jobserver Documentation

Release 0.1.0

Mirko Mälicke

Oct 04, 2018

CONTENTS:

Jobserver is a Flask application, which is serving Python functions and scripts over an RESTful server interface.

Each processing instance is called a *Job* and can be freely configured. Before the Jobserver can start the execution of Code, information on the Python function to load and run has to be specified as a Job property. Usually, information about the data to be used needs to be specified as well. Beyond that, a Job instance can take almost any information that is JSON serializable.

The first draft of the application uses MongoDB as a database Backend. The main advantage is, that as a NoSQL system it does not require a predefined data model. That means, it can capture almost any algorithm parameters and algorithm output and save it as is. This saved content is directly usable and searchable. In a relational system we would have to either implement all possible outputs and parameters and therefore normalize them, or store them as JSON strings, which would not be searchable.

That said, you need to install MongoDB locally and run it with default settings. The other possible soultion is to connect to a Cloud MongoDB instance, like MongoDB Atlas. It's not really fast and you have just 512 MB on the free plan.

CHAPTER

ONE

INSTALLATION

Jobserver needs Python 3.5 or higher. Jobserver is only available on Github as of this writing.

1.1 Create an environment

It is highly recommended to use Anaconda for managing your Python installations. A new environment for the enigma project could be created like:

```
conda create -n jobserver python=3.6 ipython numpy pandas flask dnspython
```

Note: It is recommended to install all available dependencies for jobserver via conda, as you will find pandas or numpy installations to be way easier, than using pip only. The jobserver itself is not (yet) available on pypi.

Activate the conda environement (if you build it like shown above):

Linux / macOS:

```
source activate jobserver
```

Windows:

```
activate jobserver
```

1.2 Get the files

Then clone the repository:

```
git clone https://github.com/mmaelicke/jobserver.git
cd jobserver
pip install -r requirements.txt
python setup.py install
```

There is no automatic installation script so far, therefore a few changes are necessary. Above all, the Flask server needs a valid configuration to run. There is a `config.default` file in the `jobserver` subfolder. Basically, this has to be copied and renamed to `config.py` and the jobserver will technically run.

```
cp config.default config.py
```

It assumes to find a MongoDB on localhost, standard port 27017.

1.3 Configuration

The only thing that will not work is the e-mail service. You will have to add valid information into the Config object. The code example below shows the important part of the then created `config.py`.

```
class Config:
    [...]
    MAIL_SERVER = 'smtp.yourserver.com'
    MAIL_PORT = 587
    MAIL_USERNAME = 'username'
    MAIL_PASSWORD = 'password'
    MAIL_DEFAULT_SENDER = 'registration@yourserver.com'
```

In case you are not using a local MongoDB instance, you can use any connection string suported by pymongo. This way you can also connect to cloud MongoDB instances, like MongoDB Atlas. Jobserver can handle the old style driver version (up to 3.4) and new style driver version (from 3.6 on) connection strings.

Danger: In both cases, the connection will contain your password in plain text. Therefore the `config.py` is by default listed in the `.gitignore`. Make sure that the config files is kept private!

In order to switch the server instance, you have to change a line in `jobserver/config.py`.

```
class Config:
    [...]
    MONGO_URI = "mongodb://localhost:27017/enigma"
```

Then, you'll have to replace the connection string with the one served on MongoDB Atlas. It can be found in the connection settings.

Note: In case you are planning to run jobserver as a Google App Engine, you'll need to use the old styled connection strings to reach a MongoDB instance outside of the Gloogle Cloud World. For AWS I couldn't successfully connect so far.

CHAPTER

TWO

APPLICATION WORKFLOW

2.1 Job

I have set some definitions in order to be able to model a proper workflow. We have the actual algorithm, that shall be exposed by Jobserver. These algorithms can be a Python function, Python script or even any kind of command line tool. Here we call these algortihms **scripts**.

In the App, a *script* is run by unit called **Process**. There will be different Process instance, e.g. a Process running a Python function, a Process operating multicore, a Process running a file and capturing the output etc.

In case a user wants to start a *Process*, he is creating a **Job**. One Job is bound to the processing of one single *Process*, so far. But pipelined scripts executed by a single Job instance is planned for the future. Jobs do also manage the *Process* parameters and authentication. This way, the number of Jobs (simultaneously or total, whatever you need) can be limited by User, if needed. Once a *Process* has finished, the Job will also include the output. In short: a job collects all the information needed to make a algorithm model run reproducible, while not exposing any detail about the algorithm itself. That means Jobs can public, even for restricted code.

2.1.1 Binding data to a Job

The Jobserver implements different types of data that are handled in different ways. This does not model the type of the actual data, like time series, areal photo and so on, but the way the data is stored.

- **DataFile [type='datafile']** are files that are actually stored on the filesystem. The User can set a DataFile by name that is already there or he has just uploaded. In a productive version, this could be used for example for public files.
- **Data on MongoDB [type='mongodb']** is a data model that represents data, which is stored in the MongoDB. It can be any data that is JSON serializable. It is further described by a `datatype` property, which handles the conversion into the correct Python data structure, before being passed to the script.
- **Data on the Fly [type='raw_data']** is defined as a Job property. It can be of any JSON serializable type. This is usually used in case the data is highly specific to the script, or the user is in development process.

2.1.2 Binding a Process to a Job

Any algorithm that is run on the server is represented by a *Process*. Jobserver offers different Processes, not all are tested so far.

- **Process [type='function']** as the base class of all Processes, that will run an importable function with given parameter and pre-loaded data.

- **FileProcess [type='file']** will behave much like the base class, but run an executable file from the file system. All requirements for this file have to be matched. The STDOUT and STDERR will be cached and bound to the Job as result. The *FileProcess* is not limited to Python files.
- **EvalProcess [type='eval']** takes raw Python code as input and will run that code with the bound data pre-loaded. **Note**, this Eval will have access to the server and should be limited to superusers.

As of this writing, only the base *Process* is tested.

2.2 DataFile

Note: DataFile are development only. They should not be implemented as is into a productive environment. Beforehand, they would need to be bound on user level and some kind of permission model.

A DataFile unit represents an actual file in the data subfolder of Jobserver. Only files of registered file extensions will be recognized as a DataFile. The model can return meta data about one or all files, read the content of the file and pass it to a Job instance, upload new files or delete existing files. It is controlled by the *datafile* and *datafiles* endpoints. One scenario of implementing DataFiles into a productive system could be temporary files allowing the users to upload data and test it, before passing it to the database. If used this way, you'll have to implement a control on the upload and size of this folder on the server side, as Jobserver will just save anything of allowed file extension the user uploads.

CHAPTER

THREE

USER GUIDE

3.1 Using a RESTful API

3.1.1 Sending a request

It is easiest to test the API with some kind of application. Possible applications are curl, Postman or the requests Python module. If you want to sent for example an PUT request to the server containing the data `{'foo':'bar'}`, you could do this like:

```
curl -XPUT 'http://localhost:5000/job/dk4me73jw6r'  -H 'Content-Type: application/json
↪' -d '{"foo": "bar"}'
```

or

```
import requests
payload = dict(foo='bar')
requests.put('http://localhost:5000/job/dk4me73jw6r', json=payload).contents
```

Both will return something similar to:

```
{
  "_id": "5b864f7172f30c24dda84e15",
  "created": "2018-08-29 07:46:57.133147",
  "finished": "None",
  "foo": "42",
  "result": "None",
  "started": "None"
}
```

Postman is a GUI application available as a standalone app or as a Chrome app. It's really straightforward and works graphically. Above that you are able to store requests and set up test scenarios. Therefore using Postman is highly recommended.

3.1.2 Request methods

The general meaning of each of the four common HTTP methods are shown below in the table.

method	meaning
GET	request a resource from the server. Will return a 404 if the resource cannot be found.
POST	Sent with new data to update an existing resource.
PUT	Sent with new data (may be optional for some resources) to create a new resource.
DELETE	Delete the requested resource.

3.2 Managing DataFiles

Note: The DataFiles Model and the corresponding URL endpoint are for development only. This would not be suitable to be integrated into a productive environment.

A DataFile is a model subclass of a more general Data model. It can be identified by its name, which is the filename including the file extension. That means `timeseries.csv` and `timeseries.dat` are two different DataFiles. The file extensions are not case sensitive.

3.2.1 Endpoints

The application exposes two endpoints for managing datafiles:

- [GET, POST, PUT, DELETE] `/datafile/<name>`, where `<name>` should be replaced with the actual file name
- [GET, DELETE] `/datafiles`

datafile

The datafile endpoint has four methods:

- [GET] `/datafile/name` will return metadata about the requested file. This method also accepts a URL parameter `format`. There are four valid values for `format`, 'csv','json', 'html' and 'txt'. If one of these is set, the file content will be returned in the specified format. That will **not** be of `Content-Type: application/json`.
- [PUT, POST] `/datafile/name` will upload a new file and store it as the given *name* in the data folder. The file has to be submitted using `enctype='multipart/form-data'` and be identified by the name 'file'. In Postman this can be set the Body section of type form-data and key file. If a HTML upload is used, it has to look like:

```
<form action="url/to/endpoint" method="post" enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit">
</form>
```

- [DELETE] `/datafie/name` will delete the file from the data folder, without asking again.

datafiles

The datafiles endpoints has two methods:

- [GET] `/datafiles` will return a list of metadata about all DataFiles found in the data folder.
- [DELETE] `/datafiles` will delete **all** DataFiles from the data folder.

3.3 Managing Jobs

The Job API is the core API and most important API of Jobserver.

3.3.1 Endpoints

The application exposes several endpoints to accomplish different tasks. The Job management uses the endpoint `/job/:id`, where *:id* is replaced by the actual ID of the Job to be managed. Additionally, there are several extra endpoints related to jobs:

- [GET; POST; PUT; DELETE] `/job/:id` is used to view edit, create and delete Jobs of specified `:id`.
- [PUT] `/job`, will create a job with autogenerated id
- [GET; POST; PUT] `/job:id/run` will start the Job of *:id*
- [GET, DELETE] `/jobs` will return/delete a list of all Jobs

3.4 Basic example

For brevity, we will skip the `-H 'Content-Type: application/json'` flag and the host `http://localhost:5000` from the examples and just give the method and endpoint. If you want to run the example, you'll have to add the flags to curl. An example workflow could look like:

1. Put a new Job. A Job needs at least a script specified and in most cases a data bound to it. Unless a script is called, that does not need data input. Here, to demonstrate the flexibility of the Job API, we will create a Job with a specified script, but omit the data to define it later on. As long as we wish to load a script-function from the `jobserver.scripts` submodule and no args and kwargs are specified, we can also use the script shortcut of just defining the name of the function.

```
curl -XPUT /job -d '{"script_name": "summary"}'
```

output:


```
{
    "_id": "5b9011469eb82b0d84ca212f",
    "created": "2018-09-05 17:24:22.607225",
    "finished": "None",
    "result": "None",
    "script_name": "summary",
    "started": "None"
}
```


The API will most likely return another ID. Then, you'll have to replace the ID used in these examples with your ID.

2. Before we can start the job, we have to specify the DataFile to use. This could have been done in the first step as well. To update a job, we can use the `POST` endpoint, sending the new data.

```
curl -XPOST /job/5b9011469eb82b0d84ca212f -d '{"datafile": "timeseries.csv"}'
```


```
{
    "created": "2018-09-05 17:24:22.607000",
    "started": "None",
    "finished": "None",
    "result": "None",
    "script_name": "summary",
    "edited": "2018-09-05 17:25:17.421061",
    "datafile": "timeseries.csv",
    "_id": "5b9011469eb82b0d84ca212f"
}
```

3. Start the Job.

```
curl -XGET /job/5b9011469eb82b0d84ca212f/run
```

which yields:


```
{
    "_id": "5b9011469eb82b0d84ca212f",
    "created": "2018-09-05 17:24:22.607000",
    "data": {
        "name": "timeseries.csv",
        "path": "/home/mirko/Dropbox/python/REST-nigma/rest_nigma/data/timeseries.csv
↪",
        "size": "566 KB",
        "type": "datafile"
    },
    "datafile": "timeseries.csv",
    "edited": "2018-09-05 17:25:58.674266",
    "finished": "None",
    "result": "None",
    "script": {
        "args": "[]",
        "kwargs": {},
        "name": "summary"
    },
    "script_name": "summary",
    "started": "2018-09-05 17:25:58.673719"
}
```


As you can see, the Job did set the module, and process names and arguments. The *started* indicates, that the process is now started without error. However, the *result* and *finished* are still empty, because the Job status was returned directly after starting the job. If we need the status, we have to request to `GET` the Job instance again.

Note: It is absolutely possible that the result is already there, in the response of the start job request. The reason is, that the Job is started asynchronously and the Process fills in the finish time and the result. In case the Process finishes really quickly, it will have updated the Job object in the database, before the Job could infer and fill in all the information about the data file.

---

4. Request the Job:

```
curl -XGET /job/5b9011469eb82b0d84ca212f
```

output:


```
{
    "created": "2018-09-05 17:24:22.607000",
    "started": "2018-09-05 17:25:58.673000",
    "finished": "2018-09-05 17:25:59.756000",
    "result": {
        "value": {
            "count": "12000.0",
            "mean": "40.10492551050158",
            "std": "12.660818317676181",
            "min": "6.746015545337676",
            "25%": "31.012809495045943",
            "50%": "38.82110180726302",
```


(continues on next page)

(continued from previous page)


```
            "75%": "47.78757677274963",
            "max": "104.02590583156969"
        }
    },
    "script_name": "summary",
    "datafile": "timeseries.csv",
    "edited": "2018-09-05 17:25:59.757000",
    "data": {
        "type": "datafile",
        "path": "/home/mirko/Dropbox/python/REST-nigma/rest_nigma/data/timeseries.csv
↪",
        "name": "timeseries.csv",
        "size": "566 KB"
    },
    "script": {
        "name": "summary",
        "args": "[]",
        "kwargs": {}
    },
    "time_sec": "1.083251",
    "_id": "5b9011469eb82b0d84ca212f"
}
```


Now, there is the result. This dummy script, producing just a few statistical numbers on a timeseries created from a gamma distribution took over one second. The reason is a random time delay between 1 and 5 seconds.