
PySpark Example Project

This document is designed to be read in parallel with the code in the `pyspark-template-project` repository. Together, these constitute what we consider to be a ‘best practices’ approach to writing ETL jobs using Apache Spark and its Python (‘PySpark’) APIs. This project addresses the following topics:

- how to structure ETL code in such a way that it can be easily tested and debugged;
- how to pass configuration parameters to a PySpark job;
- how to handle dependencies on other modules and packages; and,
- what constitutes a ‘meaningful’ test for an ETL job.

ETL Project Structure

The basic project structure is as follows:

```
1 root/
2 |-- configs/
3 |   |-- etl_config.json
4 |-- dependencies/
5 |   |-- logging.py
6 |   |-- spark.py
7 |-- jobs/
8 |   |-- etl_job.py
9 |-- tests/
10 |   |-- test_data/
11 |       |-- employees/
12 |       |-- employees_report/
13 |   |-- test_etl_job.py
14 |   build_dependencies.sh
15 |   packages.zip
16 |   Pipfile
17 |   Pipfile.lock
```

The main Python module containing the ETL job (which will be sent to the Spark cluster), is `jobs/etl_job.py`. Any external configuration parameters required by `etl_job.py` are stored in JSON format in `configs/etl_config.json`. Additional modules that support this job can be kept in the `dependencies` folder (more on this later). In the project’s root we include `build_dependencies.sh`, which is a bash script for building these dependencies into a zip-file to be sent to the cluster (`packages.zip`). Unit test modules are kept in the `tests` folder and small chunks of representative input and output data, to be used with the tests, are kept in `tests/test_data` folder.

Structure of an ETL Job

In order to facilitate easy debugging and testing, we recommend that the ‘Transformation’ step be isolated from the ‘Extract’ and ‘Load’ steps, into its own function - taking input data arguments in the form of DataFrames and returning the transformed data as a single DataFrame. Then, the code that surrounds the use of the transformation function in the `main()` job function, is concerned with Extracting the data, passing it to the transformation function and then Loading (or writing) the results to their ultimate destination. Testing is simplified, as mock or test data can be passed to the transformation function and the results explicitly verified, which would not be possible if all of the ETL code resided in `main()` and referenced production data sources and destinations.

More generally, transformation functions should be designed to be *idempotent*. This is a technical way of saying that the repeated application of the transformation function should have no impact on the fundamental state of output data, until the moment the input data changes. One of the key advantages of idempotent ETL jobs, is that they can be set to run repeatedly (e.g. by using `cron` to trigger the `spark-submit` command above, on a pre-defined schedule), rather than having to factor-in potential dependencies on other ETL jobs completing successfully.

Passing Configuration Parameters to the ETL Job

Although it is possible to pass arguments to `etl_job.py`, as you would for any generic Python module running as a ‘main’ program - by specifying them after the module’s filename and then parsing these command line arguments - this can get very complicated, very quickly, especially when there are a lot of parameters (e.g. credentials for multiple databases, table names, SQL snippets, etc.). This also makes debugging the code from within a Python interpreter extremely awkward, as you don’t have access to the command line arguments that would ordinarily be passed to the code, when calling it from the command line.

A much more effective solution is to send Spark a separate file - e.g. using the `--files configs/etl_config.json` flag with `spark-submit` - containing the configuration in JSON format, which can be parsed into a Python dictionary in one line of code with `json.loads(config_file_contents)`. Testing the code from within a Python interactive console session is also greatly simplified, as all one has to do to access configuration parameters for testing, is to copy and paste the contents of the file - e.g.,

```
1 import json
2
3 config = json.loads('{"field": "value"}')
```

For the exact details of how the configuration file is located, opened and parsed, please see the `start_spark()` function in `dependencies/spark.py` (also discussed further below), which

in addition to parsing the configuration file sent to Spark (and returning it as a Python dictionary), also launches the Spark driver program (the application) on the cluster and retrieves the Spark logger at the same time.

Packaging ETL Job Dependencies

In this project, functions that can be used across different ETL jobs are kept in a module called `dependencies` and referenced in specific job modules using, for example,

```
1 from dependencies.spark import start_spark
```

This package, together with any additional dependencies referenced within it, must be copied to each Spark node for all jobs that use `dependencies` to run. This can be achieved in one of several ways:

1. send all dependencies as a `zip` archive together with the job, using `--py-files` with Spark submit;
2. formally package and upload `dependencies` to somewhere like the `PyPI` archive (or a private version) and then run `pip3 install dependencies` on each node; or,
3. a combination of manually copying new modules (e.g. `dependencies`) to the Python path of each node and using `pip3 install` for additional dependencies (e.g. for `requests`).

Option (1) is by far the easiest and most flexible approach, so we will make use of this for now. To make this task easier, especially when modules such as `dependencies` have additional dependencies (e.g. the `requests` package), we have provided the `build_dependencies.sh` bash script for automating the production of `packages.zip`, given a list of dependencies documented in `Pipfile` and managed by the `pipenv` python application (discussed below).

Running the ETL job

Assuming that the `$SPARK_HOME` environment variable points to your local Spark installation folder, then the ETL job can be run from the project's root directory using the following command from the terminal,

```
1 $SPARK_HOME/bin/spark-submit \  
2 --master local[*] \  
3 --packages 'com.somesparkjar.dependency:1.0.0' \  
4 --py-files packages.zip \  
5 --files configs/etl_config.json \  
6 jobs/etl_job.py
```

Briefly, the options supplied serve the following purposes:

-
- `--master local[*]` - the address of the Spark cluster to start the job on. If you have a Spark cluster in operation (either in single-executor mode locally, or something larger in the cloud) and want to send the job there, then modify this with the appropriate Spark IP - e.g. `spark://the-clusters-ip-address:7077`;
 - `--packages 'com.somesparkjar.dependency:1.0.0,...'` - Maven coordinates for any JAR dependencies required by the job (e.g. JDBC driver for connecting to a relational database);
 - `--files configs/etl_config.json` - the (optional) path to any config file that may be required by the ETL job;
 - `--py-files packages.zip` - archive containing Python dependencies (modules) referenced by the job; and,
 - `jobs/etl_job.py` - the Python module file containing the ETL job to execute.

Full details of all possible options can be found here. Note, that we have left some options to be defined within the job (which is actually a Spark application) - e.g. `spark.cores.max` and `spark.executor.memory` are defined in the Python script as it is felt that the job should explicitly contain the requests for the required cluster resources.

Debugging Spark Jobs Using `start_spark`

It is not practical to test and debug Spark jobs by sending them to a cluster using `spark-submit` and examining stack traces for clues on what went wrong. A more productive workflow is to use an interactive console session (e.g. IPython) or a debugger (e.g. the `pdb` package in the Python standard library or the Python debugger in Visual Studio Code). In practice, however, it can be hard to test and debug Spark jobs in this way, as they implicitly rely on arguments that are sent to `spark-submit`, which are not available in a console or debug session.

We wrote the `start_spark` function - found in `dependencies/spark.py` - to facilitate the development of Spark jobs that are aware of the context in which they are being executed - i.e. as `spark-submit` jobs or within an IPython console, etc. The expected location of the Spark and job configuration parameters required by the job, is contingent on which execution context has been detected. The docstring for `start_spark` gives the precise details,

```
1 def start_spark(app_name='my_spark_app', master='local[*]',
2               jar_packages=[],
3               files=[], spark_config={}):
4     """Start Spark session, get Spark logger and load config files.
5
6     Start a Spark session on the worker node and register the Spark
7     application with the cluster. Note, that only the app_name argument
8     will apply when this is called from a script sent to spark-submit.
```

```

 8     All other arguments exist solely for testing the script from within
 9     an interactive Python console.
10
11     This function also looks for a file ending in 'config.json' that
12     can be sent with the Spark job. If it is found, it is opened,
13     the contents parsed (assuming it contains valid JSON for the ETL
14     job
15     configuration) into a dict of ETL job configuration parameters,
16     which are returned as the last element in the tuple returned by
17     this function. If the file cannot be found then the return tuple
18     only contains the Spark session and Spark logger objects and None
19     for config.
20
21     The function checks the enclosing environment to see if it is being
22     run from inside an interactive console session or from an
23     environment which has a `DEBUG` environment variable set (e.g.
24     setting `DEBUG=1` as an environment variable as part of a debug
25     configuration within an IDE such as Visual Studio Code or PyCharm.
26     In this scenario, the function uses all available function
27     arguments
28     to start a PySpark driver from the local PySpark package as opposed
29     to using the spark-submit and Spark cluster defaults. This will
30     also
31     use local module imports, as opposed to those in the zip archive
32     sent to spark via the --py-files flag in spark-submit.
33
34     :param app_name: Name of Spark app.
35     :param master: Cluster connection details (defaults to local[*]).
36     :param jar_packages: List of Spark JAR package names.
37     :param files: List of files to send to Spark cluster (master and
38     workers).
39     :param spark_config: Dictionary of config key-value pairs.
40     :return: A tuple of references to the Spark session, logger and
41     config dict (only if available).
42     """
43     # ...
44
45     return spark_sess, spark_logger, config_dict

```

For example, the following code snippet,

```

1  spark, log, config = start_spark(
2      app_name='my_etl_job',
3      jar_packages=['com.somesparkjar.dependency:1.0.0'],
4      files=['configs/etl_config.json'])

```

Will use the arguments provided to `start_spark` to setup the Spark job if executed from an interactive console session or debugger, but will look for the same arguments sent via `spark-submit` if that is how the job has been executed.

Automated Testing

In order to test with Spark, we use the `pyspark` Python package, which is bundled with the Spark JARs required to programmatically start-up and tear-down a local Spark instance, on a per-test-suite basis (we recommend using the `setUp` and `tearDown` methods in `unittest.TestCase` to do this once per test-suite). Note, that using `pyspark` to run Spark is an alternative way of developing with Spark as opposed to using the PySpark shell or `spark-submit`.

Given that we have chosen to structure our ETL jobs in such a way as to isolate the ‘Transformation’ step into its own function (see ‘Structure of an ETL job’ above), we are free to feed it a small slice of ‘real-world’ production data that has been persisted locally - e.g. in `tests/test_data` or some easily accessible network directory - and check it against known results (e.g. computed manually or interactively within a Python interactive console session).

To execute the example unit test for this project run,

```
1 pipenv run python -m unittest tests/test_*.py
```

If you’re wondering what the `pipenv` command is, then read the next section.

Managing Project Dependencies using Pipenv

We use `pipenv` for managing project dependencies and Python environments (i.e. virtual environments). All direct packages dependencies (e.g. NumPy may be used in a User Defined Function), as well as all the packages used during development (e.g. PySpark, flake8 for code linting, IPython for interactive console sessions, etc.), are described in the `Pipfile`. Their **precise** downstream dependencies are described in `Pipfile.lock`.

Installing Pipenv

To get started with `Pipenv`, first of all download it - assuming that there is a global version of Python available on your system and on the `PATH`, then this can be achieved by running the following command,

```
1 pip3 install pipenv
```

`Pipenv` is also available to install from many non-Python package managers. For example, on OS X it can be installed using the Homebrew package manager, with the following terminal command,

```
1 brew install pipenv
```

For more information, including advanced configuration options, see the official pipenv documentation.

Installing this Projects' Dependencies

Make sure that you're in the project's root directory (the same one in which the `Pipfile` resides), and then run,

```
1 pipenv install --dev
```

This will install all of the direct project dependencies as well as the development dependencies (the latter a consequence of the `--dev` flag).

Running Python and IPython from the Project's Virtual Environment

In order to continue development in a Python environment that precisely mimics the one the project was initially developed with, use Pipenv from the command line as follows,

```
1 pipenv run python3
```

The `python3` command could just as well be `ipython3`, for example,

```
1 pipenv run ipython
```

This will fire-up an IPython console session *where the default Python 3 kernel includes all of the direct and development project dependencies* - this is our preference.

Pipenv Shells

Prepending `pipenv` to every command you want to run within the context of your Pipenv-managed virtual environment can get very tedious. This can be avoided by entering into a Pipenv-managed shell,

```
1 pipenv shell
```

This is equivalent to 'activating' the virtual environment; any command will now be executed within the virtual environment. Use `exit` to leave the shell session.

Automatic Loading of Environment Variables

Pipenv will automatically pick-up and load any environment variables declared in the `.env` file, located in the package's root directory. For example, adding,

```
1 SPARK_HOME=applications/spark-2.3.1/bin
2 DEBUG=1
```

Will enable access to these variables within any Python program -e.g. via a call to `os.environ['SPARK_HOME']`. Note, that if any security credentials are placed here, then this file **must** be removed from source control - i.e. add `.env` to the `.gitignore` file to prevent potential security risks.