Quinn  
Student ID: A00279344

*The aim of this project was to create a set of Application Programming Interfaces (APIs) to allow an end user to store and retrieve data and execute a data mining algorithm on stored datasets. Specifically, the aim was to research and use a protocol that is used to do this and research python packages to implement it.*

*To achieve this, I have chosen to create a client-server application using Representational State Transfer (REST) principles using the python framework Flask. This application will contain the APIs to store and retrieve data and run a data mining algorithm on the data.*

### Contents

Representational State Transfer (REST).....	3
Types of Client and Server: .....	3
Sending Requests:.....	3
HTTP Verbs.....	3
Headers .....	3
Paths.....	3
Anatomy of a URL: .....	4
Sending Responses: .....	4
Response Body:.....	4
Content Type:.....	4
Other headers: .....	4
HTTP Status Codes: .....	5
Conclusion.....	5
Overview of the Program.....	6
Program Design.....	10

Use Case Diagram .....	10
System Design Diagram .....	11
Python Files in the Application .....	12
Application REST APIs .....	12
Testing.....	13
Research Journal .....	15
References .....	34

## Introduction: The Research

I have chosen to use Flask, a lightweight web application development framework that has a variety of features to build RESTful APIs. It does this using **Werkzeug**, a library to interact with Web Server Gateway Interface (WSGI). According to the WSGI website, “WSGI is the Web Server Gateway Interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. WSGI is a Python standard described in detail in PEP 3333.”]

Flasks use of Werkzeug and WSGI enables us to implement APIs in the REST architectural style, specifically by using HTTP protocol.

## Representational State Transfer (REST)

REST is a client-server-based architectural style that is comprised of six key constraints that standardize how content is transferred over the web. It was proposed by Roy Fielding in his 2000 dissertation ‘Architectural Styles and the Design of Network-based Software Architectures’. It uses Hyper Text Transfer Protocol (HTTP) to transfer resources or content between a client and a server.

Two key concepts in REST are the request object and the response object. A client makes a request to retrieve or modify the resource or content (the datasets or algorithms, in our case), and the server sends a response to those requests. These are the two concepts I will focus on implementing for this project.

### Types of Client and Server:

We can use different types of clients to make use of these APIs. We can use a web browser, which enables us to integrate our APIs with a user interface, or with a simple API client, such as Postman or Curl, which uses the HTTP request type and URI to make requests to our APIs. For this project Postman will be used as the API client to manually test the APIs.

### Sending Requests:

The following are the main components of a client request object.

#### HTTP Verbs

The client sends the request using HTTP protocol. HTTP verbs are a part of this protocol and define the type of operation of the request. The four basic verbs or operation types are GET to retrieve a resource, POST to create a resource, PUT to update a resource, and DELETE to remove a resource.

#### Headers

Headers are information about the request. They provide information about the request to server. Headers are written in key-value pairs. The most important request header for this project will be the Accept header. The accept header tells the server what type of content the client can receive in a response. This can be specified in many types including but not limited to JSON, text, XML, csv, and HTML. If the server tries to respond with JSON to request whose Accept header specifies HTML, then the request will throw an error. This helps standardize our APIs.

#### Paths

The path is the location of the resource we want to make requests on. The path is contained in the URL – the uniform resource location. Resources are requested by calling the specified application

URLs with a specified operation type. These are entry points to the application and are used to create, update, retrieve and delete data to and from the back end of the application, usually a database server.

#### Anatomy of a URL:

The URL specifies the protocol used for transferring content; the domain name of the application that abstracts the host IP; the port at which the application is available on, on the host; the path that specifies the resource to make a request on; and optional query parameters to specify with the resource.

Take the below as an example:

<http://datacalculator.com:8080/algorithms?type=linear-regression&dataset=wine>

**Protocol** – http

**Domain name** – datacalculator.com

**Port** – 8080

**Path** – /algorithms

**Query** – ?

**Parameters** – type=linear-regression, dataset=wine

The protocol used is hypertext transfer protocol, other types include https ('s' for secure, where sensitive information is hashed) or ftp (file transfer protocol). The domain is datacalculator.com. The port the application is available on is 8080 (where we make our requests to). The path is /algorithms and is where the data mining algorithms resource can be requested from. We can use query parameters then to specify the data mining algorithm to use, in this case linear regression, and the dataset which we want to run the algorithm on.

#### Sending Responses:

The following are the main components of the server response object.

##### Response Body:

The response body contains the requested resource or content. If we ran a GET request with the URI <http://datacalculator.com:8080/algorithms?type=linear-regression&dataset=wine> we could expect that our response body would contain the statistical information from the algorithm output. Along with this it can also contain information regarding the outcome of the request, including HTTP status Codes, and an additional informational message.

##### Content Type:

The content type header tells us what hypermedia format the response body is in. For instance, it could be in json, html, or xml format. Depending then on what our *request* accept header has specified, the response will or will not be acceptable. If the request has an accept header of json, and the response content-type header specifies xml an error will be thrown. This again is a way to standardize our APIs.

##### Other headers:

Other headers of import are Link, Location, Last-modified. The link header when specified with a relation type, or 'rel', tells us if there is another associated or related content to this. It is often used when paginating content.

## HTTP Status Codes:

HTTP status codes, or response codes, are very important component of the response body. They tell us specific information on the outcome of the request. There are many types of response codes. They are broken into categories. According to IETF, the following categories exist:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The most common are:

Status Code	Description
200	OK, the request has succeeded.
201	Created, the resource has been created.
400	Bad Request, the request could not be fulfilled due to malformed syntax.
404	Not found, the resource requested does not exist.
500	Internal Server Error, the server itself encountered some error while processing a request.

## Conclusion

Through our FLASK app APIs, we can make client requests and the server will send back responses to those requests.

For instance, we can make a request to run an algorithm on a saved dataset through one of our APIs. This request could go into the backend (the storage), retrieve the dataset specified in the request, and run the specified algorithm on that dataset. If successful, the server will return a response object containing the analysis and statistics from the algorithm, plus other information about the response such as any related links, the dataset name, the location of the dataset, etc.

The Flask framework allows for a structured way to build simple RESTful APIs to request data from a server and respond to a client with the data. We can use Flask's rich set of features to do this in a local client-server application. It is also one of the most popular web frameworks for python with a broad documentation and use.

## Overview of the Program

### 1. Describe the purpose of your program.

The purpose of the program is to store and retrieve datasets and run a data mining algorithm on specified datasets by implementing a set of RESTful APIs in a client-server application. This will be achieved by using RESTful principles and implementing a set of APIs.

### 2. How it is based on your research.

My research analysed protocols to understand how to build restful APIs and software to build APIs. I deemed Flask an excellent option to help build a simple client-server application and implement a set of RESTful APIs for data retrieval due to its rich set of libraries and wide documentation and use amongst the python community.

### 3. Any additional software that must be installed for the program to work.

I installed Flask and sklearn python packages and the rest client Postman. However, the command line tool curl can also be used for making requests to our endpoints.

# To create the client-server application to serve response install Flask.

```
$ pip install Flask
```

# To get premade datasets for mock database install sklearn

```
$ pip install sklearn
```

### 4. Overview of Use

#### 1. Start application using main.py (this prevents ModuleNotFoundError issues):

```
$ python main.py
```

#### 2. Create sample dataset. Must be a csv file:

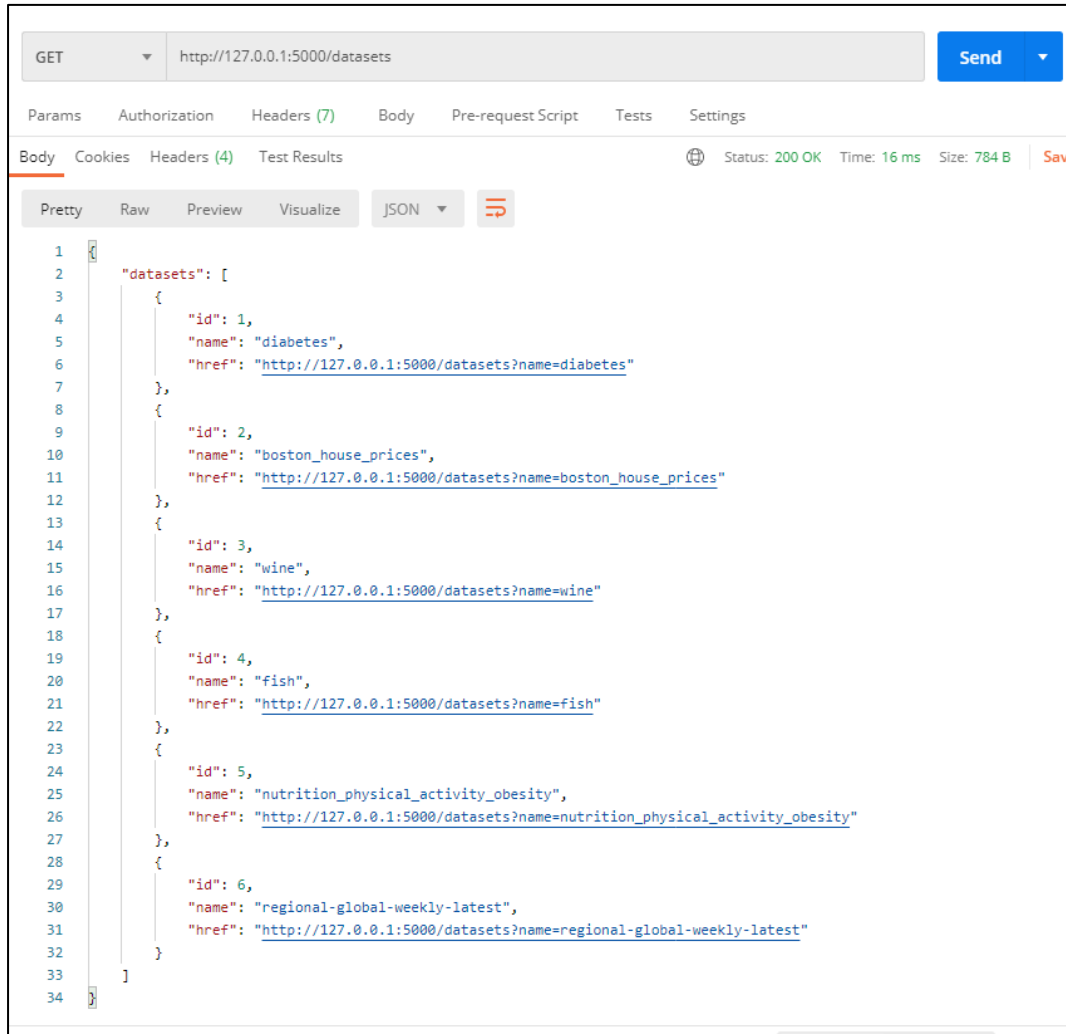
The screenshot shows a Postman interface for a POST request to `http://127.0.0.1:5000/datasets`. The request body is set to 'file' with the file 'fish.csv' selected. The response status is 200 OK, with a time of 18 ms and a size of 19.49 KB. The response headers are displayed below.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> file	fish.csv	
Key	Value	Description

KEY	VALUE
Location	<code>http://127.0.0.1:5000/datasets?id=4</code>
Content-Type	<code>application/json</code>
Content-Length	<code>19695</code>
Dataset	<code>fish</code>
Link	<code>&lt;http://127.0.0.1:5000/datasets?id=5&gt;; rel='next'</code>
Server	<code>Werkzeug/1.0.1 Python/3.9.1</code>
Date	<code>Mon, 11 Jan 2021 22:59:41 GMT</code>

### 3. Get all the datasets available datasets:



GET <http://127.0.0.1:5000/datasets> Send

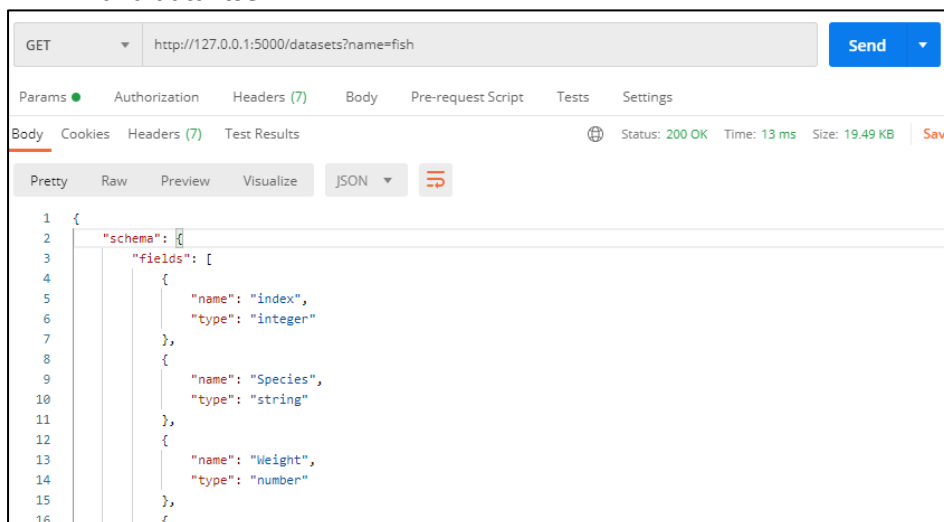
Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body Cookies Headers (4) Test Results Status: 200 OK Time: 16 ms Size: 784 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "datasets": [
3     {
4       "id": 1,
5       "name": "diabetes",
6       "href": "http://127.0.0.1:5000/datasets?name=diabetes"
7     },
8     {
9       "id": 2,
10      "name": "boston_house_prices",
11      "href": "http://127.0.0.1:5000/datasets?name=boston_house_prices"
12    },
13    {
14      "id": 3,
15      "name": "wine",
16      "href": "http://127.0.0.1:5000/datasets?name=wine"
17    },
18    {
19      "id": 4,
20      "name": "fish",
21      "href": "http://127.0.0.1:5000/datasets?name=fish"
22    },
23    {
24      "id": 5,
25      "name": "nutrition_physical_activity_obesity",
26      "href": "http://127.0.0.1:5000/datasets?name=nutrition_physical_activity_obesity"
27    },
28    {
29      "id": 6,
30      "name": "regional-global-weekly-latest",
31      "href": "http://127.0.0.1:5000/datasets?name=regional-global-weekly-latest"
32    }
33  ]
34 }
```

### 4. Get one dataset – response body will be JSON. We can view the dataset schema, fields, and data itself:



GET <http://127.0.0.1:5000/datasets?name=fish> Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Body Cookies Headers (7) Test Results Status: 200 OK Time: 13 ms Size: 19.49 KB Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "schema": {
3     "fields": [
4       {
5         "name": "index",
6         "type": "integer"
7       },
8       {
9         "name": "Species",
10        "type": "string"
11      },
12      {
13        "name": "Weight",
14        "type": "number"
15      }
16    ]
17  }
```

```

42     "data": [
43     {
44         "index": 0,
45         "Species": "Bream",
46         "Weight": 242.0,
47         "Length1": 23.2,
48         "Length2": 25.4,
49         "Length3": 30.0,
50         "Height": 11.52,
51         "Width": 4.02
52     },
53     {
54         "index": 1,
55         "Species": "Bream",
56         "Weight": 290.0,
57         "Length1": 24.0,
58         "Length2": 26.3,
59         "Length3": 31.2,
60         "Height": 12.48,
61         "Width": 4.3056
62     },

```

## 5. Delete dataset:

DELETE
http://127.0.0.1:5000/datasets/1
Send

Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings

Body
Cookies
Headers (3)
Test Results
Status: 204 NO CONTENT
Time: 6 ms
Size: 133 B
Sa

Pretty
Raw
Preview
Visualize
JSON

1

## 6. Get available data mining algorithms:

GET
http://127.0.0.1:5000/algorithms
Send

Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings

Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 17 ms
Size: 229 B
Sav

Pretty
Raw
Preview
Visualize
JSON

1 {
2 "id": "1",
3 "type": "linear regression"
4 }



## 7. Get data analysis on a dataset with query parameters:

GET

http://127.0.0.1:5000/data-analysis?algorithm=linear regression&dataset=boston\_house\_prices&independen

Send

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	algorithm	linear regression	
<input checked="" type="checkbox"/>	dataset	boston_house_prices	
<input checked="" type="checkbox"/>	independents	NOX,RM,AGE	
<input checked="" type="checkbox"/>	dependent	MEDV	
	Key	Value	Description

Below is the response with data analysis:

GET

http://127.0.0.1:5000/data-analysis?algorithm=linear regression&dataset=boston\_house\_prices&independen

Status: 200 OK Time: 31 ms

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Body

Pretty

Raw

Preview

Visualize

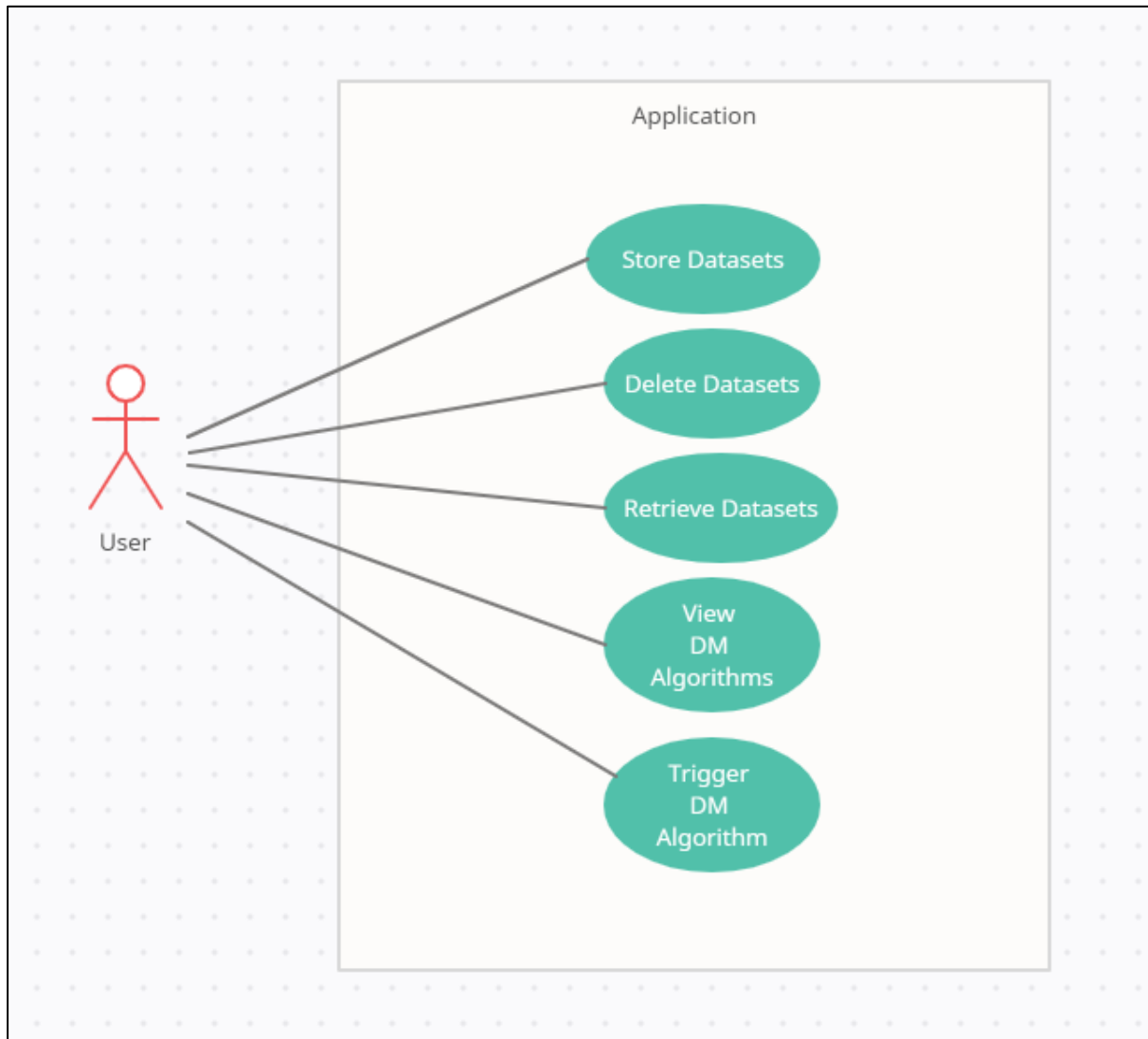
Text

```
1 | | | | | OLS Regression Results
2 |=====
3 | Dep. Variable: MEDV R-squared (uncentered): 0.930
4 | Model: OLS Adj. R-squared (uncentered): 0.930
5 | Method: Least Squares F-statistic: 2240.
6 | Date: Mon, 11 Jan 2021 Prob (F-statistic): 1.66e-290
7 | Time: 23:16:08 Log-Likelihood: -1659.0
8 | No. Observations: 506 AIC: 3324.
9 | Df Residuals: 503 BIC: 3337.
10 | Df Model: 3
11 | Covariance Type: nonrobust
12 |=====
13 | | | | | coef std err t P>|t| [0.025 0.975]
14 |-----
15 | NOX -23.3933 3.127 -7.481 0.000 -29.537 -17.249
16 | RM 5.9800 0.183 32.683 0.000 5.621 6.339
17 | AGE -0.0283 0.015 -1.905 0.057 -0.057 0.001
18 |=====
19 | Omnibus: 209.386 Durbin-Watson: 0.589
20 | Prob(Omnibus): 0.000 Jarque-Bera (JB): 1017.898
21 | Skew: 1.790 Prob(JB): 9.25e-222
22 | Kurtosis: 8.955 Cond. No. 813.
23 |=====
24 |
25 | Notes:
26 | [1] R² is computed without centering (uncentered) since the model does not contain a constant.
27 | [2] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

## Program Design

1. *Provide a detailed design of your program.*

### Use Case Diagram

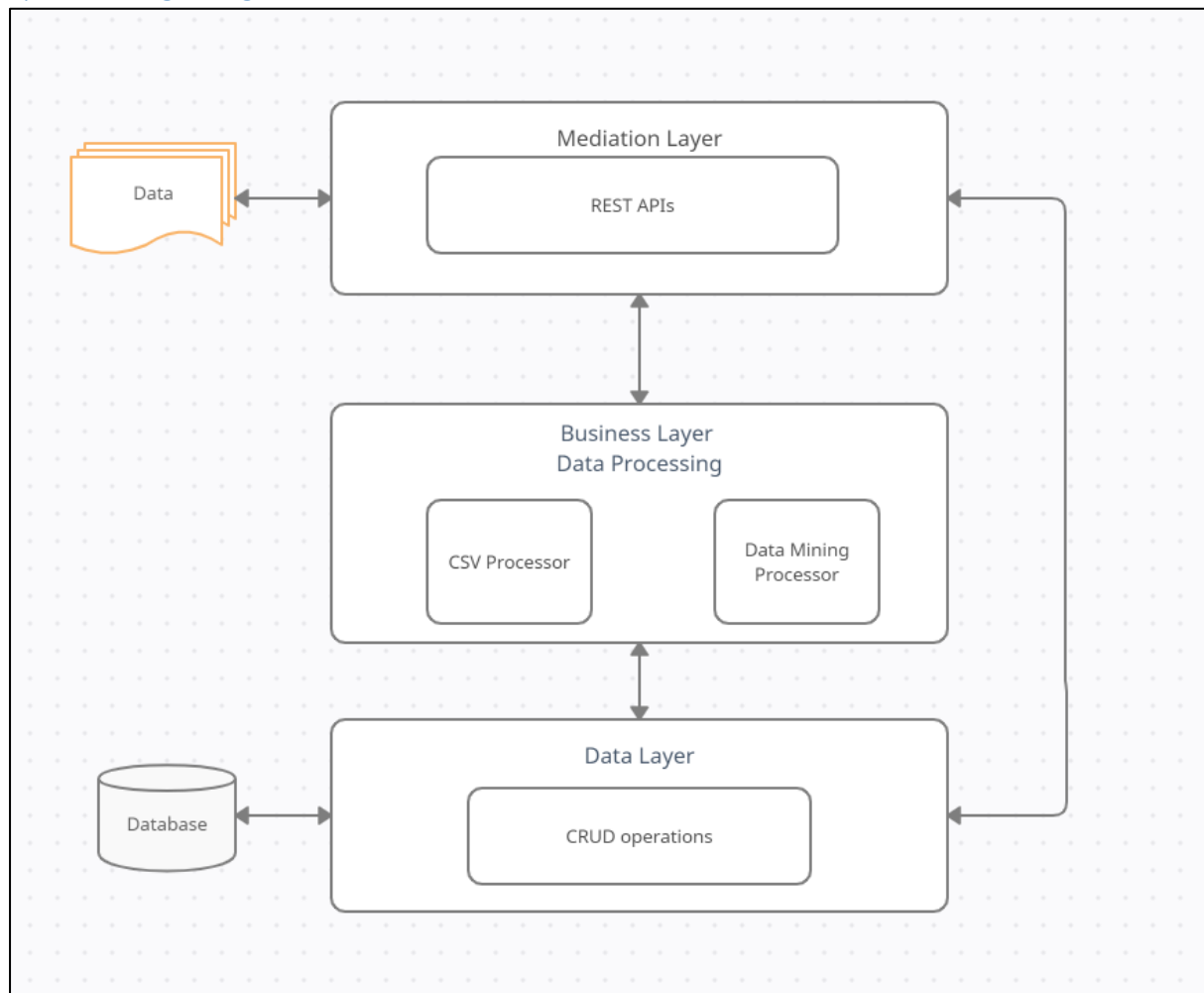


*\*DM = Data Mining*

There are five main use cases of the program:

1. Store Datasets
2. Delete Datasets
3. Retrieve Datasets
4. View Data Mining Algorithms available
5. Trigger Data Mining Algorithms on stored datasets

## System Design Diagram



The system has three primary components.

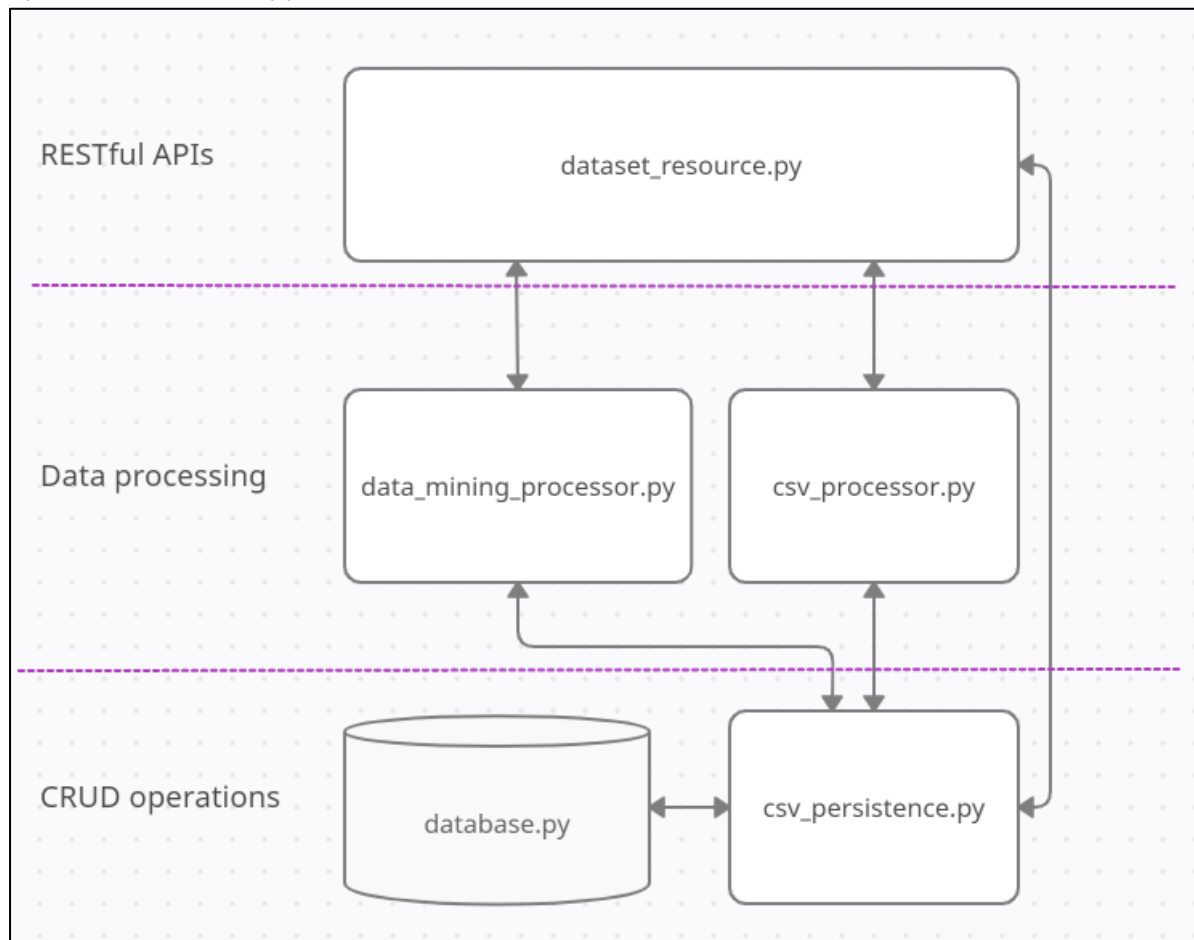
1. The mediation layer – where we mediate requests and responses. This is where REST APIs exist, the entry points to the application.
2. The data processing layer – the core business logic of the application, where data processing and manipulation occurs. This is where the data mining algorithms are run.
3. The data layer – data persistence, where CRUD operations are done against the database. These create, read, update, and delete operations align with POST, GET, PUT, and DELETE http verbs used for RESTful APIs.

The system components are split across different python files.

1. `dataset_resource.py`
2. `data_mining_processor.py`
3. `csv_processor.py`
4. `csv_persistence.py`
5. `database.py`

Included in the application is also a configuration file, `configuration.py`. This file holds the project root directory.

## Python Files in the Application



## Application REST APIs

URL for each begins with <http://<host>:<port>>, by default <http://localhost:5000/>

Id	HTTP operation	Path	Query Parameters (example values)	Description
01	POST	/datasets	NA	<p>Post a csv file to store a dataset in the database. Body must contain form-header with csv file attached to the request.</p> <p>The requests accepted MIME type is 'text/csv'.</p> <p>Response code is '201', created or '400' if exception during execution.</p> <p>If dataset already exists '303' response is sent, a redirect response, with Location header contain href to already existing resource.</p>
02	GET	/datasets	Empty, or ?id=1, or ?name=dataset123	<p>If no query parameter is specified, the names, ids, and href links of each of the available datasets are returned in JSON form.</p> <p>If the name or id query parameter is specified, retrieve the corresponding dataset from the database. Only one of 'id' or 'name' should be specified. JSON representation of the dataset is returned if the data exists.</p> <p>Accept header is 'application/json'</p> <p>Response code is '200', 'OK' or '400' if exception during execution.</p>
03	DELETE	/datasets/<id>	NA	Delete the dataset by id.

				Response code is '200', 'OK' or '400' if exception during execution.
04	GET	/algorithms	NA	Retrieve all the available data mining algorithms a user can choose from.  Accept header is 'application/json'  Response code is '200', 'OK' or '400' if exception during execution.
05	GET	/data-analysis	?algorithm='linear regression' &dataset=dataset123 &independents=feat1, feat4, feat7 &dependent=feat10	Run the data mining algorithm and get the data analysis from specified dataset.  Must contain the query parameters 'algorithm', 'dataset', 'independents', and 'dependent' for the linear regression algorithm to work.  Accept header is 'plain/text'  Response code is '200', 'OK' or '400' if exception during execution

## Testing

I wrote 7 Unit tests to test the APIs.

1. test\_get\_all\_datasets
2. test\_get\_one\_dataset
3. test\_dataset\_not\_exist
4. test\_post
5. test\_delete
6. test\_delete\_non\_existing\_resource
7. test\_get\_algorithms
8. tet\_get\_analysis

For these test cases I used the 'unittest' python package with Flask's in-built test\_client() method. This method instantiates a mocked client to make HTTP requests against the REST API.

TCID	Method	Endpoint	Description
01	GET	/datasets	Get all datasets. Assert the status code is 200 Assert the response 'datasets' dict has 5 entries.
02	GET	/datsets?name=wine	Get dataset with the name wine. Assert 200 response code Assert the first 'data' dict entry equals the expected.
03	GET	/datasets?name=unkown	Get dataset that does not exist. Assert 404 response code 'NOT FOUND'
04	POST	/datasets	Create dataset of content type 'application/json'. Assert 400 response code. Assert the error message in response body is as expected.
05	DELETE	/datasets/1	Delete existing resource with id 1. Assert response code 204.

06	DELETE	/datasets/6	Delete non-existing resource with id 6 Assert response code 404. Assert the expected message in response body.
07	GET	/algorithms	Get all the available data mining algorithms. Assert 200 response code. Assert the algorithm type is linear regression. Assert the id is equal to 1.
08	GET	/data-analysis/?algorithm=linear regression &dataset=boston_house_prices &independents=NOX,RM,AGE &dependent=MEDV	Get data-analysis on the dataset boston_house_prices, with linear regression algorithm, with the specified de- pendent and independent variables.  Assert the response contains the expected dependent variable string. Assert response contains the expected R-squared value. Assert 200 response code.

## Research Journal

Describe in detail the research you carried out and what learning you gained from it.

Entry Number: 1

Date: 23<sup>rd</sup> of December

Objective(s):

1. Think of a project idea – what would I like to learn/create?
2. Select the feature(s) to research/implement this project
3. Sketch out rough program design, configuring the features into this design

*Description of Work Done*

### 1. Think of a project idea – what would I like to learn/create?

Firstly, I have begun sketching out my idea.

I would like to create **client/server** application using **Representational State Transfer (REST)** architectural style, which incorporates a **data processing pipeline**.

My idea is for an end user to be able to save and pre-process any dataset to file storage through REST API and use other REST APIs to run statistics and data mining algorithms on any saved dataset.

The goal will be to create one simple data mining algorithm that can be used across multiple datasets. Ideally this algorithm will be configurable through API query parameter list, although this may be out of the scope of this project.

### 2. Select the feature(s) to research/implement this project

Included in the application will be:

**REST endpoints:**

- POST to create/save datasets
- DELETE to delete datasets
- GET to view saved datasets
- GET to view available data mining algorithms (e.g., 'simple linear regression').
- GET to run data mining algorithm on a specified dataset.

**Data processing:**

- Data processing module – simple data mining algorithm implementation(s) and basic statistics
- Data processing pipeline – where processing functions are called.

Points to note:

- For simplicity I will write/delete datasets to file instead of implementing database persistence.
- I will research how to create a simple client/server application in python
- I will research how to create RESTful endpoints in python.

### 3. Sketch out rough program design, configuring the features into this design

#### REST API module

- Contains the REST endpoints

#### Data layer module

- this module will contain the create, read, update and delete (CRUD) methods on the persistence.

#### Data Processing module

- Sub-module for retrieving, saving, or deleting data
- Sub-module for basic data pre-processing (data scrubbing)
- Sub-module for data mining/ML algorithm implementation

#### Roadmap:

- ➔ Research REST APIs
- ➔ Research python feature to implement REST APIs
- ➔ Implement basic REST API module
- ➔ Implement data module for CRUD functionality
- ➔ Implement the data processing module for the data mining algorithm

#### *Reflections (What I achieved/What I learned):*

I have been able to get an idea of what I want to achieve for this project. I have a basic structure of the project in mind and a roadmap of how to get there.

#### *Sources (References for this research)*

None

#### *Planning (What's Next)*

Next, I will research REST APIs.



Entry Number: 2

Date: 1<sup>st</sup> of January 2021

Objective: Research REST

*Description of Work Done*

- Read through REST Dissertation.
- HTTP protocol (operation types)
- Understand the basics of requests and response objects.
  - o Headers
  - o Status Codes
  - o MIME type / Content-type

*Reflections (What I achieved/What I learned):*

I have learned about the principles of REST architecture and how REST primarily is a client-server type architecture that leverages HTTP protocols to standardize how data/resources transport over the web. I have learned that REST APIs can be used in a versatile way by sending responses of different kinds for different use cases depending on the hypermedia types requested. They can be used to transport JSON and different text types or html pages for example.

I learned about the basic HTTP protocol operation types required for the transportation of resources. In my case I will use GET, POST and DELETE to retrieve, create and delete datasets to and from persistence.

As a core concept, I have researched the request and response objects. I have learned about the headers that can be used to give information about a request or response, i.e., meta-data about the request or response. I have also learned about status codes, positive status codes, and negative ones. This has taught how these elements help standardize how resources are transported across the web using REST principles.

I have also learned about the different parts of the Uniform Resource Identifier, which we use to call the endpoints in combination with the HTTP operation type, and optional data as well (for POST or PUT).

*Sources (References for this research)*

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>  
<http://ietf.org/assignments/http-status-codes/http-status-codes.xml>  
<https://doepud.co.uk/blog/anatomy-of-a-url>  
<https://www.codecademy.com/articles/what-is-rest>  
<https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/>

*Planning (What's Next)*

Next, I will research how to implement REST APIs using Python.

Entry Number: 3

Date: 2<sup>nd</sup> of January 2021

Objective: Research how to implement REST APIs with Python

### **Description of Work Done**

- Choosing a python package/package
- Understand how Flask implements WSGI
  - o Werkzeug modules
  - o Routing / Endpoints
- Understanding the Flask imports 'Flask', 'request', and 'response'

### **Reflections (What I achieved/What I learned):**

Firstly, I had to research what packages I could use to implement the REST APIs. Searching on Google I found that packages and frameworks could be used to build the REST APIs. I first looked in to the 'requests' python package.

The 'requests' package seemed like a good choice to learn and implement basic restful APIs. However, I could not find any information on how to support a server to listen to requests from a client such as postman or curl. I wanted to be able to use my APIs from a client such as this, so I kept searching. My search informed me of different python frameworks that could be used to do this.

The most well known were Flask and Django. Flask was touted as the simplest and most straight forward, so I decided to research that.

Flask is a framework that allows development of full-stack applications, with both front-end and back-end extensibility. I wanted to create a program that would be able to listen requests and serve responses, which Flask enables, so I continued with it. Flask enables this by using the Werkzeug library. This library is set of standardized functions to implement a HTTP server, specifically, by implementing a Web Server Gateway Interface server. This mediates the http requests to the backend of our application, to which the responses are returned to. It is specified in PEP 3333. This is implemented by Flask framework and is abstracted away from the developer. It is fully configurable, however. We can change the port and the IP/host if we wish. By default, the server is configured to listen on the localhost IP 127.0.0.1, with port 5000.

Flask has an import which is required to start the server. This import is 'Flask', this imports the WSGI application into our python file. Through this we can specify where our HTTP requests are to be mediated.

Figure 1 shows the imported server. From the Flask docs:

1. "First we imported the **Flask** class. An instance of this class will be our WSGI application."
2. "Next we create an instance of this class. The first argument is the name of the application's module or package...This is needed so that Flask knows where to look for templates, static files, and so on".

```

from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

```

Figure 1 Flask import, importing the WSGI server

We import the WSGI instance as 'app' so that we can route our request with the path location in the URI request to a method. This is commonly referred to as an endpoint. This is the entry to the program. Figure 1 shows the annotation `@app.route('/')`. '/' is the path location mapped to the `hello_world` function. The WSGI instance will route any request that has been sent to the path location '/' to this method.

Flask calls this annotation the **route()** decorator. As stated, this tells Flask what URL should trigger the method. The route decorator takes a 'rule' parameter, a string representing the URL to map the function to, and the 'options' parameter. These options are several parameters which is pushed the Werkzeug 'Rule' object that helps define allowed request characteristics. For this program the only option we will specify is the allowed HTTP methods for each endpoint.

```

from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()

```

Figure 2 Flask's `app.route()` methods parameter

Figure 2 shows the allowed HTTP methods when accessing the URL path '/login'. It is a comma separated list. By default, all HTTP methods are allowed if not specified. It also shows the 'request' import at work.

The request module allows us to access the incoming request. The Flask request object is a subclass of the werkzeug Request. This werkzeug 'Request' object contains other objects that cache the request components including the headers, the request body, files, cookies, the request path, url, base url, the accepted MIME types, HTTP operation and may other attributes and information. Figure 3 shows the Request object class's objects that holds all the request components, data, and information. For this application I will mostly call cached data and information from BaseRequest and AcceptMixin objects within the Request object.

```
class Request(
    BaseRequest,
    AcceptMixin,
    ETagRequestMixin,
    UserAgentMixin,
    AuthorizationMixin,
    CORSRequestMixin,
    CommonRequestDescriptorsMixin,
):
```

Figure 3 Werkzeug Request class.

The final key import I require for this program is Flask's Response module. This is the module that will store all the data and information to serve back to the client. These response properties include the headers, body, status, status code, data and mimetype. These are the properties that I will use in this program.

### ***Sources (References for this research)***

<https://palletsprojects.com/p/werkzeug/>  
<https://wsgi.readthedocs.io/en/latest/what.html>  
<https://www.python.org/dev/peps/pep-3333/>  
<https://werkzeug.palletsprojects.com/en/1.0.x/routing/#werkzeug.routing.Rule>  
<https://flask.palletsprojects.com/en/1.1.x/api/?highlight=request#flask.request>  
<https://werkzeug.palletsprojects.com/en/1.0.x/wrappers/#werkzeug.wrappers.Request>  
<https://flask.palletsprojects.com/en/1.1.x/api/?highlight=response#flask.Response>  
<https://werkzeug.palletsprojects.com/en/1.0.x/wrappers/?highlight=response#werkzeug.wrappers.BaseResponse.response>

### ***Planning (What's Next)***

Next, I will set up a Flask app and implement basic REST endpoints

Entry Number: 4

Date: 3<sup>rd</sup> of January 2021

Objective: Set up Flask app and implement basic REST endpoints

### **Description of Work Done**

- Installed flask with pip.

First, I install the python framework Flask using pip.

```
Josh@DESKTOP-JP7EV08 MINGW64 ~  
$ pip install flask
```

- Started up 'hello world' app.

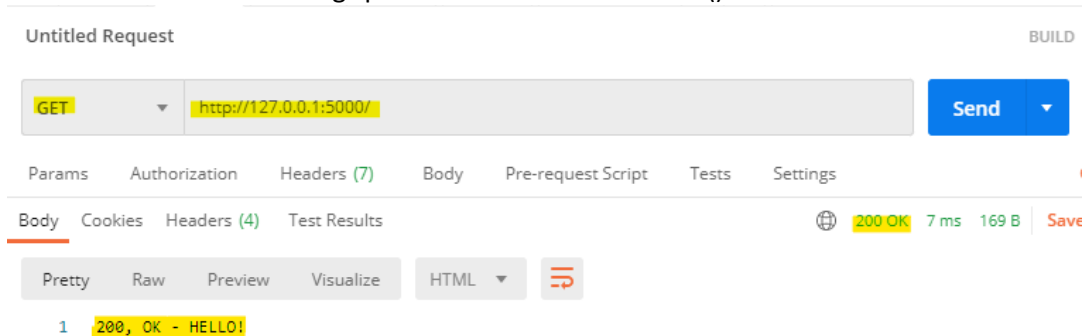
Once installed, I created a project in my IDE and set up a basic hello world project, following the Flask documentation to do so and using my research.

```
app.py x  
1 from flask import Flask  
2  
3 app = Flask(__name__)  
4  
5  
6 @app.route('/', methods=['GET'])  
7 def hello():  
8     return '200, OK - HELLO!'  
9  
10
```

I run the app.py file from the command line:

```
Josh@DESKTOP-JP7EV08 MINGW64 ~/Documents/University Work/AIT/MSc/Applied Scripting L  
$ python app.py  
* Serving Flask app "app" (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Using postman, I can make a request to URI specified above with a GET HTTP operation type and I will receive the return String specified in the function hello():



- Created the basic endpoints.

Following getting a basic hello world client-server app up and running I created the all the endpoints that I would need for my program:

```
from flask import Flask
app = Flask(__name__)

@app.route('/', methods=['GET'])
def hello():
    return '200, OK - HELLO!'

@app.route('/datasets', methods=['POST'])
def create_dataset():
    pass

@app.route('/datasets', methods=['GET'])
def get_dataset():
    pass

@app.route('/datasets/all', methods=['GET'])
def get_all_datasets():
    pass

@app.route('/datasets/<id>', methods=['DELETE'])
def delete_dataset(id):
    pass

@app.route('/algorithms', methods=['GET'])
def get_available_algorithms():
    pass

@app.route('/data-analysis/', methods=['GET'])
def get_data_analysis():
    pass
```

***Reflections (What I achieved/What I learned):***

I learnt how to implement a basic client-server application with Flask and how to create routes to map URIs to python methods. I was able to take everything I learnt from the research and apply it here. I have the six APIs I want for my program now in place.

***Sources (References for this research)***

<https://flask.palletsprojects.com/en/1.1.x/api/>

***Planning (What's Next)***

Next, I will create a persistence layer to store and retrieve the datasets.

Entry Number: 5

Date: 4<sup>th</sup> of January 2021

Objective: Add persistence layer

### **Description of Work Done**

#### 1. Implemented the data layer

To save and retrieve datasets to memory I will need to create a data layer. This layer will contain the functions to interact with the database. The create, read and delete operations I will need to perform align with the POST, GET, and DELETE HTTP operations I am implementing in my REST APIs.

#### 2. Created mock db

For the purposes of this research project and to focus my attention on developing the APIs, I decided to create a mock database. I created this by implementing a python file that creates a globally available data structure in memory upon start up. The file is called database.py and contains two functions `init_data()` and `create_data()`.

```
def init_data():
    global all_datasets, dataset_keys, all_algorithms
    all_algorithms = {}
    all_datasets = {}
    dataset_keys = {}
    create_data(all_datasets, dataset_keys, all_algorithms)
    if len(os.listdir(DATA_PATH)) != 0:
        files = os.listdir(DATA_PATH)
        for file in files:
            add_to_db(file, all_datasets, dataset_keys)
```

The `init_data()` function creates 3 data structures in memory and passes them to the `create_data()` method. This method adds the actual datasets as a pandas dataframe objects to the one of the dictionaries. A second dictionary contains information about the algorithms available, and a third dictionary stores the id and name of each of the available datasets.

I used the 'sklearn' module's 'datasets' module to add pre-existing datasets to the 'database' dictionary. These datasets have there dependent and independent columns separated so I used numpy to combine these into a single pandas dataframe object.

```
#!/usr/bin/env python
# database.py
# python file to mock persistence for project.
# Have created global objects in order call data from anywhere in project.
import os

from sklearn import datasets
import pandas as pd
import numpy as np

from configuration import DATA_PATH
```



```
def create_data(all_data, dataset_keys, all_algorithms):

    # DATASETS
    diabetes_data = np.c_[datasets.load_diabetes().data, datasets.load_diabetes().target],
    diabetes_cols = np.append(datasets.load_diabetes().feature_names, "diabetes_progression")
    diabetes_df = pd.DataFrame(diabetes_data, columns=diabetes_cols)
    all_data["diabetes"] = diabetes_df
    dataset_keys[1] = "diabetes"

    boston_data = np.c_[datasets.load_boston().data, datasets.load_boston().target]
    boston_cols = np.append(datasets.load_boston().feature_names, 'MEDV')
    boston_df = pd.DataFrame(boston_data, columns=boston_cols)
    all_data["boston_house_prices"] = boston_df
    dataset_keys[2] = "boston_house_prices"

    wine = pd.DataFrame(datasets.load_wine().data, columns=datasets.load_wine().feature_names)
    all_data["wine"] = wine
    dataset_keys[3] = "wine"

    # ALGORITHMS
    all_algorithms[1] = "linear regression"
```

The `init_data()` function also checks if the directory 'data' contains any files. The path to this is set in the variable `DATA_PATH` from a configuration file. If the 'data' directory does contain data it will load each file into a pandas dataframe object and store them in the "database" dictionary and add a new id and name entry to the dictionary storing the available datasets.

```
def add_to_db(filename, all_datasets, dataset_keys):
    csv_df = pd.read_csv(os.path.join(DATA_PATH, filename))
    all_datasets[filename.replace(".csv", "")] = csv_df
    dataset_keys[len(dataset_keys) + 1] = filename.replace(".csv", "")
```

To initialize the data I import the database file into the main application class with the APIs and call the `init_data()` function.

### 3. Persistence module

This module contains all the functions to read, delete and create datasets to and from persistence.

There is the function to create datasets. They are only created if it does not exist already. It saves the csv file to 'data' directory. That way upon restarting the app, it will be read from memory from `database.init_data()` method. After saving the csv file, it is read from file and converted into a pandas dataframe and added to global "database" dictionary.

```
def create_dataset(csv_file):
    try:
        filename = csv_file.filename
        if find_data_by_name(filename.replace(".csv", "")) is None:
            csv_file.save(os.path.join(DATA_PATH, csv_file.filename))
            db.add_to_db(filename, all_datasets=db.all_datasets, dataset_keys=db.dataset_keys)
            return True
        else:
            print("Dataset already exists in persistence")
            return False
    except IOError:
        print("I/O error")
        return False
```

Function to find a dataset:

```
def find_dataset(id, name):
    try:
        if name is not None:
            return find_data_by_name(name)
        if id is not None:
            id = int(id)
            return find_data_by_id(id)
    except Exception as e:
        print("Error while trying process request.", e)
    return e
```

Function to retrieve all the dataset names, ids and hyperlink references:

```
def find_all_datasets_info(uri):
    all_data = []
    for id, name in db.dataset_keys.items():
        data = {}
        data["id"] = id
        data["name"] = name
        data["href"] = uri+f"?name={name}"
        all_data.append(data)
    return {"datasets": all_data}
```

Remove datasets:

```
def remove_csv(id):
    data = find_data_by_id(id)
    if data is not None:
        delete_data_by_id(id)
    else:
        return None
```

```
def delete_data_by_id(id):
    key = db.dataset_keys.get(id)
    del db.all_datasets[key]
```

Find data by name or id:

```
def find_data_by_id(id):
    name = db.dataset_keys.get(id)
    return db.all_datasets.get(name)
```

```
def find_data_by_name(name):
    try:
        return db.all_datasets[name]
    except Exception as e:
        print("Dataset does not exist.", e)
```

Find the id corresponding to dataset name and vice versa:

```
def idbyname(name):
    return list(db.dataset_keys.keys())[list(db.dataset_keys.values()).index(name)]

def namebyid(id):
    return db.dataset_keys.get(id)
```

Once all of the persistence layer was done I was able to extend and develop the APIs to call these functions. This is where I implemented the request and response objects for each path.

#### ***Reflections (What I achieved/What I learned):***

I was able to create the persistence layer of the application, and create all the methods I would need for the REST APIs to POST, GET and DELETE datasets.

I learned how to create globally available objects using the global keyword. I also learned to create a sort of global configuration file. For my use it was to pass the root directory path around to different files in different locations.

#### ***Sources (References for this research)***

<https://docs.python.org/3/faq/programming.html#how-do-i-share-global-variables-across-modules>  
[https://numpy.org/doc/stable/reference/generated/numpy.c\\_.html?highlight=c\\_#numpy.c\\_](https://numpy.org/doc/stable/reference/generated/numpy.c_.html?highlight=c_#numpy.c_)  
<https://numpy.org/doc/stable/reference/generated/numpy.append.html?highlight=append#numpy.append>

#### ***Planning (What's Next)***

Next, I will extend and develop the functions mapped to the URI paths.

Entry Number: 6

Date: 10<sup>th</sup> of January 2021

Objective: Extend and Develop the APIs

#### *Description of Work Done*

- Develop the POST, GET and DELETE datasets APIs

Today I developed the datasets APIs. This is where I implemented the request and response objects. I will go through each function to explain the request and response. Firstly, the imports:

```
from persistence import database

database.init_data()

from flask import Flask, request, Response
from data_processor import data_mining_processor as datamine
from persistence import csv_persistence as cp

import json

app = Flask(__name__)
```

This is the dataset\_resource.py file. It is the main file as it contains the WSGI instance used to route the requests. First, I am import the database file and call the init\_data() function immediately. This will setup the global variables before any other imports are called. Then I import the Flask class to instantiate the WSGI instance, and the request and Response classes that I researched earlier. I also of course need the files I created to call CRUD operations and the data\_mining\_processor file which is still empty. I use the json module to create json objects for the response objects.

The first function is a generic exception\_resonse() function. I use to create and return an error response when there has been an exception during execution of a request. It has default values for all parameters except error which is generally the Exception error.

```
def exception_response(error, message='Error processing request.', status=400, href=None):
    if href is None:
        return Response(response=json.dumps({'message': message, 'error': error, 'status': status}), status=400,
                        mimetype='application/json')
    elif href is not None:
        return Response(json=json.dumps({'message': message, 'error': error, 'status': status, 'href': href}),
                        status=400,
                        mimetype='application/json')
```

The second is create\_dataset() function.

Atop I use the route() decorator that specifies the path in URI to map the function to. This function accepts csv files, so I firstly ensure the mixed media type, or mimetype, is 'text/csv' else I send an error response using the generic exception response handler with an appropriate message and 400 status code. I then use the request.files() function to retrieve the csv file from the request. This is sent into the csv\_persistence.create\_dataset() function. If this returns true, then I create a response object with json object containing a message; I set the status to 201, created; add the Location header of the created dataset; and the dataset name. If False was returned, then a 303 response is returned as the dataset already exists. This response contains the Location header to already existing dataset.

```

@app.route('/datasets', methods=['POST'])
def create_dataset():
    try:
        if request.accept_mimetypes['text/csv']:
            csv = request.files['file']
            if cp.create_dataset(csv) is True:
                id = len(cp.find_number_of_dataset())
                message = "Resource created successfully."
                response = Response(response=json.dumps({'message': message}),
                                    status=201, mimetype='application/json',
                                    headers={'Location': '/datasets?id=' + str(id)})
                response.headers['Dataset'] = cp.find_data_by_id(id)
                return response
            else:
                message = "Resource could not be created as it already exists."
                id = cp.idbyname(csv.filename.replace(".csv", ""))
                body = json.dumps(
                    {'message': message, 'dataset': csv.filename.replace(".csv", ""),
                     'href': f'<{request.url}?id={id}'})
                response = Response(response=body, status=303, mimetype='application/json')
                response.headers["Location"] = '/datasets?id=' + str(id)
                return response
        else:
            message = "Resource could not be created. " \
                "Check the CSV file is in the correct format and " \
                "is attached in 'form-data' parameter. " \
                "Ensure the accept header is 'text/csv'."
            error = "Request Error"
            return exception_response(error=error, message=message)
    except Exception as e:
        print("Error while trying process request.", e)
        return exception_response(e)

```

The third function is the `get_dataset()` function. This shows the use of query parameters. In this method I use the `request.args.get()` method to retrieve the query parameters I expect 'id' or 'name'. I then try to retrieve the dataset by either of these. If a dataset is returned, its returned as dataframe object. I use the pandas dataframe function `.to_json()` and set the orient parameter as 'table'. This turns the dataframe into a json object with each instance turned into an individual json object. I also use the 'Link' header to paginate the results. This header is used to indicate any links that are related the current resource being requested.

If the `request.args` are empty I forward the request to the `get_all_datasets()` method. This method returns all the information about the available datasets such as the id, name and href of the dataset.

I also implemented the `delete_dataset()` function that follows the same route decorator and request and response pattern as the above functions.

#### *Reflections (What I achieved/What I learned):*

From my research previously I have been able to successfully implment Flasks request and response objects and send appropriate responses with the correct status codes and headers. I also learned

about the pandas function `to_json()` and its `orient` option which is very useful for application/json responses.

#### *Sources (References for this research)*

Same resources used for the research above: the Flask docs.

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\\_json.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html)

#### *Planning (What's Next)*

Next, I will implement the data mining algorithm endpoints and processing module.

Entry Number: 7

Date: 4<sup>th</sup> of January 2021

Objective: Implement Data Mining Algorithm

### Description of Work Done

- Extended the REST endpoints
  - o Get available algorithms
  - o Get data analysis
- Add data processing – linear regression algorithm.

These two endpoints were much the same as the previous in terms of using the request and response objects.

I chose to implement a simple linear regression algorithm for the purpose of this project to demonstrate a full end to end flow of the initial project idea.

The `get_data_analysis()` function is mapped to path  `'/data-analysis'` and takes a number of query parameters to configure the linear regression algorithm. We need the dataset name, the algorithm name, the list of independent variables and the dependent variable. If any of these do not exist in the URI, then a 400 response will be returned.

```
@app.route('/data-analysis/', methods=['GET'])
def get_data_analysis():
    try:
        algorithm_config = {}
        algorithm_name = request.args.get("algorithm")
        dataset_name = request.args.get("dataset")
        algorithm_config["independents"] = request.args.get("independents").split(",")
        algorithm_config["dependent"] = request.args.get("dependent")

        if algorithm_name is None or dataset_name is None or len(algorithm_config.values()) == 0:
            message = 'Query parameters "dataset" and "algorithm" must be present in URI.'
            return exception_response(error="Algorithm execution error.", message=message)
        else:
            dataset = cp.find_data_by_name(dataset_name)
            analysis = datamine.run(dataset, algorithm_config, algorithm_name)
            response = Response(response=str(analysis), status=200, mimetype='text/plain',
                                headers={'Location': '/run-algorithm'})
            return response
    except Exception as e:
        print("Error while trying process request.", e)
        return exception_response(e)
```

If all the correct information is there it is passed to `datamine.run()` function. This function finds the algorithm to run with the provided algorithm configuration and dataset.

```
def run(dataset, algo_config, algorithm_name):
    if algorithm_name.lower().replace(" ", "") == 'linearregression':
        return linearregression(dataset, algo_config["independents"], algo_config["dependent"])
```

If the algorithm name specified is “linear regression” then the linearregression() is called with the configuration. The data mining algorithm is run on the dataset and returned is summary of the statistics.

```
import statsmodels.api as sm

def linearregression(dataset, independents, dependent):
    # define the data
    X = dataset[independents]
    y = dataset[dependent]

    model = sm.OLS(y, X).fit()
    predictions = model.predict(X) # make the predictions by the model

    # Print out the statistics
    return model.summary()
```

#### ***Reflections (What I achieved/What I learned):***

I learned the basics of the statsmodels package. This package is a basic machine learning algorithm module.

This processing module is very bare now but could be easily extended. A much richer set of configuration parameters could be set in the query parameter list in connection to a wider set of available algorithms.

#### ***Sources (References for this research)***

<https://www.statsmodels.org/stable/regression.html>

<https://www.statsmodels.org/stable/examples/notebooks/generated/ols.html>

#### ***Planning (What's Next)***

Implement unit tests



Entry Number: 8

Date: 10<sup>th</sup> of January 2021

Objective: Implement Unit Tests

*Description of Work Done*

I wrote 8 Test cases.

I have detailed these in the Test section of the report.

*Reflections (What I achieved/What I learned):*

I learned how to use Flask's inbuilt FlaskClient object imported using the test\_client() method. This is another part of Flask's rich libraries to test applications. It is modelled from Werkzeug test client.

*Sources (References for this research)*

[https://flask.palletsprojects.com/en/1.1.x/api/?highlight=flask%20testing#flask.Flask.test\\_client](https://flask.palletsprojects.com/en/1.1.x/api/?highlight=flask%20testing#flask.Flask.test_client)

## References

All sources are listed under each ***sources*** section in the **Research Journal**.