



OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

INF

FAKULTÄT FÜR
INFORMATIK

Documentation VLBA II – System Architectures

Airflow & Taverna

Nikhil Mandya Parashivamurthy
Student ID :220641
Magdeburg, 27.06.2018.

Table of contents

Introduction	2
2 Workflows	2
3 Airflow.	2
4 Taverna	13
5 References	21

Introduction

2 Workflows

Workflow management is the coordination of tasks that make up the work an organisation does. By 'workflow' we mean a sequence of tasks that are part of some larger task, and is sometimes synonymous with 'business process'. The purpose of a workflow is to achieve some result, and the purpose of workflow management is to achieve better results according to some set of goals.

3 Airflow.

Airflow is a platform to programmatically author, schedule and monitor workflows.

Airflow converts the work into simple and well organised by dividing it to smaller chunks(not always) task .It uses the Directed Acyclic Graphs (DAG's) for managing the arrays of task without violating the dependencies constraints(like one task as to complete before the other and the new task depends on the result of the previous task).

- **Dynamic** - The pipeline is written in python code.It allows for dynamic pipeline creation
- **Extensible** - The library can be easily extend to our needs by defining the operators, plugins and executors.
- **Elegant** - Airflow pipelines are lean and explicit.It uses Jinja template to define some of the information in the dag file
- **Scalable** - Airflow has a modular architecture and uses a message queue to manage multiple workers.

3.1 Basics of airflow.

DAG's - Directed acyclic graph -its is a collection of all the task that you want to run and define the relation or dependencies of the task on one another.

- All the DAG's are defined as the python script which is placed in the dag folder the .cfg file should be configured to the path.
- Whenever a new DAG's are placed in the folder all the DAG's are picked up by the airflow.

Operators - All the single task are defined by the operator (bashOperator,Python operator...) Operators determine what gets done.

- Task - All the task are uniquely identified by task_id.
- Task Instance - A task executed at a time is called Task Instance.

Scheduling the Task - All the task are scheduled to run on the on the particular time and particular number of time.Schedule_interval parameter is used to set the frequency the time of the particular task.

Executor - Once the dags are place in the db and picked up by the scheduler it sends the task to the executor to run the task ,executor will be sitting idle waiting for the task to be assign by the scheduler

- Sequential - The task are test driven it runs one after the other no parallelization.
- Local - It is similar to the sequential, but it can be parallelized depends on the system on which it resides.
- Celery - It is a open source distributed task engine based on message queue the workers(executor) may be remote ,multiple servers (processors).Message queues are handled by RabbitMQ or Redis Server.

3.2 why to use airflow?

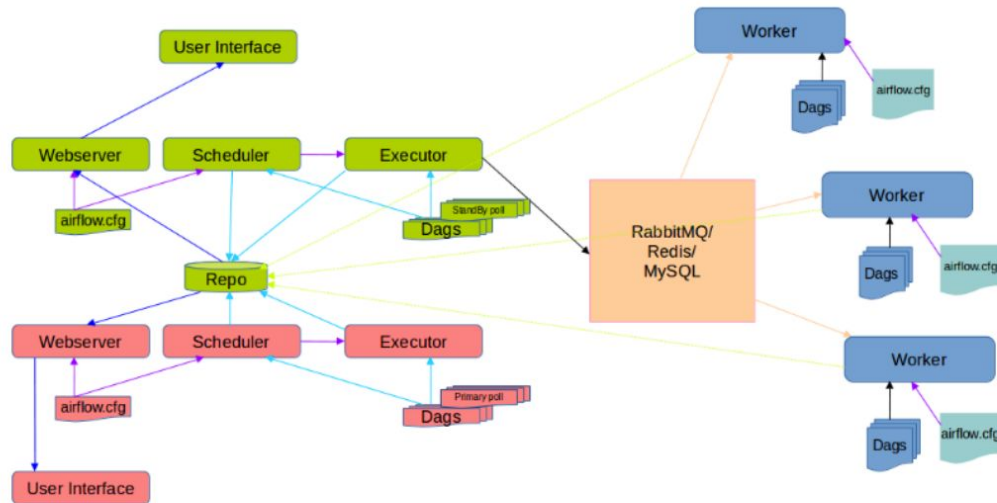
- Pipelines are configured via code making the pipelines dynamic
- A GUI representation of the DAG instances and Task Instances along with the metrics.
- Scalability:The task can be distributed among multiple workers ,the task that are independent of one other ,in case if a task fails the other (independent) task continue to run without stopping the process.
- Easy to add new Tasks
- Rich UI for monitoring purpose, better statistical results are presented in graph.
- Ability to re-run the task that have been failed, and the better log are maintained in DB.
- Easy to edit task and add variable to it.

3.3 Architecture of Airflow.

The basic components containing the Airflow.

- **Configuration file** - Its is the heart of the airflow where everything is controlled from here like DB connection, worker configuration ,web-server port and much more.
- **Metadata database** - Mysql or Postgres DB is used in Airflow, it is used to maintain all the metadata about the DAG's ,DAG's run, Variable, and store info about the DAG's how long it took to run, and which task took more time to run all info is stored in DB.
- **DAG's** - (Directed Acyclic Graph) These are the logical unit which contains the task and there dependencies it is acyclic means there no interdependent tasks. These are the actual tasks need to be picked by scheduler and send them to the worker.
- **Scheduler** - It triggers the DAG instances and load all the task according to scheduled interval and all the schedule interval are independent means if one scheduled task fails it doesn't affect the other task.
- **Broker** - The RabbitMQ or Redis server acts as broker to manage the multiple worker by passing message queue between the workers.
- **Worker Nodes** - Worker nodes are the one where the task get executed and return the result of the task.

- **Web-Server** - Web-server is used to provide the user with the rich UI on the browser ,it provide many control to the user such as configuration of the DB connection and provide user with the ability to run the task from web-server and see the result from the browser.



All the DAG's are picked up by the scheduler and places the metadata to the DB(mysql or postgresql) . Worker will be ready to pick the task, until scheduler tell the worker to task the worker will be sitting idle, whenever a schedule sends the task to do and writes the status to the DB.

The Web-server uses the data stored in the repo(DB) and provides a rich UI and controls to the user like connection and configurations.

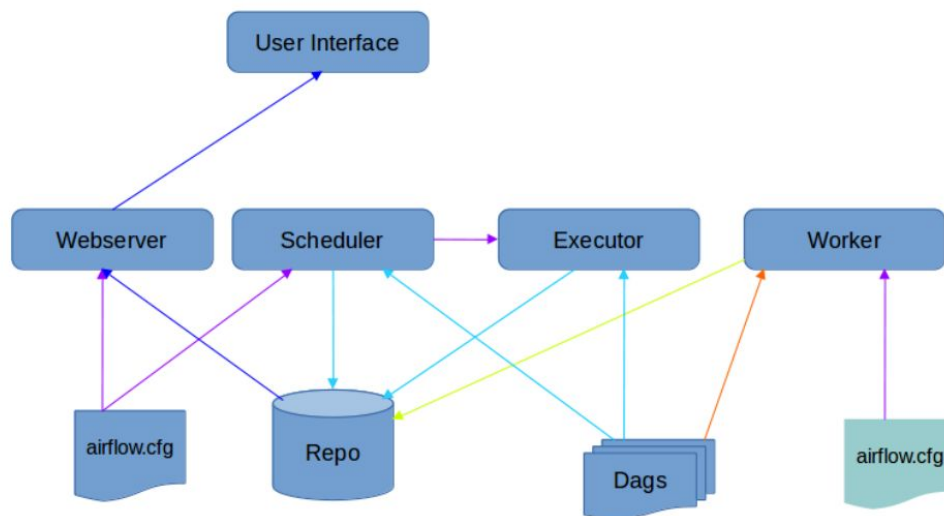
3.4 Deployment of Airflow.

There are two options to put airflow to up and running .

1. **Standalone Mode.**
2. **Distributed Mode.**

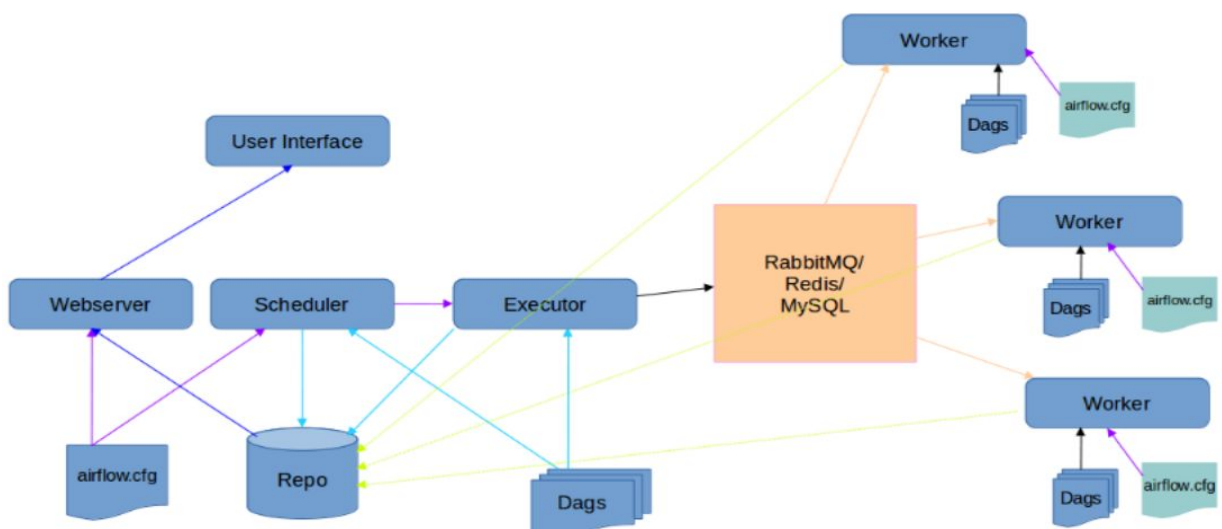
3.4.1 Standalone Mode:

- **Configuration File(.cfg)** - It contains the information about where to pick the DAG's from the folder ,which Executor to use, the DB connection, which port to start the WebServer.
- **Metadata Repository** - Mysql or Postgresql are used typically, all the info about DAG's are stored in it. The stats like how long it took to complete the task, status of each tasks.
- **Web-server** - Render the beautiful UI that contains all the DAG's, and provide controls to run the DAG's.
- **Executor** - This runs the tasks that are put up into queue by the scheduler and stores the results into DB.



3.4.2 Distributed Mode:

- **Redis-server**- Redis-server is the distributed messaging services that the celery Executor uses to put the task to pool. The worker take up this task . Broker URL that is used by redis- server for the celery executor and workers to talk.
- **Executor** - The executor we use is the Celery executor ,the Redis-server for message passing is configured.
- **Worker** - The worker running on multiple node reads the broker url for the task and work on it, write's the result back to the DB.



The business model must be separate from the airflow final implementation model, most of people including me first tried out the model by hand(executing and checking) and then integrating to airflow as final step.

3.5 Installation Procedure.

Note: All the installation process showed here for Ubuntu 16.04 LTS.

1)Airflow Installation.

```
$pip install apache-airflow
```

2)Postgresql DataBase.

```
$sudo apt-get install postgresql
```

connect to the postgresql and create a new db called “airflow”

```
$psql
```

```
nikhil=# create database airflow
```

```
nikhil=#\q
```

3)Psycopg2

```
$pip install psycopg2
```

it is the most popular PostgreSQL adapter for the Python programming language.

4)Celery Executor

```
$pip install apache-airflow[celery]
```

5)Redis-server

```
$pip install -U "celery[redis]"
```

6)install textblob and tweepy

```
$pip install textblob
```

```
$pip install tweepy
```

Navigate to the airflow installation folder and change the following in the airflow.cfg file.

- set Executor to Celery Executor

```
Executor = CeleryExecutor
```

- configure the sql

```
sql_alchemy_conn=postgresql+psycopg2://your_postgres_user_name:your_postgres_password@host_name/database_name
```

- configure the Broker to Redis

broker_url=redis+psycpg2://your_postgres_user_name:your_postgres_password@host_name/database_name

- Create a new file in airflow directory called “dag” where all the DAG’s are placed

These are the things to be started before running a task.(run these in separate terminal)

1. airflow initdb
2. airflow scheduler
3. airflow worker
4. airflow webserver --port 8080

Now the airflow server will be started in the port localhost:8080 we can use the web browser to check and run all the DAG’s.

3.5.1 Simple Example

All the DAGs and the tasks are uniquely identified by their id’s sample code.

```
default_args = {
    'owner': 'airflow',
    'start_date': datetime.now(),
    'retries':2
}

dag = DAG(
    'simple_example', default_args=default_args, schedule_interval='@once')
```

‘simple_example’ is the name of the DAG

```
ex="""
echo "hello vlba"
"""
t1=BashOperator(
    task_id='bash_operator',
    bash_command=ex,
    dag=dag)
```

The above picture is the bash_operator it prints “hello vlba” on the screen.

```
t2=PythonOperator(
    task_id='python_operator',
    python_callable=examplePrint,
    provide_context=True,
    dag=dag)
```

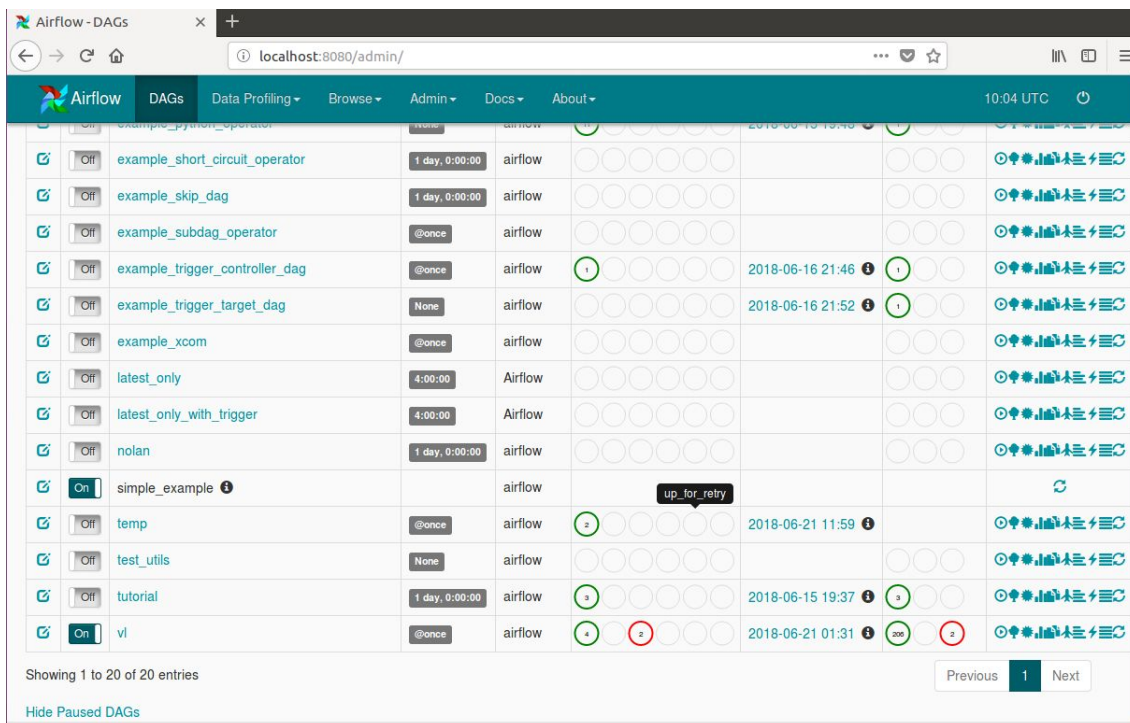
```
def examplePrint(**kwargs):
    print("Simple dag(task)")
    print("these is dependent on previous task")
```

It is a Python_operator it call the python function defined within the DAG file (examplePrint(**kwargs))



finally the dependencies are set using the upstream command.(means t1 should complete first).

```
t2.set_upstream(t1)
```

The home page of the airflow will contains all the DAG's and there status ,it looks like

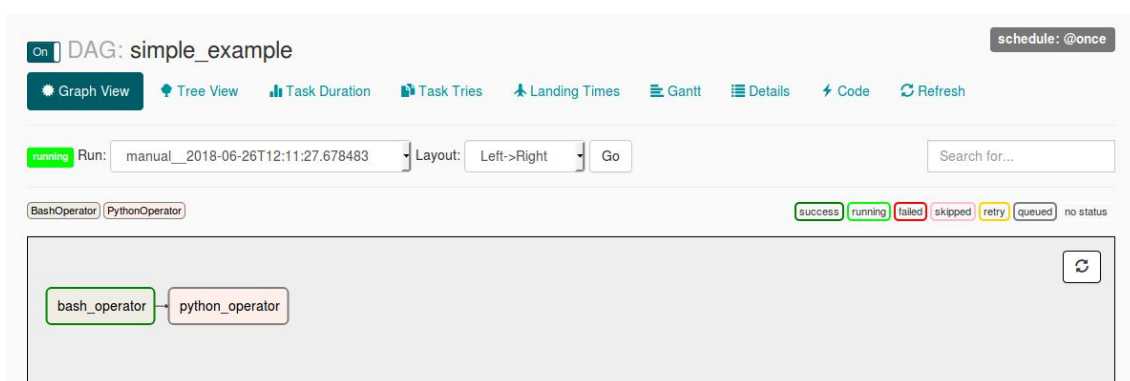


DAG	Frequency	Next Run	Status	Last Run	Buttons
example_short_circuit_operator	1 day, 0:00:00		Off		Play, Stop, Refresh, etc.
example_skip_dag	1 day, 0:00:00		Off		Play, Stop, Refresh, etc.
example_subdag_operator	@once		Off		Play, Stop, Refresh, etc.
example_trigger_controller_dag	@once	2018-06-16 21:46	1		Play, Stop, Refresh, etc.
example_trigger_target_dag	None	2018-06-16 21:52	1		Play, Stop, Refresh, etc.
example_xcom	@once		Off		Play, Stop, Refresh, etc.
latest_only	4:00:00		Off		Play, Stop, Refresh, etc.
latest_only_with_trigger	4:00:00		Off		Play, Stop, Refresh, etc.
nolan	1 day, 0:00:00		Off		Play, Stop, Refresh, etc.
simple_example	@once	2018-06-21 11:59	2		Play, Stop, Refresh, etc.
temp	@once		Off		Play, Stop, Refresh, etc.
test_utils	None		Off		Play, Stop, Refresh, etc.
tutorial	1 day, 0:00:00	2018-06-15 19:37	3		Play, Stop, Refresh, etc.
vi	@once	2018-06-21 01:31	200		Play, Stop, Refresh, etc.

To run a first turn on the DAG using the radio button on the left when clicked it should turn to green and “on” text will be displayed, and click the play alike button on the right (displayed on the picture ) it will prompt a pop-up click run ,now the DAG is picked by the scheduler and should be running.to view the task and there status of the current DAG press() .



simple_example	@once	airflow	2	2018-06-21 12:04	2	Buttons
----------------	-------	---------	---	------------------	---	---------



The output will be displayed in the log of that particular task to view the logs click on task(python_operator) and click on view logs Button.

The simple example includes two Operator

- 1) bash_operator
- 2) python_operator

The both operator prints the “hello on the screen” the python_operator is dependent on the bash_opertaor that means the python_operator can’t start until the bash_operator task is complete.

3.5.2 Another Example

In these example (DAG name VL) the operators that are

- 1)bash_operator.
- 2)python_operator.

In the second example

```
t1=BashOperator(
    task_id='write_to_sport',
    bash_command="python3 /home/nikhil/airflow/script/sports.py",
    dag=dag)

t2=BashOperator(
    task_id='write_to_politics',
    bash_command="python3 /home/nikhil/airflow/script/politics.py",
    dag=dag)
```

the task “t1 and t2” uses the bash command to call the python file and execute the files, the python script contains the code to get the data from the twitter using twitter api and writes it to a file.

```
t3=BashOperator(
    task_id="calculate_sentiment_sports",
    bash_command='python3 /home/nikhil/airflow/script/sentisports.py',
    dag=dag)

t4=BashOperator(
    task_id="calculate_sentiment_politics",
    bash_command='python3 /home/nikhil/airflow/script/sentipolitics.py',
    dag=dag)
```

The task “t3 and t4” fetch’s the file written by the task “t1and t2” earlier and calculate the sentiment of those text and write it to the same file.

```
t5=PythonOperator(
    task_id="total_politics",
    python_callable=totalpolitics,
    provide_context=True,
    dag=dag)
t6=PythonOperator(
    task_id="total_sports",
    python_callable=totalsports,
    provide_context=True,
    dag=dag)
```

The task “t5 and t6” are the python operator it call the python function defined within the DAG file that sum up’s all the sentiment and uses the Xcom to push the value.

```
def totalpolitics(**context):
    ti=context['task_instance']
    df=pd.read_csv("/home/nikhil/airflow/script/Politics/politics.csv")
    temp=df['polarity'].sum()
    ti.xcom_push(key="total_politics",value=temp)
    print(polt)
    return polt
def totalsports(**context):
    ti=context['task_instance']
    df=pd.read_csv("/home/nikhil/airflow/script/Sports/sports.csv")
    temp=df['polarity'].sum()
    ti.xcom_push(key="total_politics",value=temp)

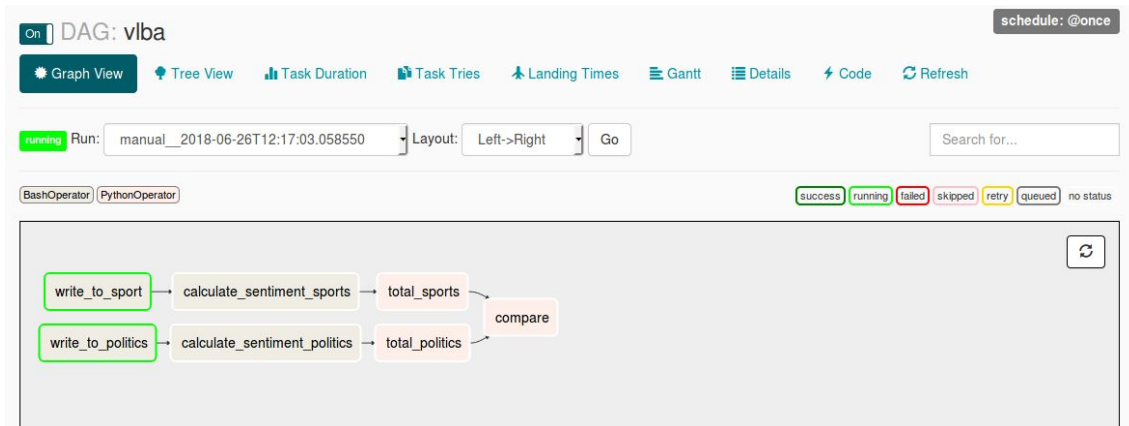
    print(polt)
    return polt
```

The Task “t7” compares sentiment of the two files “sports and politics” and prints which one has the higher in the positivity.

```
t7=PythonOperator(
    task_id="compare",
    python_callable=compare,
    provide_context=True,
    dag=dag)
```

```
def compare(**context):
    ti=context['task_instance']
    sports=ti.xcom_pull(task_ids='total_sports')
    politics=ti.xcom_pull(task_ids='total_politics')
    if(sports>politics):
        print("sports has more positive")
    else:
        print("politics have more positive")
```

To run the second example turn on the DAG's which says "vlba" and click on run button as in first example the DAG should be running.



the output of these workflow will be printed on the log to get there click on the "compare" task and the click on view log's.

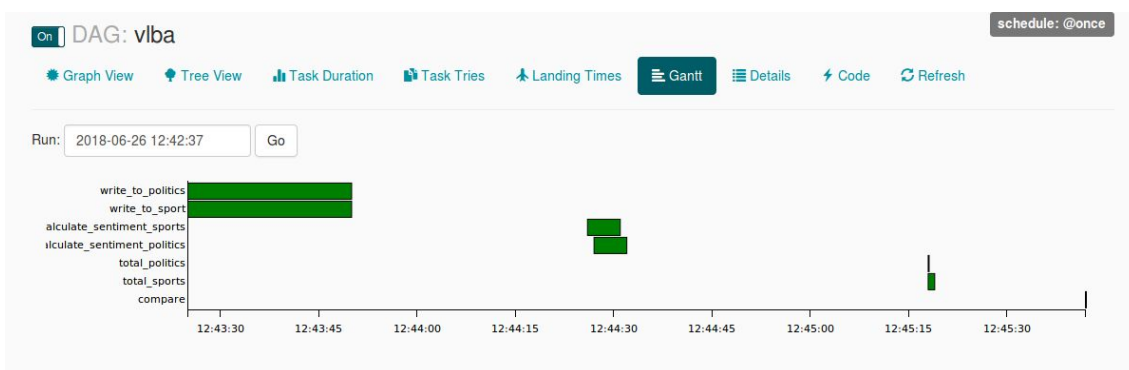
Task Instance: compare 2018-06-26 12:17:03

[Task Instance Details](#) [Rendered Template](#) [Log](#) [XCom](#)

Log

```
[2018-06-26 12:21:53,864] {models.py:167} INFO - Filling up the DagBag from /home/nikhil/airflow/dags/vlba.py
[2018-06-26 12:21:56,284] {base_task_runner.py:112} INFO - Running: ['bash', '-c', 'airflow run vlba compare 2018-06-26T12:17:03.058550 --job
[2018-06-26 12:21:57,694] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:21:57,694] {__init__.py:57} INFO - Using executor CeleryExe
[2018-06-26 12:21:58,272] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:21:58,271] {driver.py:120} INFO - Generating grammar tables
[2018-06-26 12:21:58,404] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:21:58,392] {driver.py:120} INFO - Generating grammar tables
[2018-06-26 12:21:59,469] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:21:59,469] {models.py:167} INFO - Filling up the DagBag fro
[2018-06-26 12:22:01,157] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:22:01,157] {models.py:1126} INFO - Dependencies all met for
[2018-06-26 12:22:01,205] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:22:01,205] {models.py:1126} INFO - Dependencies all met for
[2018-06-26 12:22:01,205] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:22:01,205] {models.py:1318} INFO -
[2018-06-26 12:22:01,205] {base_task_runner.py:95} INFO - Subtask: -----
[2018-06-26 12:22:01,205] {base_task_runner.py:95} INFO - Subtask: Starting attempt 1 of 3
[2018-06-26 12:22:01,205] {base_task_runner.py:95} INFO - Subtask: -----
[2018-06-26 12:22:01,237] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:22:01,237] {models.py:1342} INFO - Executing <Task(PythonOp
[2018-06-26 12:22:01,544] {base_task_runner.py:95} INFO - Subtask: [2018-06-26 12:22:01,544] {python_operator.py:81} INFO - Done. Returned va
[2018-06-26 12:22:01,599] {base_task_runner.py:95} INFO - Subtask: /home/nikhil/.local/lib/python3.5/site-packages/airflow/ti_deps/deps/base_
[2018-06-26 12:22:01,599] {base_task_runner.py:95} INFO - Subtask: for dep_status in self._get_dep_statuses(ti, session, dep_context):
[2018-06-26 12:22:06,914] {jobs.py:2083} INFO - Task exited with return code 0
```

The performance of the task and time take by each task can be viewed in "Gantt" tab.



For these task Xcom is used to push the data from one task to another task ,Xcom is useful when we are running in the distributed executor which helps in passing message(value) from one task

to another, and xcom of a particular task can view by clicking on that task and navigate to Xcom it shows the key ,value of that task.

Task Instance: total_politics

2018-06-26 12:42:37

Task Instance Details

Rendered Template

Log

XCom

XCom

Key	Value
total_politics	0.8509604978354975
return_value	0

4. Taverna

Taverna is an open-source workbench for the design and execution of scientific workflows. Aimed primarily at the life sciences community, its main goal is to make the design and execution of workflows accessible to bioinformaticians who are not necessarily experts in web services and programming.

A Taverna workflow is a linked graph of processors, which represent Web services or other executable components, each of which transforms a set of data inputs into a set of data outputs. These workflows are represented in XML syntax, and executed according to a functional programming model.

Taverna also provides a plug-in architecture so that additional applications, such as secure Web Services, can be populated to it. The design and implementation of workflow systems for scientific purposes has been a subject of considerable research. In this analysis, we consider the entire scientific workflow lifecycle, from service discovery to service composition, workflow execution, and workflow result analysis.

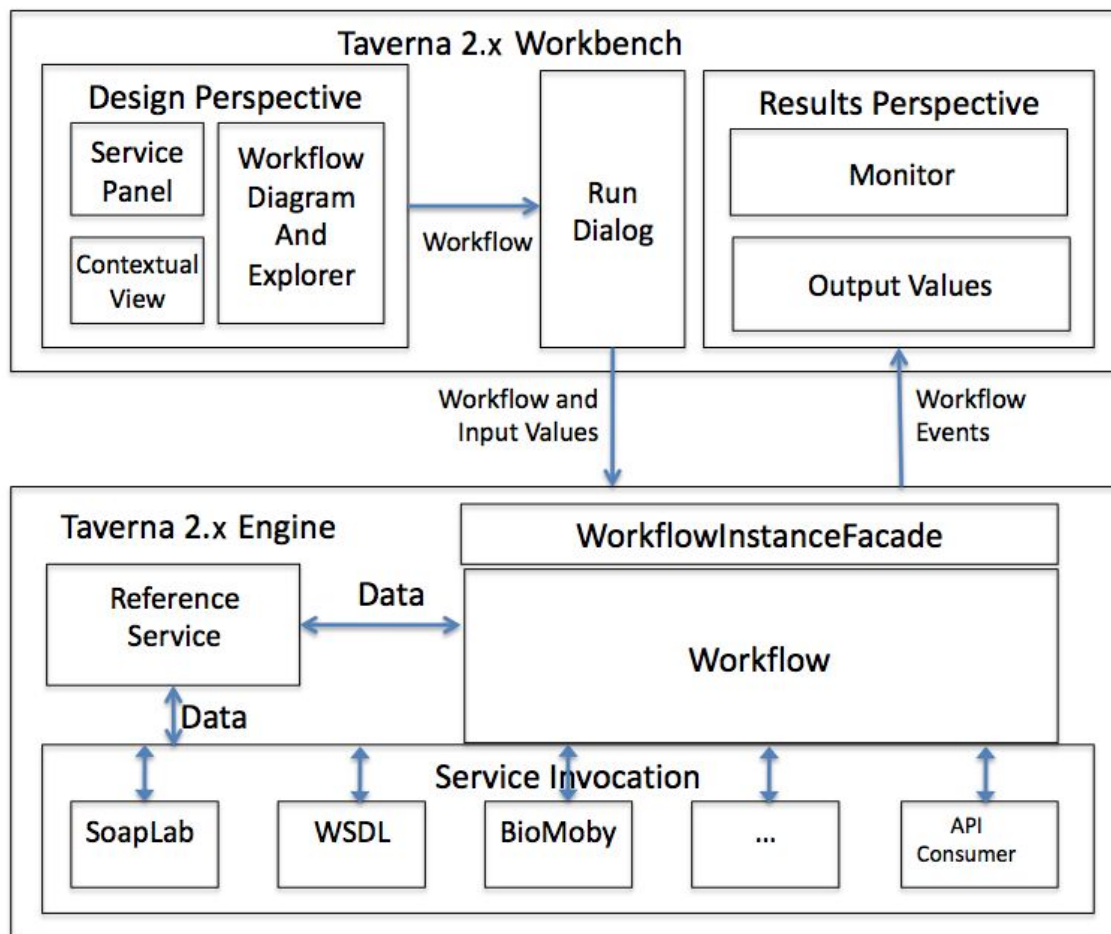
4.1 Types of Services

Taverna uses the java API and exposes methods in that API as a service .

- **Beanshell** - Beanshell is a Java scripting language. A Beanshell service in Taverna enables you to write simple Java code snippets and execute them as part of your workflows. Beanshells in Taverna typically perform data cleaning and processing of the data.
- **BioMart** - A data warehouse and management by the EBI (European Bioinformatics Institute) and the OiCR (Ontario Institute for Cancer Research). Biomart contains genomic, proteomic and any Biomart database if you provide the URL to the BioMart service interface (MartService). It provide a service like you can query for the information from Workflow to the EBI and OiCR.
- **Command Line Tool** - Taverna provides a Command Line Tool is a command line script that give access to run the taverna workflow without the overhead of the GUI. It receives the workflow to execute and inputs as command line parameters and writes the outputs to a folder on a disk.
- **Control Layer** - Taverna adds a control layer to every service in a workflow to provide users with more power to control over how a service is being invoked. The users can set a condition till what times it should run and till the certain criterion has meet. It also allows user to define how a service should handle incoming list items where two or more lists are being passed to the service on two or more different ports. In future, it will also include defining an alternate service of similar/equivalent functionality to replace the original one in a case of failure.

-
- **Control Link** - Enables you to set dependencies between services in a workflow that do not directly share data (i.e. that are not otherwise linked by passing data from one to the other directly or indirectly). A control link allows you to delay the invocation of a service until another has finished.
 - **Nested Workflows** - A workflow within a workflow. In an abstract sense, a nested workflow is just another kind of service that can be added into a workflow, except that instead of it being a black box, it is a white box so you can see what is happening inside. It is often the case that a workflow designed for one purpose can be used again for other experiments and can be imported and added to another workflow. Nested workflows can be added to a workflow by dragging the nested workflow service description into the Workflow Diagram.
 - **Rest** - Taverna provides the Rest service to call Webservice there are four types of Rest calls.
 - GET - to get the resource.
 - POST - to make a new resource.
 - PUT - to update a resource.
 - DELETE - to remove a resource.
 - **Rshell** - A service that enables analyses using the R statistical package to be incorporated into the workflow. The example used here requires the R package to be installed in the local(taverna installed) machine.
 - **Soaplab** - A tool for wrapping command-line and legacy programs automatically as Web services. Soaplab is particularly designed for people who prefer to program in perl or python.
 - **Spreadsheet import service** - A service that provide user to import the Excel sheet or CSV directly into the workflow space and provide some of the functionality to edit the CSV or Excel
 - select which rows and columns to output
 - how to deal with empty cells
 - whether to include the header row, and
 - what name to use for the data from a given column

4.2 Taverna Architecture



If the user decides to run the workflow, then they specify values for the input ports of the workflow within the Run Dialog that pops up if the workflow has any inputs defined.

In Taverna there is no separate enactment system. Instead, a **WorkflowInstanceFacade** is created to represent a run of the workflow. The enactment of the workflow is performed by the workflow itself; similarly the invocation of external services is done by the activities (service instances) within the workflow. The information about a particular workflow run is identified by the **WorkflowInstanceFacade** being used for that run.

Rather than transferring data around within the workflow, Taverna uses a **Reference Service** that proxies the actual data by references to them. The workflow run thus passes around the references. When the values are needed, for example to be used in a call of a service, the data is obtained from the **Reference Service** and rendered.

The events that happen during the execution of the workflow run are listened to by a monitor within the **Results Perspective** of Taverna Workbench. The final results can be browsed.

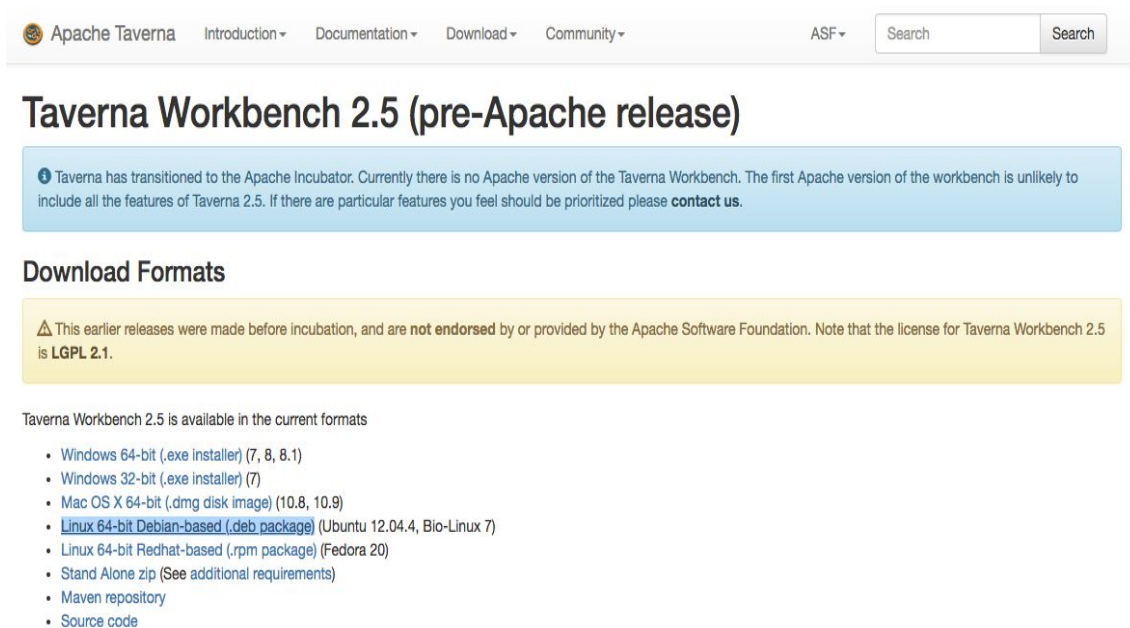
Taverna Engine also has the infrastructure to generate provenance information. Provenance data contains information about the workflow run, such as date and time of the run, intermediate values generated by services in the workflow during the run, as well as the final results.

Provenance data can be saved in a database, making it persistent over the Workbench restarts. However, this also makes the workflow run slower as the Workbench needs to talk to the database all the time to write data to it.

Alternatively, provenance data can be saved in memory, but it will disappear when you close Taverna unless you explicitly tell Taverna to save it to the database. Using the in-memory provenance speeds up the workflow run and is recommended during the design and testing of your workflows. Once you are satisfied that your workflow is doing what you want, it is recommended to switch to database if you want your provenance data saved. Switching between the in-memory and database can be done from the Preferences in the Workbench.

4.3 Installation Procedure.

Note: All the installation process showed here for Ubuntu 16.04 LTS.



Apache Taverna Introduction Documentation Download Community ASF Search Search

Taverna Workbench 2.5 (pre-Apache release)

Taverna has transitioned to the Apache Incubator. Currently there is no Apache version of the Taverna Workbench. The first Apache version of the workbench is unlikely to include all the features of Taverna 2.5. If there are particular features you feel should be prioritized please **contact us**.

Download Formats

This earlier releases were made before incubation, and are **not endorsed** by or provided by the Apache Software Foundation. Note that the license for Taverna Workbench 2.5 is **LGPL 2.1**.

Taverna Workbench 2.5 is available in the current formats

- Windows 64-bit (.exe installer) (7, 8, 8.1)
- Windows 32-bit (.exe installer) (7)
- Mac OS X 64-bit (.dmg disk image) (10.8, 10.9)
- [Linux 64-bit Debian-based \(.deb package\)](#) (Ubuntu 12.04.4, Bio-Linux 7)
- Linux 64-bit Redhat-based (.rpm package) (Fedora 20)
- Stand Alone zip (See additional requirements)
- Maven repository
- Source code

Download the Linux 64-bit Debian-based (.deb package) and install in the local machine, and install Rstudio from the website <https://www.rstudio.com/products/rstudio/download/>

The dependencies that are required are R shell and Rserver, sentimentr. To install R shell in ubuntu open the Terminal and run

```
$sudo apt-get install r-base
```

and run the r shell from terminal

```
$sudo -i R
```

Install the required packages for the using the command in the R shell.

```
install.packages("Rserve")
```

```
install.packages("sentimentr")
```

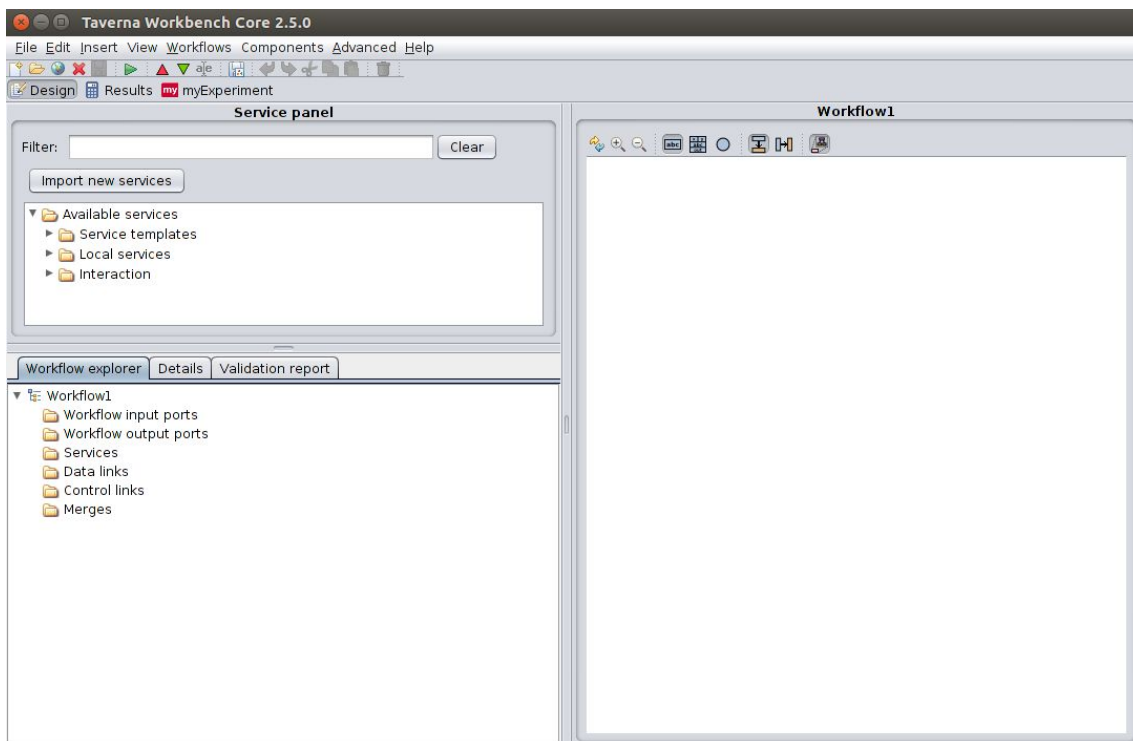
```
install.packages("jsonlite")
```

open Rstudio and run the following command to run the Rserve so that it can listen to the incoming request from the Taverna.

```
>library(Rserve)
```

```
>Rserve(args="--no-save")
```

The Taverna workbench look like the figure shown below

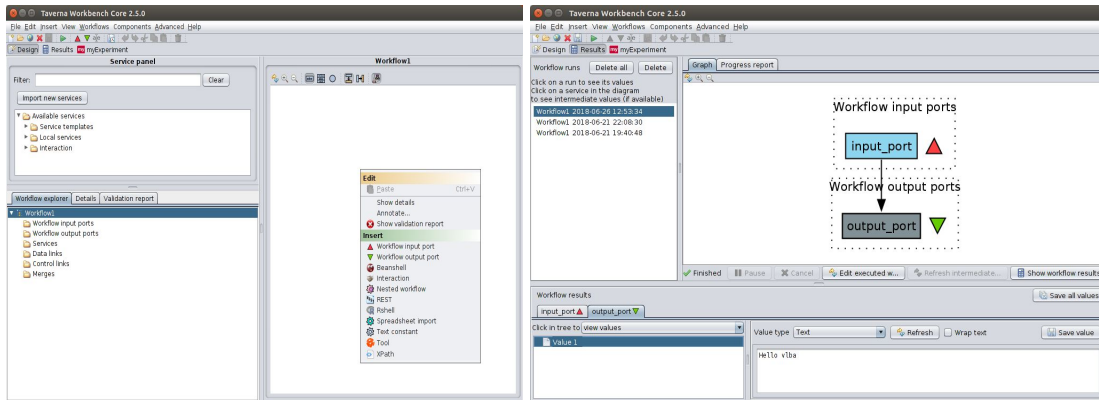


The service on the top left I used to import the services provided by the Taverna and directly use here, the middle part has the “workflow explorer”, “Details” and “validation report” it contains all the services and there options to the user. The right workspace displays the workflow diagram and the dependencies.

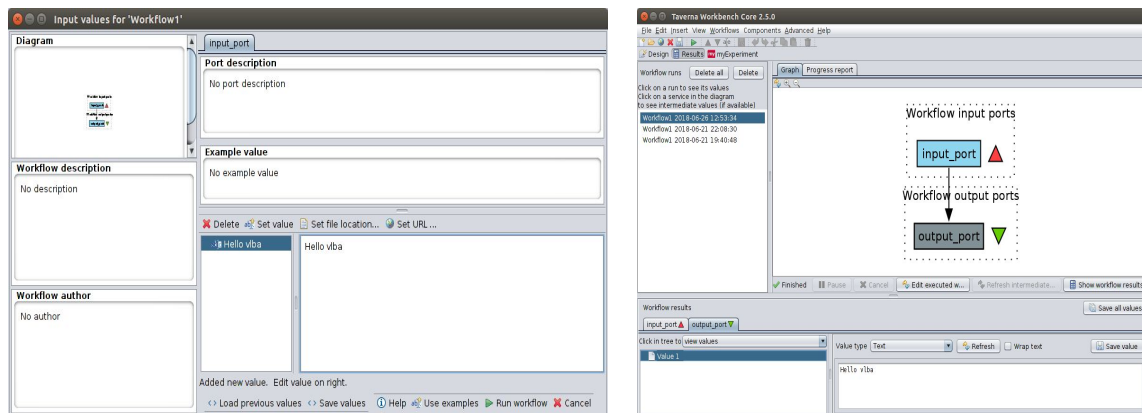
The Taverna is built to for the scientific community it makes it easy to share there workflow with other, they can just share the .t2flow file to other they import to there workspace and modify and run that. There is online portal also where you can contribute to by placing your workflows and other can directly import by using the link provided in that description.

4.4 Examples

- **simple example .**
 - It includes the simple workflow that has only input port and output port to create a new workflow right-click on the design area and select input port and do the same for output port now connect the input to the output port as shown in figure.



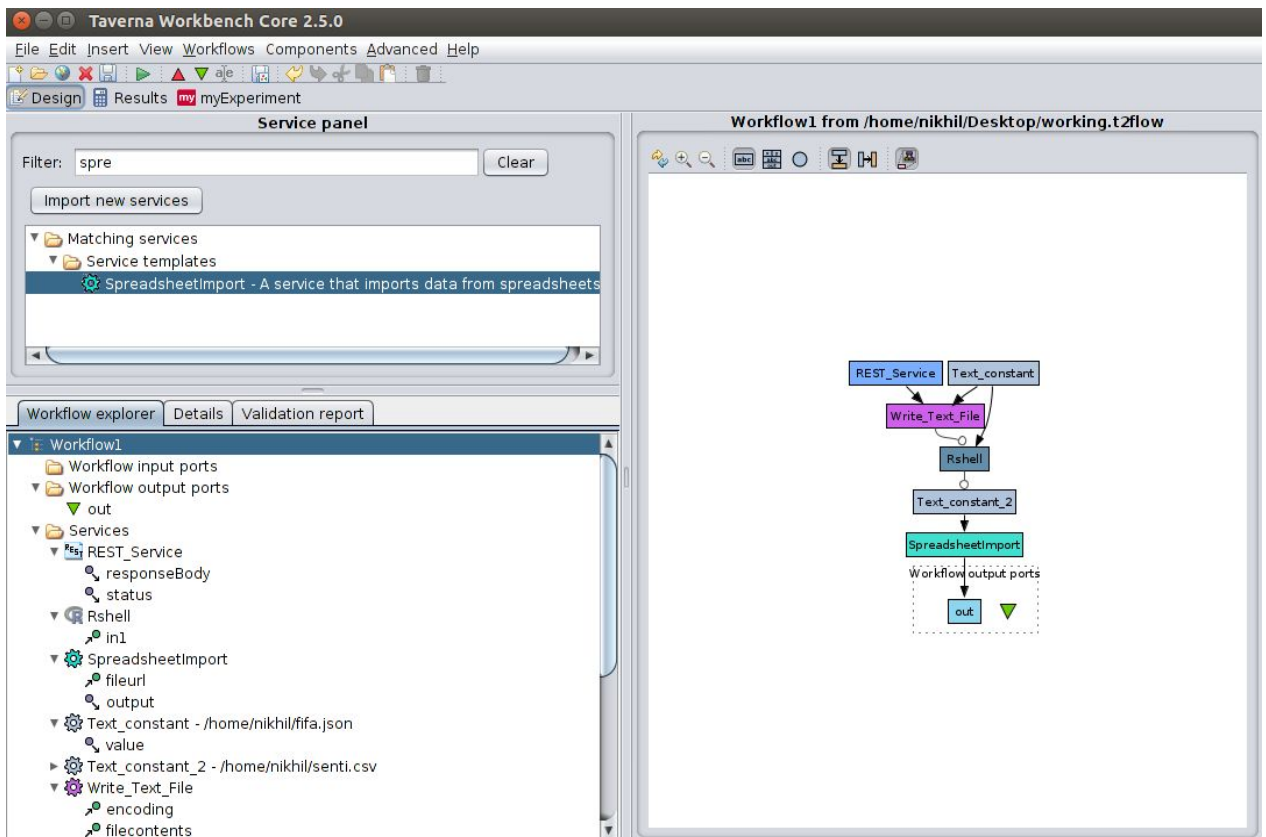
After these run the workflow by click the run button on top of the window, it will prompt a window ,set a value to the input port and click the “run workflow” button at bottom of the screen.



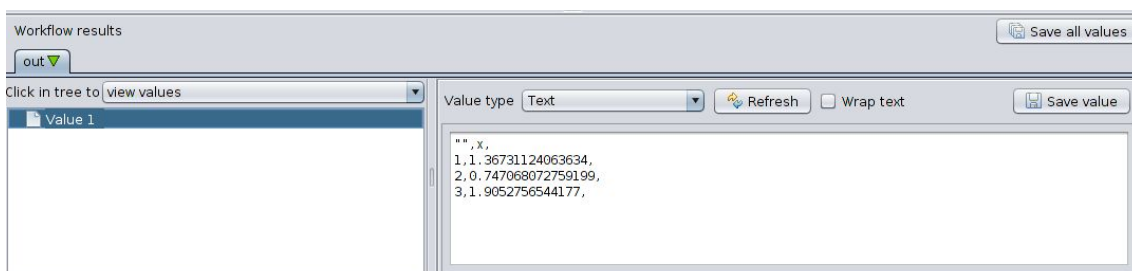
The output is displayed in the bottom of the workbench ,there is a progress chart tab where we can view the the current working task(services).

- **Another Example.**
 - For these task the service that are used is
 - REST
 - Rshell
 - SpreadsheetImport
 - Text_constant
 - Write_to_file
 - **REST** - These service is used to get the json GET response from the GoogleNewsApi about the “sports” category .
 - **Write_to_File** - these service is used to write the json response from REST service to a file, The file name and directory given as a input to these service by “Text_constant”.

- **Rshell** - Rshell service is used to calculate the sentiment of the each text by the json response stored in the file and write back the result to the new .CSV file the Text_constant is used to give the name and the directory of the file to write.
- **SpreadSheetImport** - It is used to read the Excel or .csv file the file created by the Rshell can be read by these service and an output port is connected to Spreadsheetimport to display the content of the spreadsheet.



The output of the Workflow is displayed in picture below, and also it create two files one as “fifa.json” and “fifa.csv”.



The workflow file for both the first example and the second example(.t2flow files) are on provided with the VMware snapshot and a github hub link is provided <https://github.com>

5. References

1. Apache Airflow- <https://airflow.apache.org/project.html>
2. Apache Taverna - <https://taverna.incubator.apache.org/documentation/>
3. Github link for the source code - <https://github.com/nikhilmandya/VLBA>
4. Blog referred are
 - a. <https://blog.godatadriven.com/practical-airflow-tutorial>
 - b. <https://danidelvalle.me/2016/09/12/im-sorry-cron-ive-met-airbnbs-airflow/>
 - c. <https://github.com/vishalsatam/Data-Pipelining/tree/master/Airflow>
 - d. <http://michal.karzynski.pl/blog/2017/03/19/developing-workflows-with-apache-airflow/>
 - e. https://www.myexperiment.org/workflows?order=last_updated