Lab 1: "Hello, Flink!" - A Robust Local Development Workflow

**Goal:** Set up a local standalone Flink cluster, submit a Python job using a production-style workflow, and monitor that job using the Flink Web UI.

# 

# INDEX

[**Purpose of this Lab**](#_p7f0ylk2ib9x) **3**

[**Prerequisites**](#_yy74lt8y925e) **3**

[**Project Structure**](#_zes36rlq28cl) **6**

[**Part 1: Download and Set Up the Flink Cluster**](#_29bawngsdbc) **6**

[Step 1: Download, Unpack, and Relocate Flink](#_sumgkwhgkhmj) 6

[Step 2: Configure Your Shell Environment](#_lj4hunio8f2q) 7

[**Part 2: Create the Python Project and Configure Flink**](#_tt57411c7v3p) **8**

[Step 1: Create the Project Directory and Virtual Environment](#_p4az266d0pfb) 8

[Step 2: Install apache-flink](#_vx0cfxfwy42m) 9

[Step 3: Get the Python Executable Path and Configure Flink](#_b6vv0ncewbvc) 10

[**Part 3: Start the Flink Cluster and Submit the Job**](#_gislkc5x8u67) **10**

[Step 1: Start the Standalone Cluster](#_g7cuz4piyw8s) 10

[Step 2: Access the Flink Web UI](#_sj451ln0gv7w) 11

[Step 3: Create the Python Script](#_kyopx99c550d) 11

[Step 4: Submit the Job to the Cluster](#_8powbxzd1e5b) 12

[**Part 4: Observe and Cleanup**](#_u019cr1r5n19) **13**

[Step 1: Check the Web UI](#_z6bujymjt807) 13

[Step 2: Find the Output](#_fu04ucu2c5rd) 13

[Step 3: Cleanup](#_be20eu8fayxj) 14

[**Next Steps**](#_8r9yen47dyot) **14**

[**Troubleshooting Common Issues**](#_ne89dz15deex) **15**

# 

# Purpose of this Lab

This lab provides a hands-on introduction to the PyFlink DataStream API, emphasizing a correct and repeatable development workflow. It's designed for first-time Flink users to understand how to interact with a Flink cluster from the command line. By completing this lab, you will:

* **Provision:** Download and set up a standard Flink binary distribution using the command line.
* **Configure:** Correctly configure Flink to use an isolated Python virtual environment and configure your shell for easy access.
* **Operate:** Start a standalone Flink cluster and verify its Web UI is running.
* **Develop:** Write a simple PyFlink job that demonstrates a basic data transformation.
* **Submit & Monitor:** Submit your Python script to the running cluster and see it in the Web UI.
* **Inspect:** Analyze the job's output by inspecting the Flink TaskManager logs directly.

# Prerequisites

This lab assumes you are using an **Ubuntu** environment. Before starting, ensure you have these command-line tools installed.

**Visual Studio Code**

We'll need a text editor to write our Python script. First, check if you already have VS Code installed:

|  |
| --- |
| command -v code |



If this command prints a path, VS Code is installed, and you can skip the next step. If it prints nothing, install it from the terminal:

|  |
| --- |
| sudo snap install --classic code |

**Java Development Kit (JDK 11 or 17)**

This is the most critical prerequisite, as the Flink cluster itself is a Java application.You can use either JDK 11 or JDK 17 for this lab.

Check your version:

|  |
| --- |
| java -version |



If you don't have either version 11 or 17 installed, choose one and install it.

To install JDK 11:

|  |
| --- |
| sudo apt update sudo apt install openjdk-11-jdk -y |

To install JDK 17:

|  |
| --- |
| sudo apt update  sudo apt install openjdk-17-jdk -y |

After installation, set it as the default version for your system. You may be prompted to select the number corresponding to java-11-openjdk.

|  |
| --- |
| sudo update-alternatives --config java |

**Python 3.10**

This lab requires Python 3.10. First, check your system's default Python version.

|  |
| --- |
| python --version |

If the output is not Python 3.10.x, or if the python command is not found, follow these steps to install it and set it as the default.

Add the deadsnakes PPA to get specific Python versions

|  |
| --- |
| sudo add-apt-repository ppa:deadsnakes/ppa sudo apt update |

Install Python 3.10 and the corresponding venv package

|  |
| --- |
| sudo apt install python3.10 python3.10-venv -y |

Set python3.10 as the default 'python' command

|  |
| --- |
| sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.10 1 sudo update-alternatives --config python |

In the prompt from the last command, select the number corresponding to /usr/bin/python3.10.

Verify the change by checking the version again:

|  |
| --- |
| python --version |

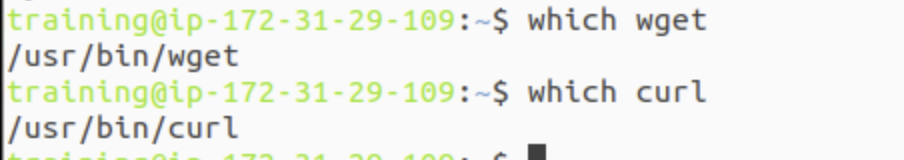


**wget or curl**

A command-line tool for downloading files. Most Ubuntu systems have one pre-installed.

Check for wget: which wget

Check for curl: which curl



# Project Structure

By the end of this lab, your project directory will look like this. Understanding this structure is key to managing larger projects.

|  |
| --- |
| ~/flink-lab-1/ ├── venv/ # The isolated Python virtual environment for this project │ ├── bin/ │ ├── lib/ │ └── ... └── hello\_flink.py # Your PyFlink job script |

# Part 1: Download and Set Up the Flink Cluster

We will script the download and setup process to avoid manual errors and ensure consistency.

## Step 1: Download, Unpack, and Relocate Flink

Open a terminal. This command block will download Flink 2.0.0, unpack it, and move it to a clean location in your home directory (~).

Define the Flink version we want to use

|  |
| --- |
| FLINK\_VERSION="2.0.0" |

Use wget to download the specific binary. Note the Scala 2.12 part.

|  |
| --- |
| wget https://dlcdn.apache.org/flink/flink-${FLINK\_VERSION}/flink-${FLINK\_VERSION}-bin-scala\_2.12.tgz |

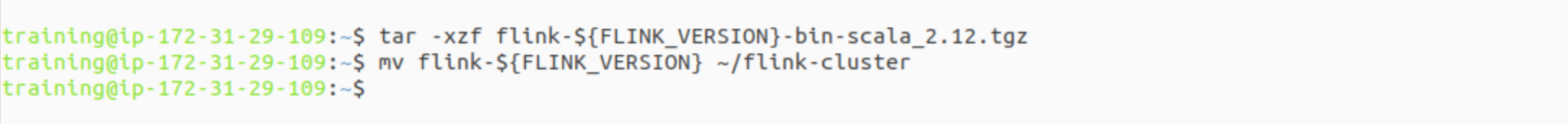


Unpack the downloaded archive

|  |
| --- |
| tar -xzf flink-${FLINK\_VERSION}-bin-scala\_2.12.tgz |

Move the directory to your home folder to ensure a clean, space-free path

|  |
| --- |
| mv flink-${FLINK\_VERSION} ~/flink-cluster |



## Step 2: Configure Your Shell Environment

To ensure the flink command is available in all your terminal sessions, you must set the FLINK\_HOME variable and add Flink's bin directory to your PATH. This is the most robust way to configure your system.

Add FLINK\_HOME and update the PATH in your shell's configuration file.  
This makes the setting permanent across all terminal windows.

|  |
| --- |
| echo '' >> ~/.bashrc echo '# Flink Environment Variables' >> ~/.bashrc echo 'export FLINK\_HOME=~/flink-cluster' >> ~/.bashrc echo 'export PATH=$PATH:$FLINK\_HOME/bin' >> ~/.bashrc |

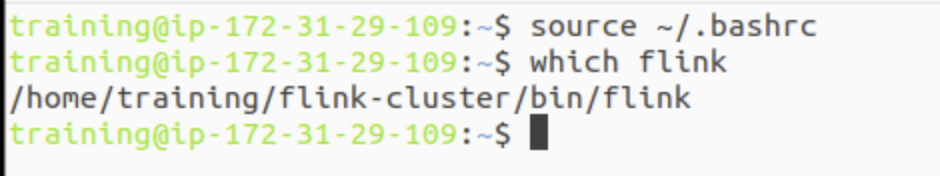


Apply the changes to your current terminal session

|  |
| --- |
| source ~/.bashrc |

Verify that the 'flink' command is now available from any directory

|  |
| --- |
| which flink |



**Note:** If you are using Zsh, replace ~/.bashrc with ~/.zshrc in the commands above.

# Part 2: Create the Python Project and Configure Flink

Now, we'll set up a dedicated Python environment for our job and tell Flink where to find its Python interpreter.

## Step 1: Create the Project Directory and Virtual Environment

It's best practice to create a new directory for each project.

Create and navigate to the lab directory in your home folder

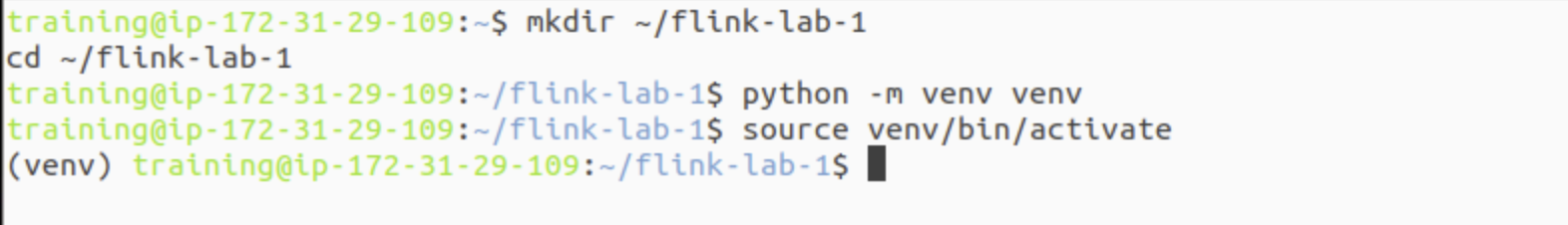
|  |
| --- |
| mkdir ~/flink-lab-1 cd ~/flink-lab-1 |

Create a Python virtual environment named 'venv'

|  |
| --- |
| python -m venv venv |

Activate the virtual environment. Your shell prompt will change.

|  |
| --- |
| source venv/bin/activate |

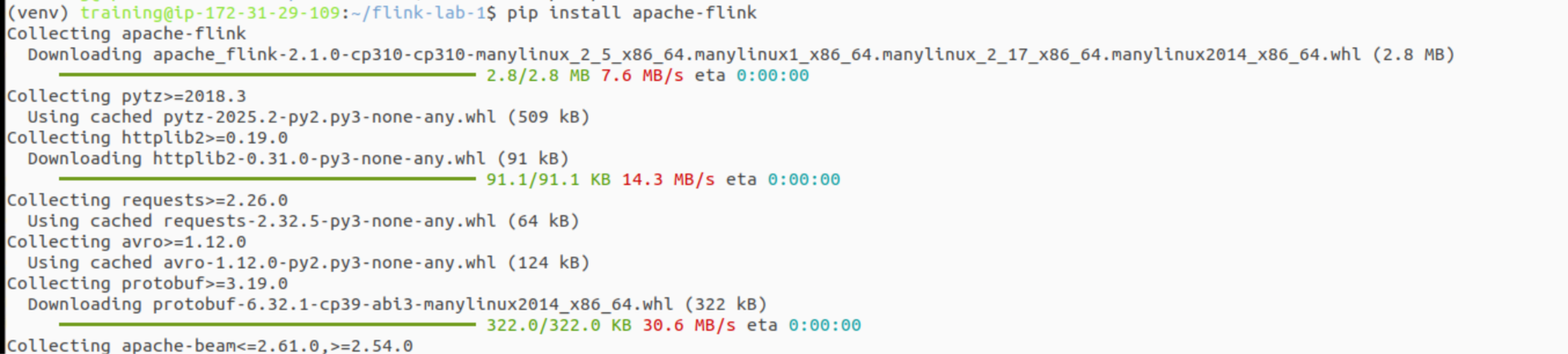
**

*What's Happening?* A virtual environment (venv) isolates the project's Python dependencies (apache-flink) from your global Python installation, preventing version conflicts.

## Step 2: Install apache-flink

With the venv active, install the necessary PyFlink library.

|  |
| --- |
| pip install apache-flink |



## Step 3: Get the Python Executable Path and Configure Flink

Flink needs the absolute path to the Python interpreter inside your virtual environment. We'll get this path and add it to Flink's configuration file programmatically.

Get the absolute path to the Python executable within the active venv

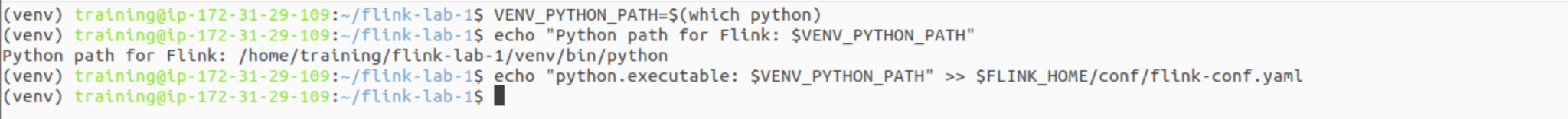
|  |
| --- |
| VENV\_PYTHON\_PATH=$(which python) |

Display the path to confirm it's correct (it should be inside ~/flink-lab-1/venv)

|  |
| --- |
| echo "Python path for Flink: $VENV\_PYTHON\_PATH" |

Now, use a command to append this configuration to Flink's config file.  
This is more robust than editing it by hand.

|  |
| --- |
| echo "python.executable: $VENV\_PYTHON\_PATH" >> $FLINK\_HOME/conf/flink-conf.yaml |



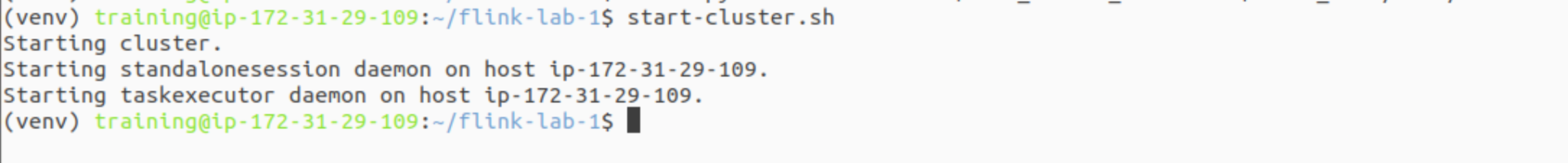
# Part 3: Start the Flink Cluster and Submit the Job

## Step 1: Start the Standalone Cluster

Because you configured your PATH, you can run this command from any directory.

Start the cluster using the globally available script

|  |
| --- |
| start-cluster.sh |



You should see output indicating the JobManager and TaskManager have started.

*What's Happening?*

* **JobManager:** The coordinator of the cluster. It receives jobs, plans their execution, and manages resources.
* **TaskManager:** The worker node. It executes the actual data processing tasks assigned by the JobManager.

## Step 2: Access the Flink Web UI

Open your web browser and navigate to <http://localhost:8081>. You should see the Flink Web Dashboard. Take a moment to look around.

## Step 3: Create the Python Script

Go back to your terminal in the ~/flink-lab-1 directory, where the (venv) is still active. Create a file named hello\_flink.py using VS Code or another editor, and add the following code:

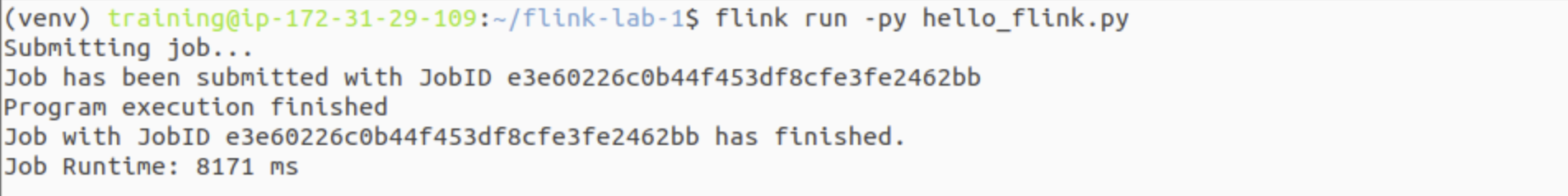
|  |
| --- |
| # hello\_flink.py import datetime from pyflink.datastream import StreamExecutionEnvironment from pyflink.datastream.functions import MapFunction  def main():  """  The main entry point for the Flink job.  """  env = StreamExecutionEnvironment.get\_execution\_environment()  # Set parallelism to 1 to ensure output is printed in order  env.set\_parallelism(1)   class AddTimestamp(MapFunction):  """  A simple MapFunction that adds a processing timestamp to each event.  """  def map(self, value):  processing\_time = datetime.datetime.now()  # The format includes milliseconds for clarity  return f"{value}, processed\_at: {processing\_time.strftime('%H:%M:%S.%f')[:-3]}"   print("Submitting job...")   # Create a simple data stream from a Python list  data\_stream = env.from\_collection(  collection=[  'event 1: Welcome to Flink',  'event 2: This is a simple job',  'event 3: Processing time is easy'  ]  )    # Apply the mapping function to transform the data  transformed\_stream = data\_stream.map(AddTimestamp())    # The print() sink is a simple way to see results during development.  # It prints the output to the TaskManager's standard out log.  transformed\_stream.print()    # Execute the job with a descriptive name  env.execute("hello\_flink\_processing\_time")  if \_\_name\_\_ == '\_\_main\_\_':  main() |

## Step 4: Submit the Job to the Cluster

Now, submit the job using the flink run command.

Use the globally available flink executable to run the Python script

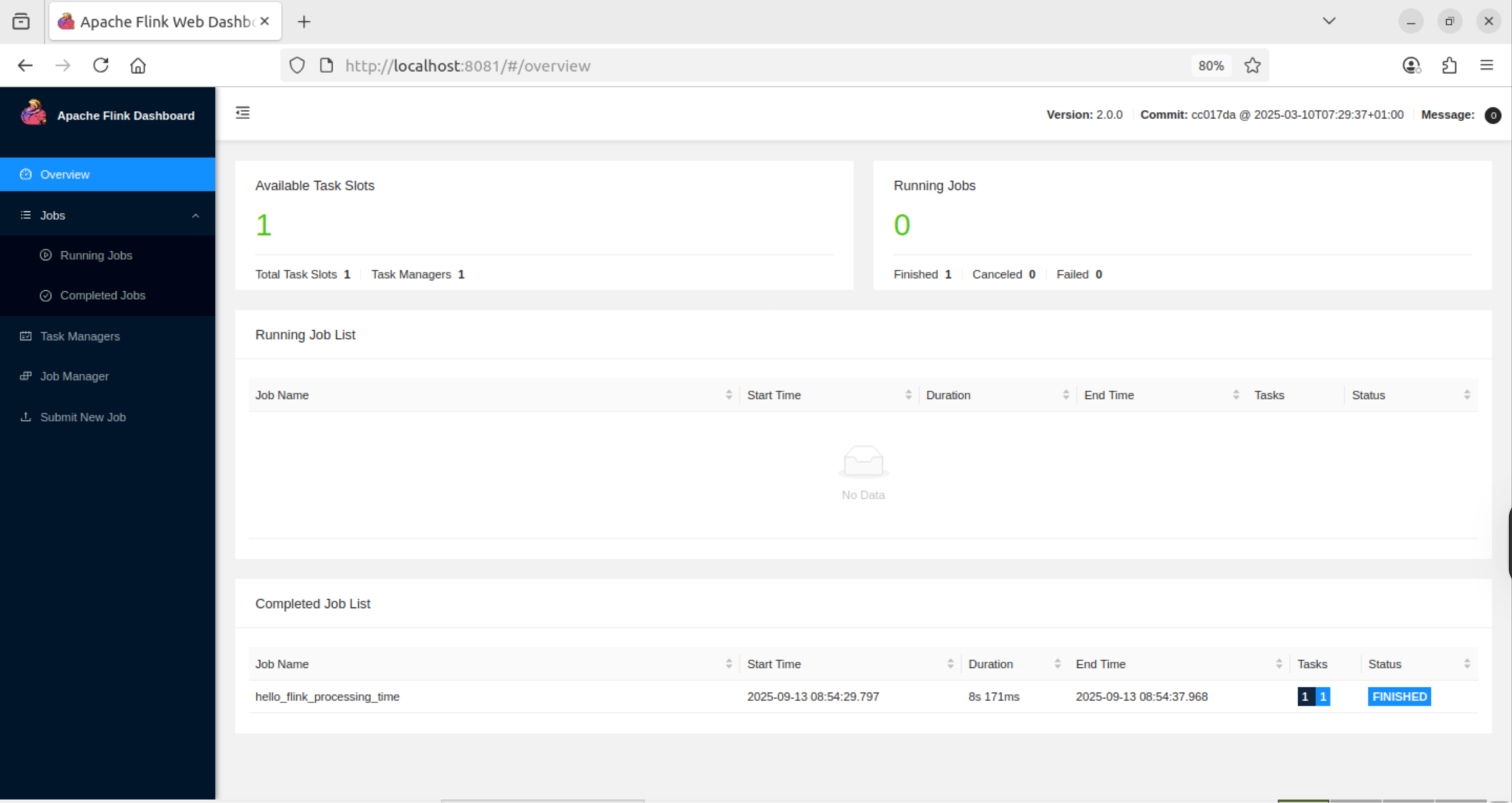
|  |
| --- |
| flink run -py hello\_flink.py |



# Part 4: Observe and Cleanup

## Step 1: Check the Web UI

Refresh the Flink dashboard at <http://localhost:8081>. Go to the "Completed Jobs" section. You should see your hello\_flink\_processing\_time job listed with a "FINISHED" status.



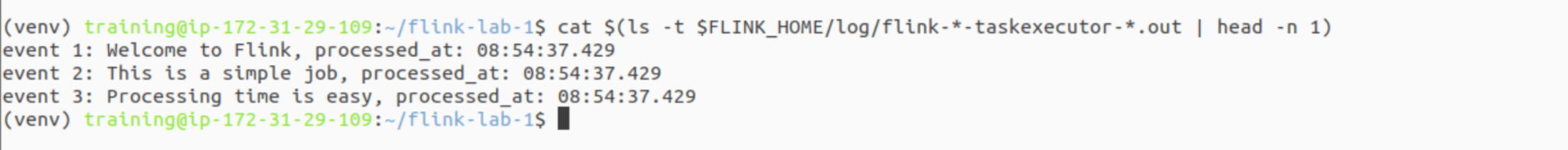
## Step 2: Find the Output

The print() sink writes its output to the TaskManager's .out log file. Here is a robust command to find and view the most recent one:

Find the most recent TaskManager .out log file and display its contents

|  |
| --- |
| cat $(ls -t $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | head -n 1) |

You should see your processed events, each with a timestamp, like this:



## Step 3: Cleanup

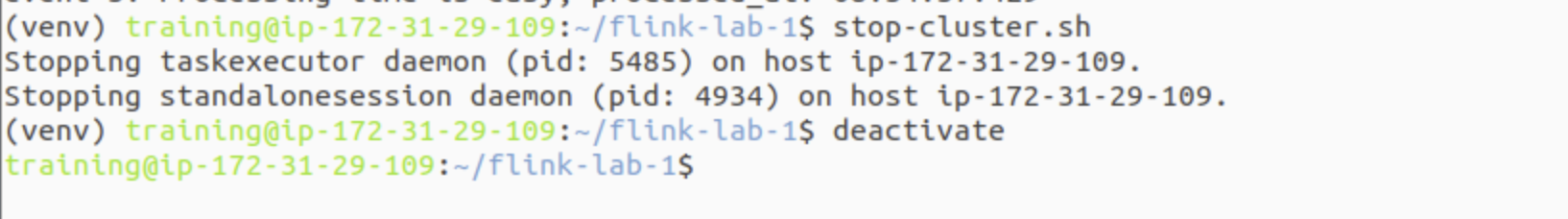
When you're finished, stop the cluster.

Stop the cluster

|  |
| --- |
| stop-cluster.sh |

To exit the Python virtual environment, simply run:

|  |
| --- |
| deactivate |



# Next Steps

Congratulations on running your first Flink job! You now have a robust local development setup. Here are some ideas to continue your learning:

* **Explore other Sources:** Modify your script to read from a file instead of a static collection. Look into env.from\_path('file:///path/to/your/file.txt').
* **Explore other Sinks:** Instead of print(), try writing the output to a file using transformed\_stream.sink\_to\_file().
* **Add more Transformations:** Experiment with other DataStream transformations like .filter() to select certain events or .flat\_map() to produce multiple results from one input.
* **Consult the Docs:** The official [Apache Flink Documentation](https://nightlies.apache.org/flink/flink-docs-stable/) is the best resource for in-depth information on all available APIs and features.

# Troubleshooting Common Issues

* **Port 8081 already in use:** Another service is using the Flink UI port. Stop the other service or change Flink's port in $FLINK\_HOME/conf/flink-conf.yaml by modifying rest.port: 8081.
* **Permission Denied when running scripts:** If you get a permission error, the scripts may not be executable. Run chmod +x $FLINK\_HOME/bin/\*.
* **Job Fails Immediately:** The most common cause is an incorrect Python path. Re-run the command from Part 2, Step 3 (echo "python.executable: ...") to ensure the correct path is in flink-conf.yaml.
* **java -version shows the wrong version:** Your JAVA\_HOME environment variable or system PATH may be pointing to an older Java installation. Ensure JDK 11 is the default Java on your PATH.