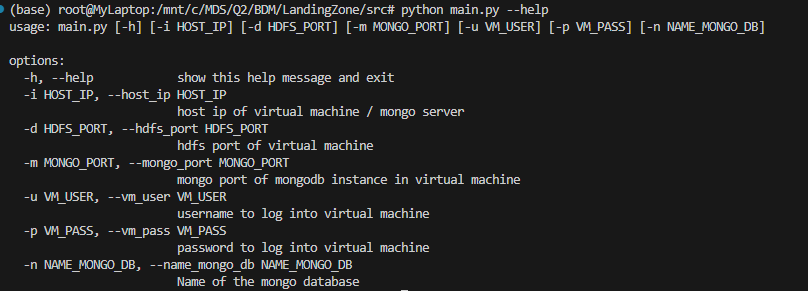
## QuickStart instructions

In order to work with the repository, we will need to have installed python3 in our case we used Python 3.11. First, we will have to install all dependencies by using the requirements.txt file in the root directory of the repository. After that we must have all the information, we want to load in the resources folder (not the unemployment data which is pulled by an API). Subsequently, all the programs can be processed through the main.py, we can modify the following parameters to maximize reusability as follows:



# Temporal Landing Zone

## Architecture

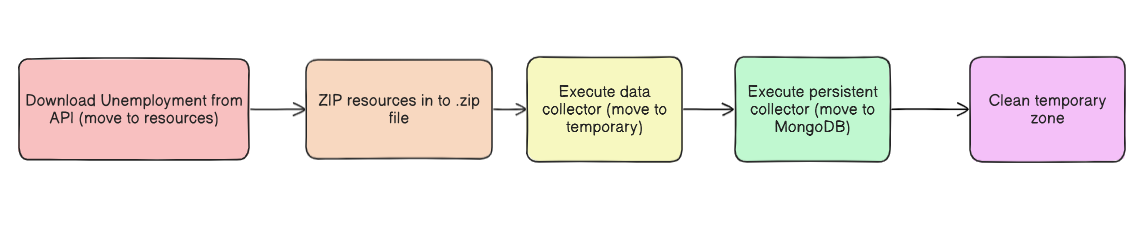
The heart of this architecture lies within the HDFS, residing on the virtual machine. This choice of HDFS as the foundation because of its robustness and distributed storage capability. HDFS stands out for its ability to handle various file types without necessitating modifications. This feature proves invaluable in managing source heterogeneity, HDFS accommodates diverse data formats effortlessly.

One of the key features of HDFS is automatic replication, ensuring data security and availability. In this architecture, HDFS replicates data 3 times which provides from fault tolerance. Additionally, data distribution across available machines occurs randomly.

The architecture is very simple to scale by simply adding extra nodes and increasing capacity of compute and storage although in this demo is not shown as we only use 1 VM.

The lowest level of abstraction in the whole architecture will be the file, not the row/tuple. So, we will move the data by files in the temporal zone. Storing original formats of source files will guarantee flexibility and resistance to changes in data sources over time.

The landing zone resides within the HDFS, accessible at /user/temporary. In order to copy the data in the landing zone we will download, locally, the unemployment information through an API, after that we will ZIP the whole contents and upload the information to the VM. Subsequently, we will copy the information from the zip file (uncompressed) to the Landing Zone. We can also see how it works in the following diagram:



## Assumptions

1. We assume that placing all files in an HDFS without a distributed setup would be impractical for our project
2. We assume that without the context of big data, employing an HDFS wouldn't be justified, as a fuller database leads to a more balanced distribution
3. We assume that this approach implies a single, distributed temporal landing zone (due to the the way of obtaining the data), reinforcing our decision-making process
4. We assume that given this is a temporary landing zone where no alterations to the data occur, files are ingested in their original file format

# Persistent Landing Zone

## Architecture

The provided code aims to automate the process of loading data from a virtual machine (VM) to both a Hadoop Distributed File System (HDFS) and a MongoDB database. This automation is particularly useful in environments dealing with large volumes of data and requiring efficient data management.

The workflow begins with the `**persistent\_collector**` function, serving as the main entry point. This function establishes a connection with the VM using the provided IP address, username, and password as arguments. Once the connection is established, a MongoDB instance is initiated in the VM to serve as the target database for subsequently loaded data.

After initializing MongoDB, a HDFS client is initialized using the `**hdfs**` library imported at the beginning of the code. This HDFS client is used to access and manipulate data stored in the distributed file system from the VM.

Next, a local directory named "**tracking\_data**" is created where a record of files that have already been loaded into the MongoDB database will be stored. This is done to avoid duplicating file loads and ensure data integrity.

Once the connection to the VM and the HDFS client are configured, the code iterates through files in the HDFS's temporary directory to find data to load into MongoDB. For each found file, its type (JSON or CSV) is determined, and it is loaded into the corresponding MongoDB database. If the file has already been loaded previously (according to the tracking record), it is skipped to prevent duplication.

The files are picked up and checked if they are a CSV or JSON. In the case that it is a JSON, it is directly imported into MongoDB, and if it is a CSV, it is read with pandas and converted from pandas to the .ToRecords format, which is basically a JSON format (implicitly transformed but not directly written). Then, this is sent to MongoDB.

The code also includes several helper functions such as `**initialize\_hdfs`, `initialize\_mongodb`, `load\_json`, `load\_csv`, `load\_tracking\_files**`, and `update\_tracking\_files`, used for specific tasks such as initializing HDFS and MongoDB clients, loading data from JSON or CSV files, and managing the tracking record of loaded files.

In summary, this code provides an automated and scalable solution for loading data from a VM to a MongoDB database, facilitating the management and analysis of large volumes of data in distributed environments.

## Assumptions

1. We assume that our system deals with significant volumes of data, necessitating distribution across multiple machines for efficient processing.
2. We assume that data distribution and processing occur across multiple machines rather than being centralized in one location, leveraging parallelism for improved performance.
3. We recognize the need to transform data from a temporal landing zone to a persistent zone, involving formatting and structuring for storage or further analysis.
4. We assume that Python scripts for data transformation are integrated into the virtual machines themselves, allowing parallel execution within the distributed environment for efficiency.
5. We assume that data transformation tasks are executed concurrently across distributed virtual machines, maximizing resource utilization and speeding up the transformation process.
6. From our perspective, the current implementation serves as a proof-of-concept rather than a fully practical solution, indicating potential for refinement and optimization
7. We assume that transformed data is likely stored in MongoDB, but the current approach of sending data to a local computer before transferring it introduces inefficiencies.
8. We recognize the importance of minimizing network transfers and reducing bottlenecks for optimizing system performance. Ideally, data storage and processing should occur within the distributed environment.

## Conclusion

The tracking method has been implemented using a .pickle file due to its ease of use and seamless integration with Python, as pickle is the standard method for saving Python objects. However, our loading process is not ideal, as Python lacks parallelization capabilities. To overcome this limitation, we should consider utilizing Spark or another parallelized framework. Unfortunately, our attempts to connect the master with our machine were unsuccessful, possibly due to version incompatibility. If data volumes were to increase, more optimal solutions would be necessary.

Using HDFS for the temporary landing zone and MongoDB for the persistent landing zone presents a practical albeit imperfect solution. This approach primarily addresses the Volume and Variety aspects of big data, providing flexible and distributed functionalities to meet diverse needs.