Published in Towards Data Science

David Suarez Follow

Oct 4, 2021 · 10 min read · Listen

Save

Install custom Python Libraries from private PyPI on Databricks

In this blog post I'm going to explain how to integrate your private PyPI repositories on Databricks clusters step by step. As a result, you will be able to install your own custom Python Libraries as easily as you normally do with the public ones.

Photo by Anaya Katlego on Unsplash

This method opens up the door for sharing code and libraries across data teams while keeping versioning. Moreover, it gives the possibility to apply hybrid coding approaches on Databricks where you can combine libraries written on local machine (properly tested and released using CI/CD pipelines) and Notebooks using those libraries.

> *Note: this tutorial is focused on Azure. If you happen to be working on a different cloud provider but you are still interested in this topic, keep reading. You will most likely be able to easily translate the concept to other environments as well.*

Why should you use a private PyPI repository

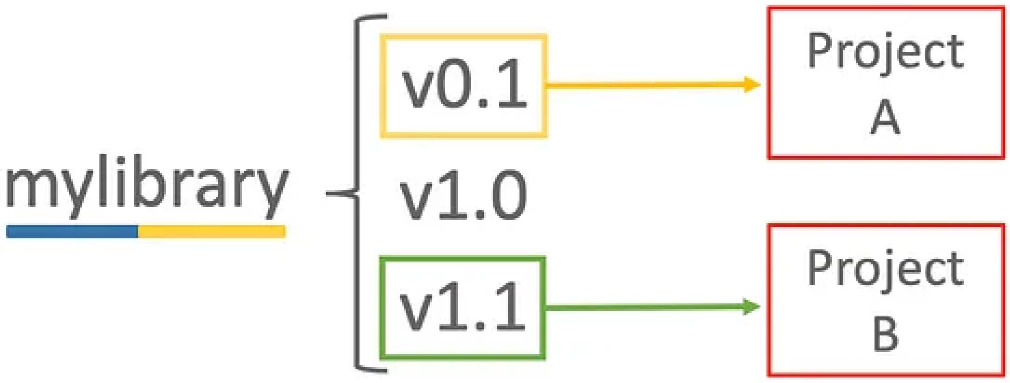
Databricks provides a very simple way of installing public Libraries on Clusters, so you can install them with just a few clicks. Unfortunately, when it comes to self-made custom Libraries, the process is not as easy.

In this case, the most popular solution that I have seen so far is packaging the code into a Python Wheel and upload it to a Blob Storage, so you can install it as a DBFS Library. The second solution found is installing a Library directly from a Git Repository, so you have to provide Git URL and authentication method. Although both solutions do the trick, we are missing something very important here: versioning.

Why Library versioning is important?

When writing your own libraries, sometimes you'll be writing code that is so tailored to a specific solution that the reuse possibilities are not too high. In other occasions, you might create a code package with classes and utility functions that you would like to share with your co-workers so they don't have to reinvent the wheel. In both cases, if you keep working on your library, you will be introducing changes that could mess up solutions that are even running in production.

The only way to prevent that from happening and keep everybody happy, is by introducing library versioning. This way, each solution or environment can use a different fixed version of the library and nothing will break when changes are introduced.



Two projects using different versions of the same Library | Image by Author

Private PyPI repositories on Databricks

Coming back to self-made Libraries in Databricks, none of the previous presented approaches come with a robust solution for this issue. You can either upload and maintain Python Wheels with different version names on Blob Storage, or release branches on the Git Repo. Even if that does the trick, it will be hard to trace back the changes of each version, and you will have to build and maintain the mechanism yourself, and even harder, align with all the users of your library. In other words, these are clunky workarounds for an already solved problem.

PyPI repositories have been around for years, and the most popular Python libraries out there are published on a public PyPI repo. In the same way, you can have your own PyPI repository and benefit from the versioning capabilities as well — and you may want this repository to be private so your organization libraries are not publicly available to everyone.

On **Azure DevOps**, it's super easy to have your own private PyPI repository. You just need to create an **Artifact Feed**, which under the hood has package repositories for the most common programming languages. After that, you can publish your **Python Wheels** there and benefit from all the features of it. For example: the ability of keeping track from what commit of your Git Repo each version was released.

Tutorial

So now that I have explained some of the benefits of using a Private PyPI repository for keeping our custom Libraries' versions, let's see how can we integrate it on Databricks Clusters so you are able to install your custom Libraries.

Prerequisites

In summary, you need your custom **Python Package published** in an Azure **Artifact Feed**, and a **KeyVault** registered as a **secret scope** in a **Databricks** Workspace.

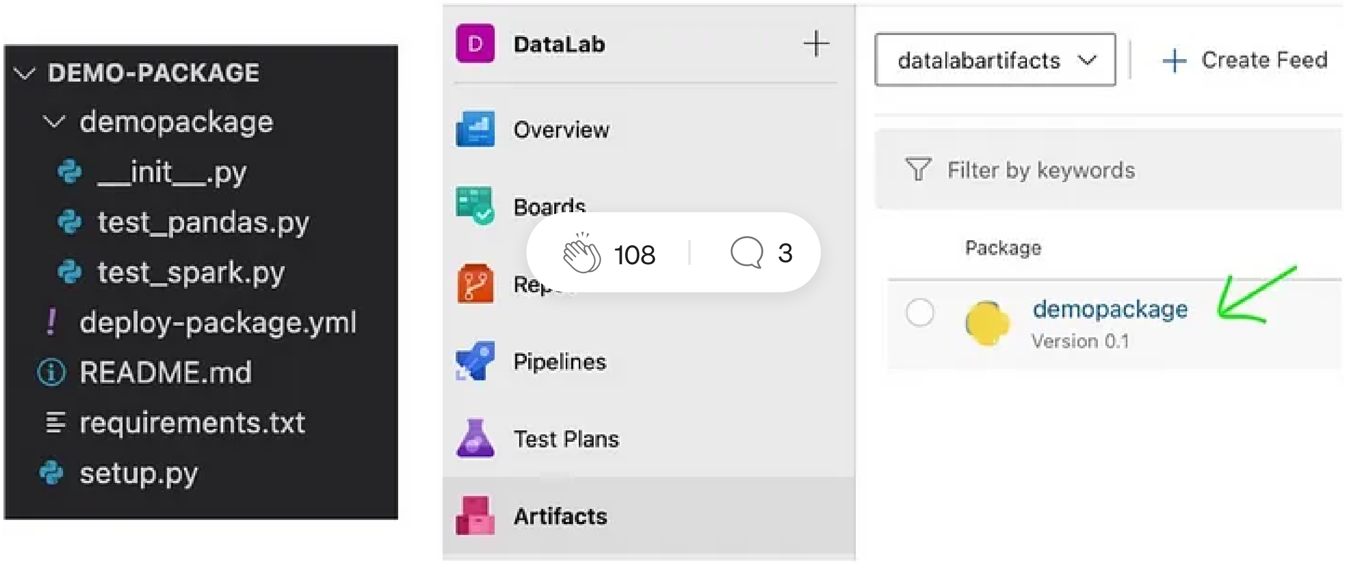
If you are a bit lost and don't know how to get there yet, here's a list with all

- Git Repo in Azure Repos (how to create a Git repo here).
- Python code in the Git Repo with a `setup.py` to generate a Python Wheel (how to generate a Python Wheel here).
- Artifact Feed (how to create an Artifact Feed here).
- Azure Pipeline YAML file in the Git Repo to generate and publish the Python Wheel to the Artifact Feed (code here).
- Register and run Azure Pipeline from YAML file (how to do it here).
- Azure Databricks Workspace.
- Azure Key Vault.
- Azure Key Vault registered in Databricks Workspace as a Secret Scope (how to do it here).

My own demo setup

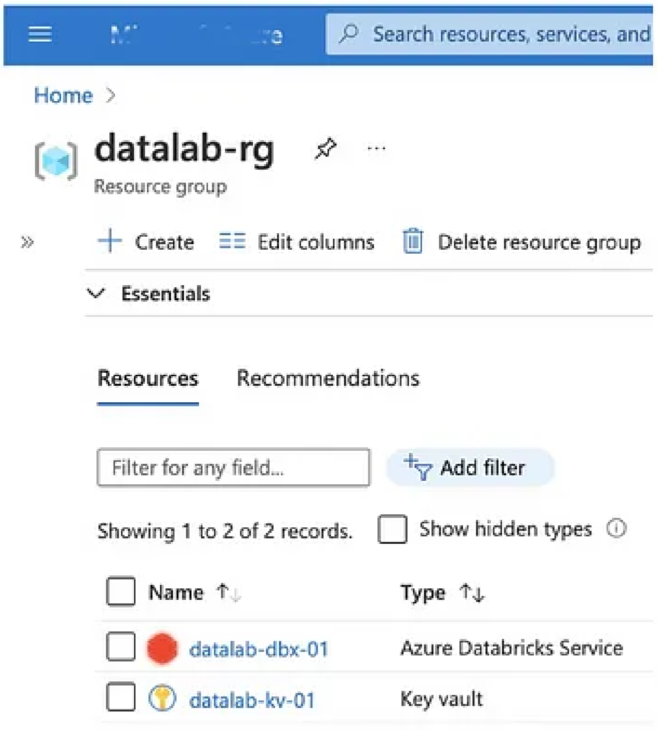
For this article I've made a demo setup, and it might be good for you to see it for just in case you see any reference.

The Python package that I use for the demo (called "demopackage") is very simple. It contains just a couple of functions that generates Dataframes with PySpark and Pandas. As you can see in the picture, it's also been published to my Artifact Feed called "datalabartifacts".



Code folder structure and published package on my own Artifact Feed | Image by Author

On Azure, I just have a Resource Group with Databricks and Key Vault. Furthermore, the Key Vault has been registered in Databricks as a Secret Scope with the same name.



Resources on my own Resource group | Image by Author

The magic recipe

The goal of this process is to allow the underlying VMs of the Spark Clusters in Databricks to integrate your Private PyPI repository existing in Artifact Feed and to be able to install Python Libraries from it.

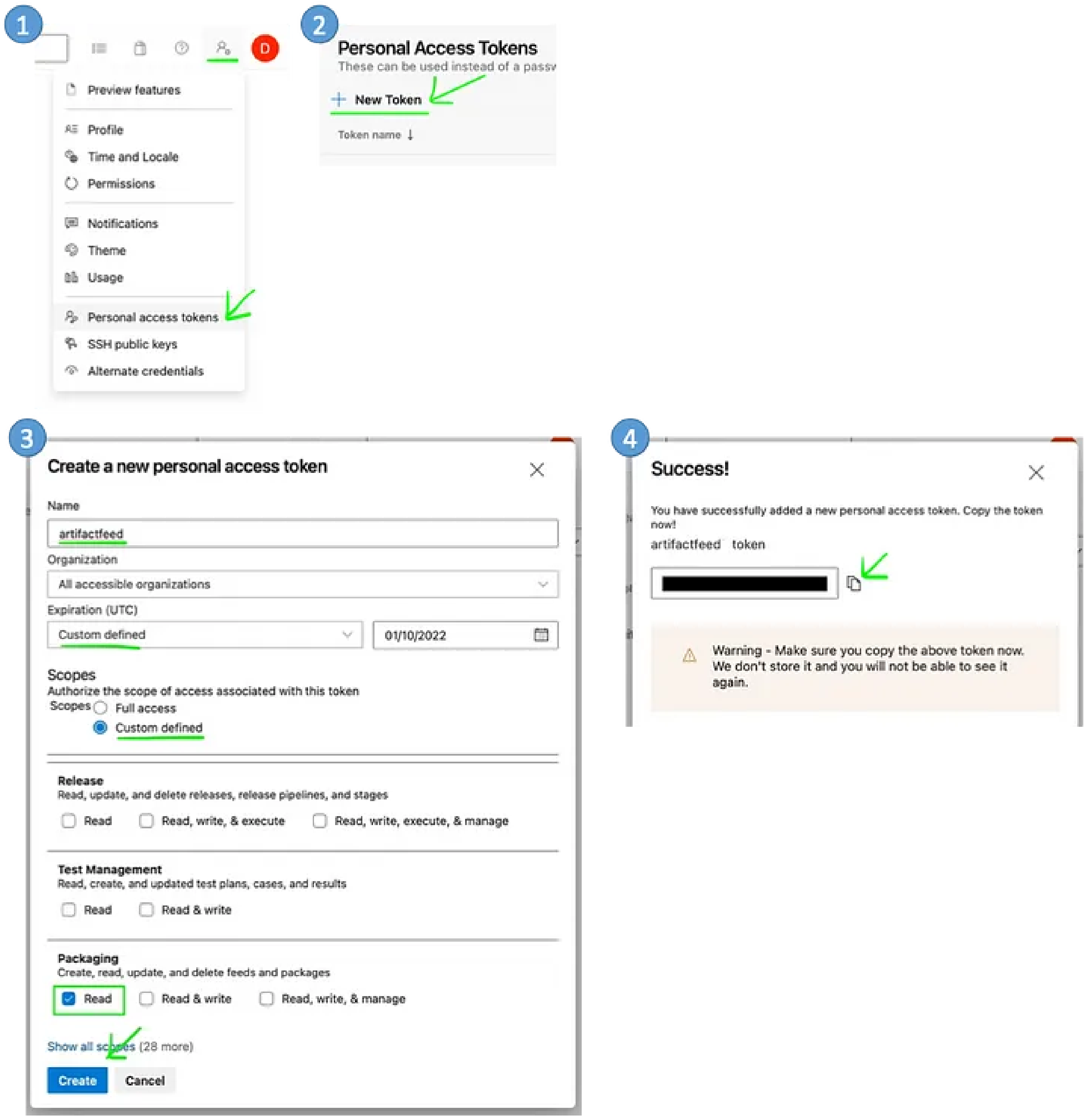
1. Generate Personal Access Token on Azure DevOps

Because our Artifact Feed is private (and we want to keep it private), we need to provide a way for our VMs to authenticate against the Arifact Feed.

Unfortunately, after doing a lot of research, the securest way of doing so that I've found is using an Azure DevOps Personal Access Tokens (PAT). This PAT is personal and issued per DevOps user, so the Spark VMs will use this token to impersonate as the user to do the authentication. This means that in the Artifact Feed registry you will see that the issuer user is the one downloading packages. Moreover, the PAT expires after one year, so be aware you will need to renew it before it expires.

The advice here is to create a Service Account (fake user in DevOps) with restricted access and only used for generating this PAT, and renew it before the expiration day.

In any case, you can generate a Personal Access Token from Azure DevOps by clicking on *User Settings → Personal access tokens → New Token*. Give it a name, specify the expiration time, and the permissions of the PAT (only Read on Packaging is required). Once you are done, copy the generated token.



Create a new personal access token

Name: artifactfeed

Organization: All accessible organizations

Expiration (UTC): Custom defined 01/10/2022

Scopes
Authorize the scope of access associated with this token
Scopes ○ Full access
[✓ Custom defined]

Release
Read, update, and delete releases, release pipelines, and stages
☐ Read ☐ Read, write, & execute ☐ Read, write, execute, & manage

Test Management
Read, create, and updated test plans, cases, and results
☐ Read ☐ Read & write

Packaging
Create, read, update, and delete feeds and packages
☑ Read ☐ Read & write ☐ Read, write, & manage

Success!

You have successfully added a new personal access token. Copy the token now!

artifactfeed token

[redacted]

⚠ Warning - Make sure you copy the above token now. We don't store it and you will not be able to see it again.

Steps to generate Personal Access Token on Azure DevOps | Image by Author

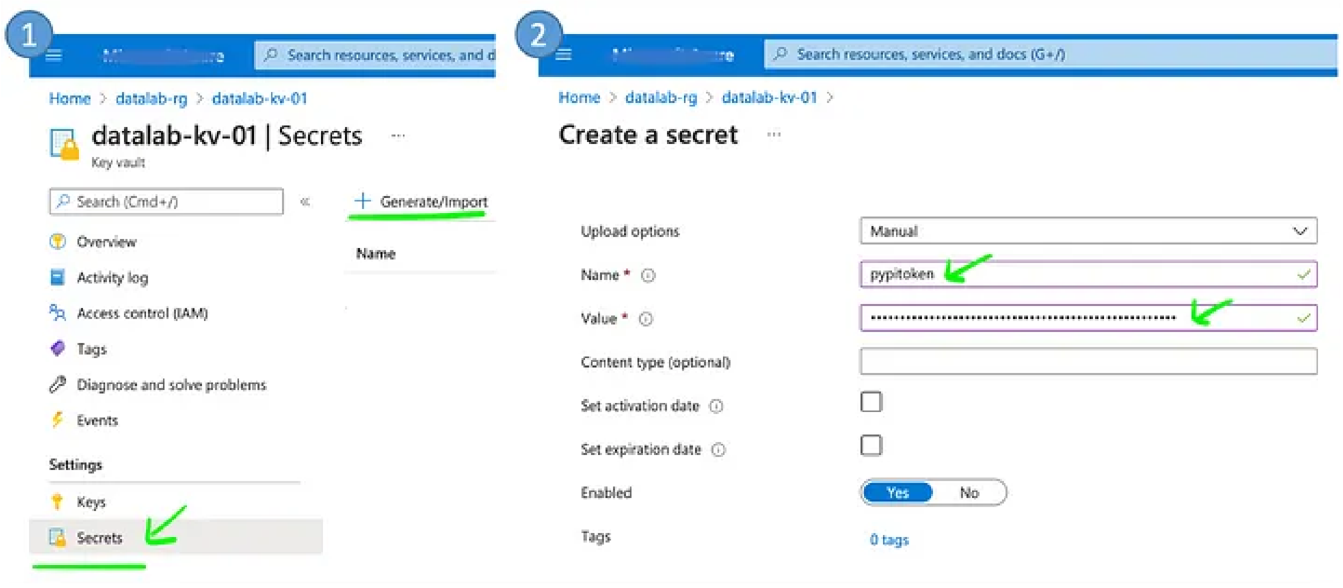
2. Add the token as a Secret in KeyVault

The generated token can be used to retrieve packages from the Artifact Feed. If this gets exfiltrated outside of your organisation, this can be a big problem. It means that hackers can steal the software packages available in the Artifact Feeds.

Nobody wants that to happen, so we need to store the token in a safe place so we can use it without exposing it as plain text. For that, we need to create a new Secret in Azure Key Vault to store it.

Go to your Key Vault, click on *Secrets → Generate/Import*, and create a secret by giving it a name and using the PAT token generated in the previous step as value.

> *Note: the Key Vault used needs to be the one registered in the Databricks Workspace as Secret Scope.*



Upload options	Manual
Name *	pypitoken
Value *	••
Content type (optional)	
Set activation date	☐
Set expiration date	☐
Enabled	[✓ Yes] No
Tags	0 tags

Steps for creating a secret in Azure Key Vault | Image by Author

3. Add Environment Variable to Databricks Cluster

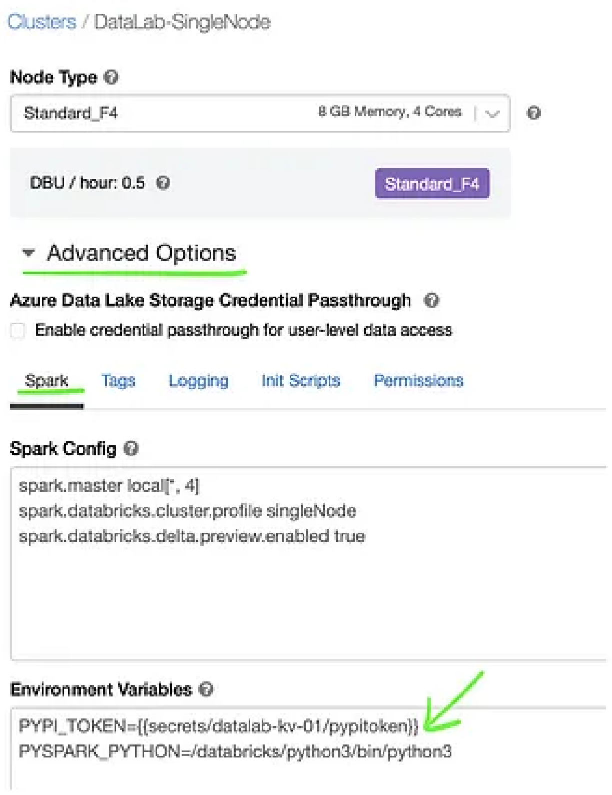
It's finally time to jump into Databricks Workspace!

Go to *Compute→ select Cluster → Cofiguration*, and proceed to edit it. Under *Advanced Options*, add the following *Environment Variable*:

```
PYPI_TOKEN={{secrets/YourSecretScopeName/pypitoken}}
```

This way, we are setting a value from a Secret on the Secret Scope that is connected to our Key Vault.

> *Note: Don't forget to replace the Secret Scope and Secret names by your own.*



```
spark.master local[*, 4]
spark.databricks.cluster.profile singleNode
spark.databricks.delta.preview.enabled true
```

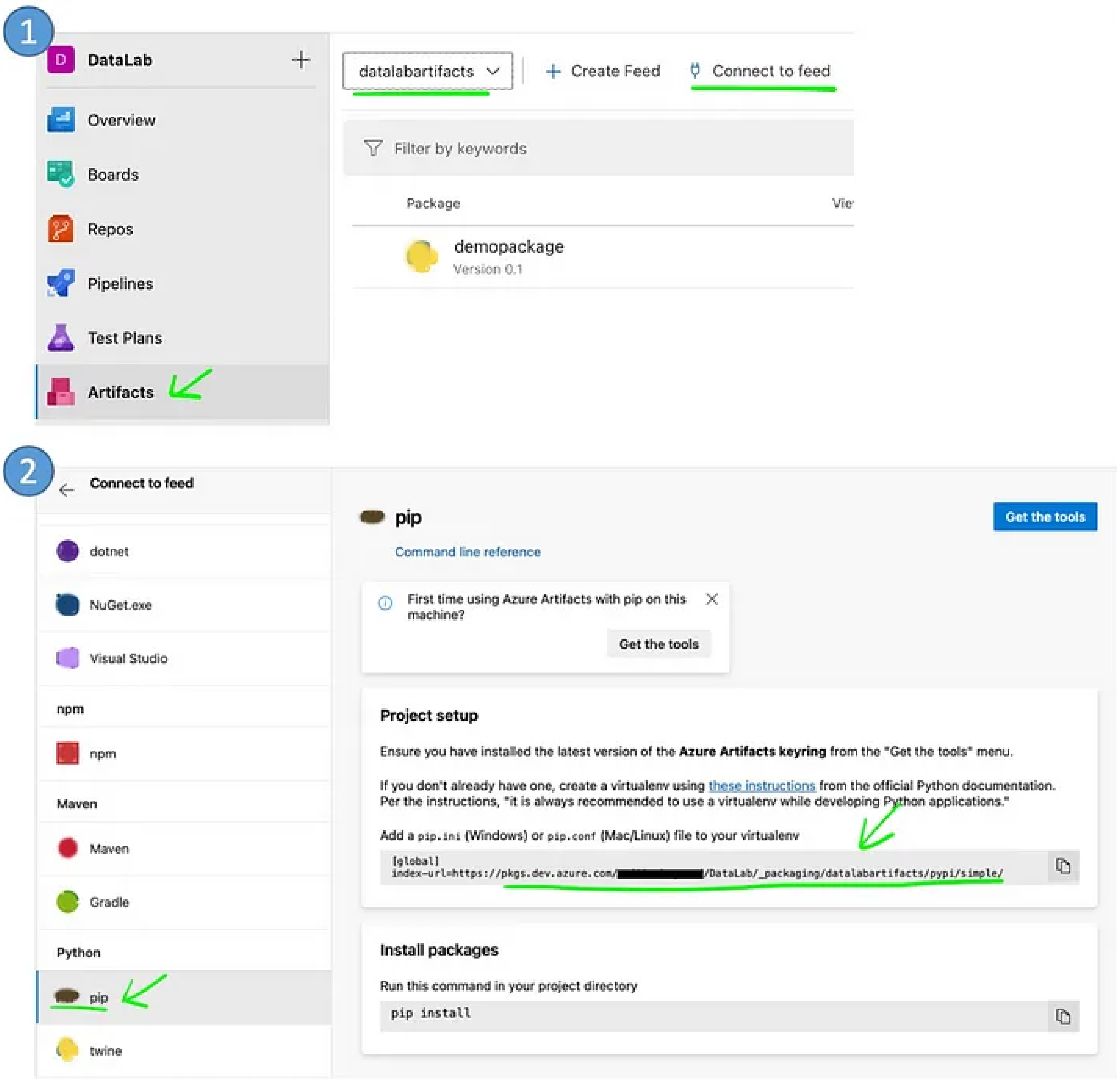
```
PYPI_TOKEN={{secrets/datalab-kv-01/pypitoken}}
PYSPARK_PYTHON=/databricks/python3/bin/python3
```

Set Environment Variable in Databricks Cluster from Secret Scope | Image by Author

4. Get PyPI repository URL

Let's move back to Azure DevOps for a second and get the URL of the private PyPI repository.

To do this, click on *Artifacts* → *select your Artifact Feed*→ *Connect to feed* → *pip*, and then copy the URL.



Project setup

Ensure you have installed the latest version of the **Azure Artifacts keyring** from the "Get the tools" menu.

If you don't already have one, create a virtualenv using these instructions from the official Python documentation. Per the instructions, "it is always recommended to use a virtualenv while developing Python applications."

Add a `pip.ini` (Windows) or `pip.conf` (Mac/Linux) file to your virtualenv

```
[global]
index-url=https://pkgs.dev.azure.com/[redacted]/DataLab/_packaging/datalabartifacts/pypi/simple/
```

Install packages

Run this command in your project directory

```
pip install
```

Get PyPI repository URL | Image by Author

4. Create Init Script for Databricks Clusters with the magic sauce

Before introducing the magic sauce, let me first explain the trick.

When you install a Library on a Databricks Cluster using the UI, Databricks instructs all the nodes to install the Library individually, so they pull the package and proceed with the installation.

This means that if we want to install packages from private PyPI repositories in Databricks Clusters, every node needs to be able to 1) find the private PyPI repo, and 2) authenticate successfully against it.

In order to do that, we have to tell each cluster node Pip installation what is the URL of the private PyPI repo, and how to authenticate, in this case by using the token authentication. The place to do that is the `/etc/pip.conf` file where we have to add new a `extra-index-url`.

In practice, we can achieve this in Databricks by making use of Init Scripts. These are Shell scripts that run on each node during cluster initialization.

The Init Script on itself would look like this (note that you have to replace the PyPI URL by your own):

```
#!/bin/bash
if [[ $PYPI_TOKEN ]]; then
   use $PYPI_TOKEN
fi
echo $PYPI_TOKEN
printf "[global]\n" > /etc/pip.conf
printf "extra-index-url =\n" >> /etc/pip.conf
printf
"\thttps://$PYPI_TOKEN@pkgs.dev.azure.com/organization/DataLab/_pack
aging/datalabartifacts/pypi/simple/\n" >> /etc/pip.conf
```

As you can see, we are setting references of the previously created Cluster Environment Variable `PYPI_TOKEN` into the `/etc/pip.conf` file, meaning no plain text is shown at all since the value will resolve at runtime.

> *Note: In case you try to display its value on Logs using `echo` command, you will only see `[REDACTED]` since Databricks hide all values coming from Secret Scopes.*

Because it can be a bit tricky to upload the script manually to Databricks, it's better to generate it easily from a Notebook by running the following code on a cell (don't forget to replace the PyPI URL by your own):

```
script = r"""
#!/bin/bash
if [[ $PYPI_TOKEN ]]; then
  use $PYPI_TOKEN
fi
echo $PYPI_TOKEN
printf "[global]\n" > /etc/pip.conf
printf "extra-index-url =\n" >> /etc/pip.conf
```

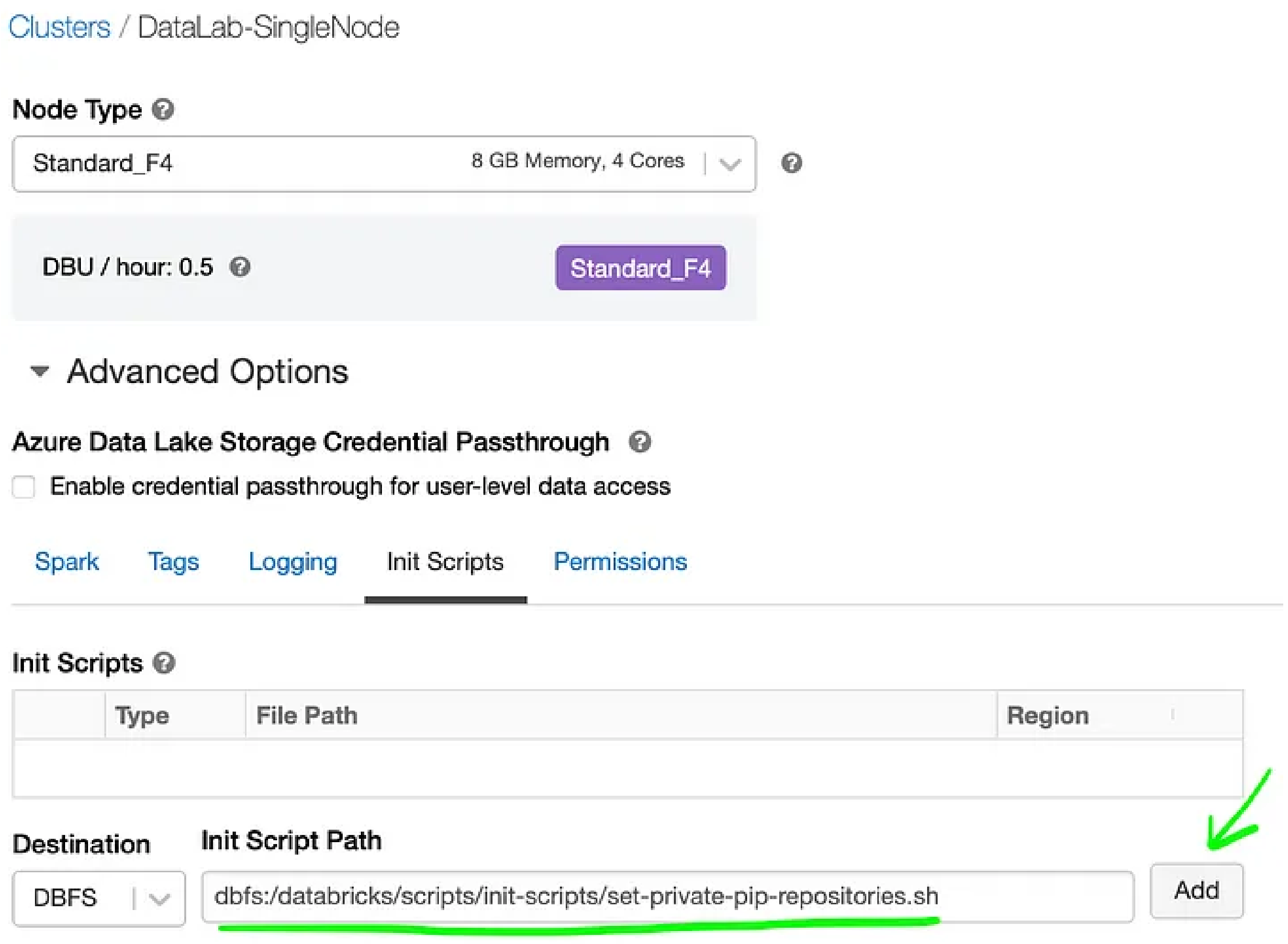
```
printf
"\thttps://$PYPI_TOKEN@pkgs.dev.azure.com/organization/DataLab/_pack
aging/datalabartifacts/pypi/simple/\n" >> /etc/pip.conf
"""

dbutils.fs.put("/databricks/scripts/init-scripts/set-private-pip-
repositories.sh", script, True)
```

> *Note: in this way we are creating a Cluster-scoped init script, better option than Global Init Script in this case, since the absence of the Env Variable would make other clusters' initialization fail.*

5. Add Init Script to your Databricks Cluster

Now we want our cluster to run this Init Script during initialization process. For that we go to *Compute → select Cluster → Configuration*, and edit the same cluster again. This time, we will add the *Init Script* DBFS path under *Advanced Options*.


Clusters / DataLab-SingleNode

Node Type: Standard_F4 — 8 GB Memory, 4 Cores

DBU / hour: 0.5 — Standard_F4

Advanced Options

Azure Data Lake Storage Credential Passthrough

- [ ] Enable credential passthrough for user-level data access

Spark | Tags | Logging | Init Scripts | Permissions

Init Scripts

	Type	File Path	Region

Destination: DBFS

Init Script Path: dbfs:/databricks/scripts/init-scripts/set-private-pip-repositories.sh

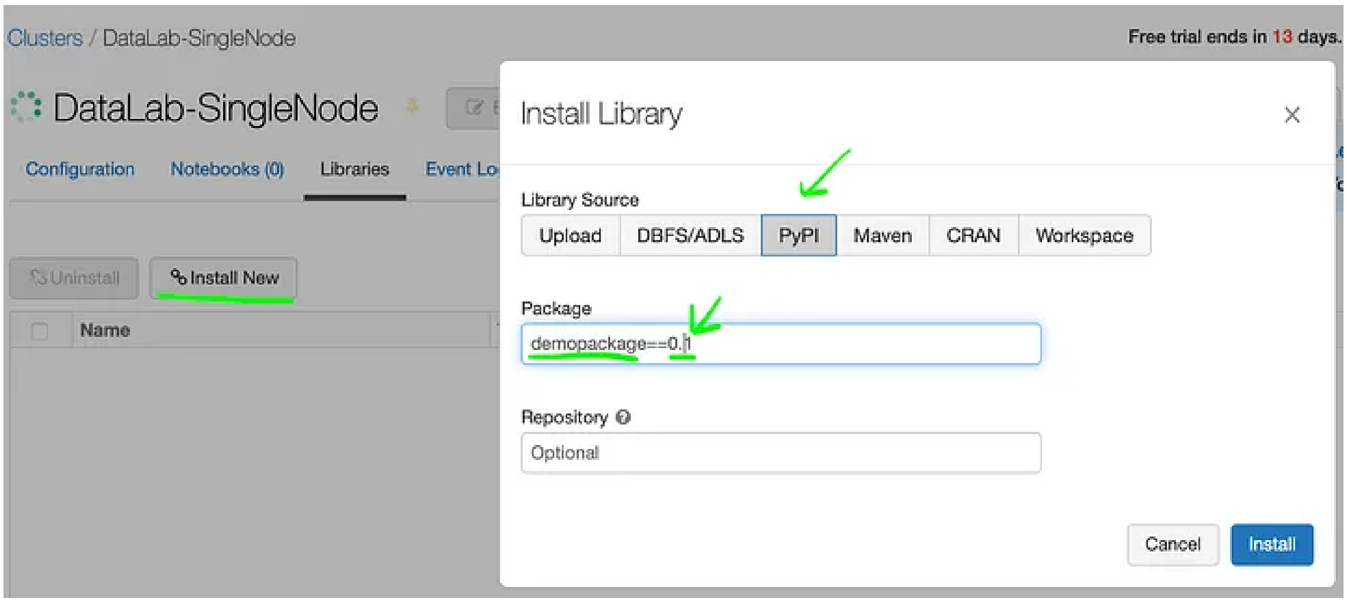
Add Init Script to Databricks Cluster | Image by Author

At this point in time, you can already start the cluster. It will run the Init Script and therefor your Private PyPI repository of your Artifact Feed will be totally accessible.

6. Install your Python Library in your Databricks Cluster

Just as usual, go to *Compute → select your Cluster → Libraries → Install New Library*.

Here you have to specify the name of your published package in the Artifact Feed, together with the specific version you want to install (unfortunately, it seems to be mandatory).



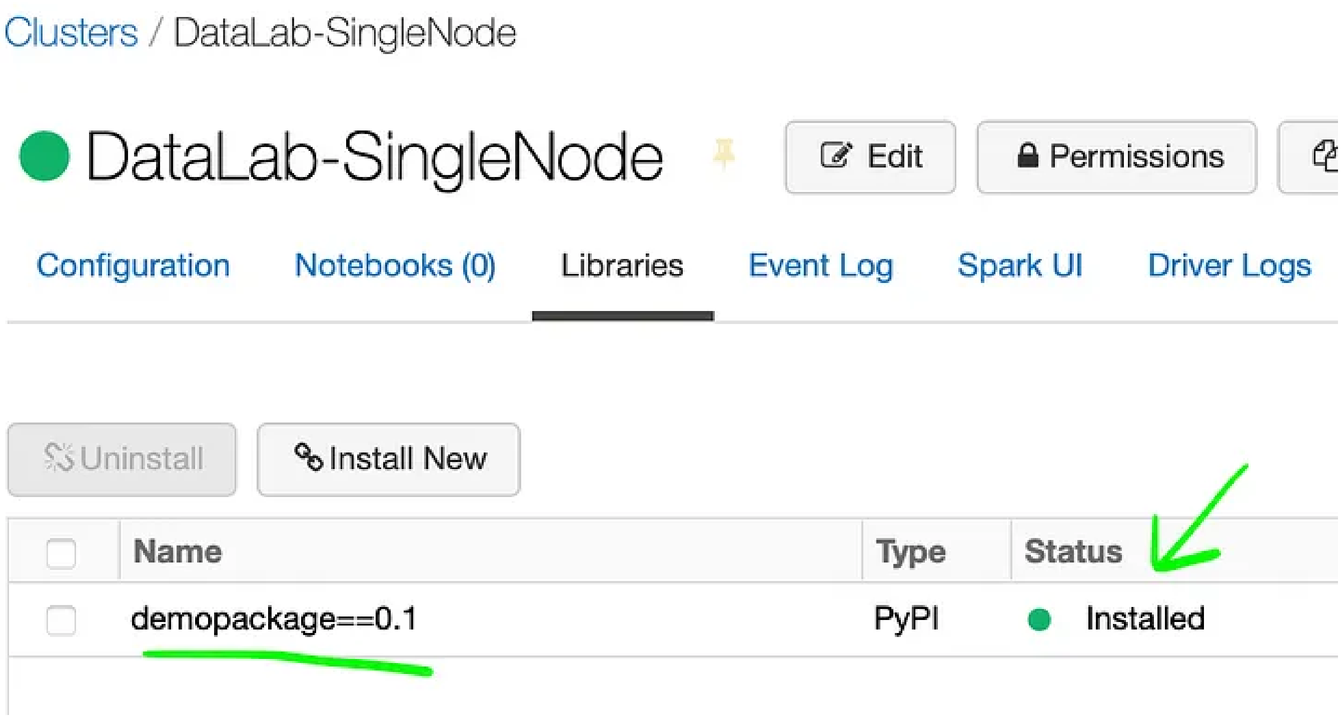
Library Source: PyPI

Package: demopackage==0.1

Repository: Optional

Install PyPI Library on Databricks Cluster | Image by Author

After a while you will see that the installation has succeeded! 🥳



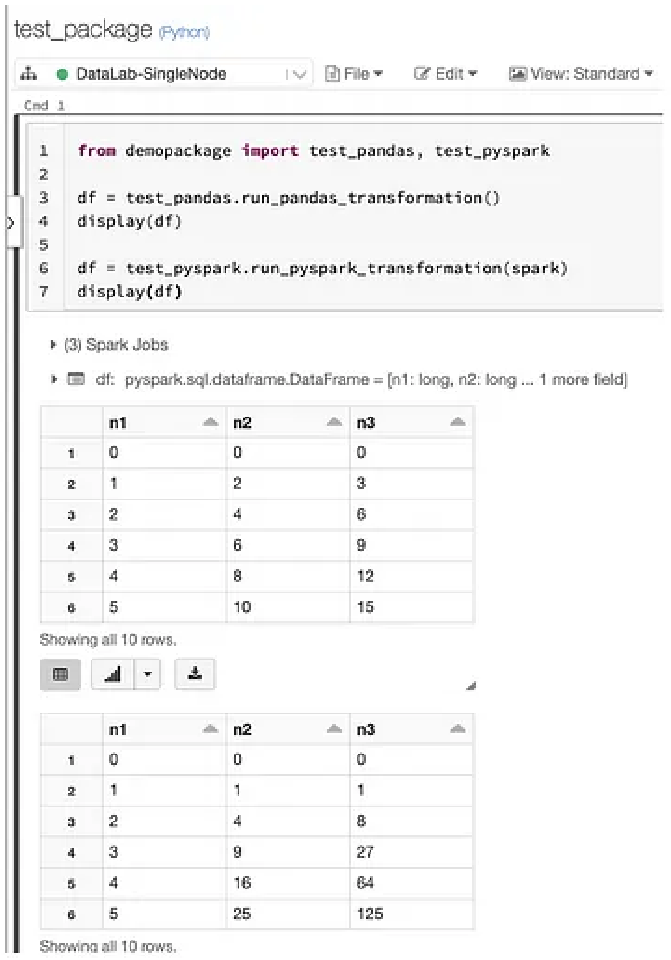
Name	Type	Status
demopackage==0.1	PyPI	● Installed

PyPI Library Installed | Image by Author

> *Note: be aware that now Pip is searching for packages in more than one PyPI repository. Because of this, you could have Library name collisions if there is another Library with the same name in another repository. If you don't use unique Library names, it's not guaranteed that you will install the Library you actually want..*

7. Use your Python Library from a Databricks Notebook

Finally, you can just open a new Databricks Notebook, import your library and enjoy the results! 🚀



```
from demopackage import test_pandas, test_pyspark

df = test_pandas.run_pandas_transformation()
display(df)

df = test_pyspark.run_pyspark_transformation(spark)
display(df)
```

(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [n1: long, n2: long ... 1 more field]

	n1	n2	n3
1	0	0	0
2	1	2	3
3	2	4	6
4	3	6	9
5	4	8	12
6	5	10	15

Showing all 10 rows.

	n1	n2	n3
1	0	0	0
2	1	1	1
3	2	4	8
4	3	9	27
5	4	16	64
6	5	25	125

Showing all 10 rows.

Using Custom Python Library on Databricks Notebook | Image by Author

Conclusions

Once you have reached this point, I think you can already imagine how nice this can be. This opens up the door for sharing code and libraries across data teams while keeping versioning.

Furthermore, it gives the possibility of thinking of hybrid coding approaches on Databricks where you can combine libraries written on local machine (properly tested and released using CI/CD pipelines) and Notebooks using those libraries.

I haven't seen this method published anywhere else on the internet, so that's why I decided to publish it here. I hope this is as helpful to you as it was for me a many of my colleagues. Good luck!

Databricks Python Libraries Data DevOps