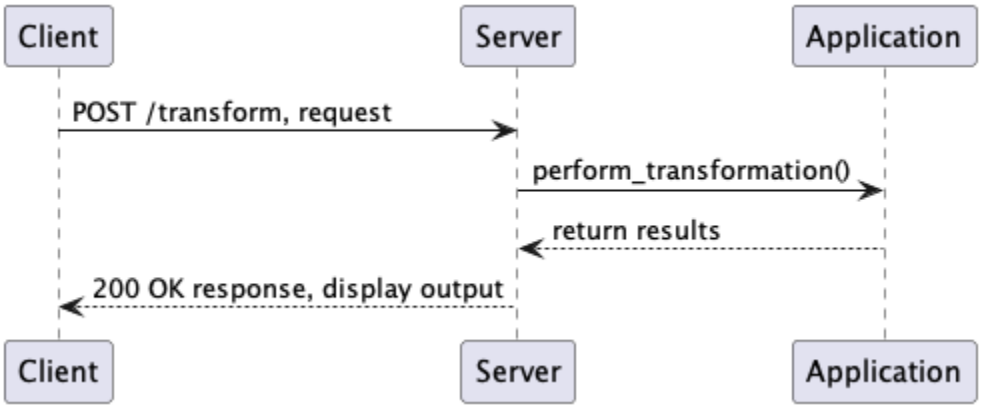
Crisp App Overview, Assumptions, Simplications, Next Steps

Overview

This take-home assessment is a web application developed in the Python programming language using the Flask web framework as a development server. The client-side part of the web app is written in HTML and CSS scripting languages and Jinja templating language to format the layout of webpages. The server-side of the web app is composed of two REST API endpoints that receive client requests for access to a homepage and transformation of uploaded input data. Additionally, the server-side uses the Pandas tool for transformation data wrangling and data manipulation. A simplified sequence diagram for a transformation request of the web app is included below.



Assumptions

- Given my interpretation of the requirements, the app has been tested to ingest a range of input csv file sizes from 830 B to 13 GB.

- Given the requirement of few external dependencies, the Python module Pandas was chosen as the tool for the transformation component. The benefits of Pandas over the csv module (or other Python modules) is that it is performant for large data sizes and row-wise wrangling due to vectorized operations (e.g. apply()). Pandas also offers a range of manipulation abilities needed for data wrangling and data analysis.

- All invalid rows in the input data are flagged while reading in the input data into a Pandas DataFrame and logged as a single log statement. Line numbers are included for each invalid row for troubleshooting.

- A rotating file handler was chosen for the Python logger given the requirement of large input data. A new log rotates approximately every 10 MB of logs with a maximum of 5 backup logs stored.

- The input data file extension allowed by the app is .csv only given the requirements. Other input data file extensions are rejected when a request for transformation is made.

- The external configuration file type is YAML for its shareability and ease of commenting. The config is modeled off of the transformation requirements, and the entire transformation process relies on the config file. Finally, the config file is uploaded (along with the input data file) when making a request for transformation. File extensions other than .yaml or .yml are rejected during a transformation request.

- Given the required JVM data types of the target fields in the transformed output, a decision was made to create equivalences to Python data types. For example, the equivalent Python data type to Java's BigDecimal is the decimal.Decimal data type.

- The app has been converted into a Python package for ease of sharing and use (like any other Python module/library).

Simplications

- Given that the requirements did not specify what to do with the transformed output, I made the decision for simplification by displaying a truncated view of the transformation results back to the client (via HTML), in addition to displaying the dimensions of the input data results and transformation results (via counts of data rows and columns). This outcome allows a visual confirmation that the transformation is performing as expected in addition to confirming that the dimensions of the input data and transformed data (given the limited transformation) should be equal in row count (while the transformation column count should be smaller than the input data column count).

- The config file structure is designed for ease of use and extension by non-technical users.

- The app's mechanism for allowing the ingestion of a wide range of input data sizes is to break the data out into chunks when reading it onto disk.

- Besides an index view, there are two additional views (transform and success). This is based on the requirements scope.

- Pandas read_csv() is inferring the data types of the input data file. This is an intentional choice since it's established that Pandas allocates generous memory for data types by default (because the tool prioritizes correctness over efficiency. Additionally, performance gains from explicit declaration, while helpful, are not significant enough to warrant use.

- Simplified exception handling was integrated into transformation.perform_transformation() but not into app.py.

Next Steps

If this task were a real project, homing in on what to do with the transformation output/how to share it would be the first objective. It is clear that downstream stakeholders would need to use/take action on the output and making the output usable would make that possible. My immediate instincts as a developer (if not requested by the stakeholder) would be to offer a download feature (so the user can directly control where the output goes) or upload the data to a known location (another server/etc.) Additional consideration on I/O and design would be needed and discussed though, since the requirements state that large input data is a use case.

Additionally, I would focus on doing what I could to speed up I/O. The chunking concept works well for ingesting (and outputting) large amounts of data relatively quickly (e.g. the app was tested for ingesting up to 13 GB of input data files), but I imagine performance (and storage drive capacity) would be improved on enterprise servers. I would look to improve memory usage and storage options. And finally, I would demonstrate if streaming was appropriate and cost-effective for the use case.

In terms of the development of the app, I would add necessary environment config files (Dev, UAT, Prod) that are present in typical web apps. Finally, on the front-end, I would add more user-friendly features to the scripting languages. Currently the client-side is functional but not eye-popping. Depending on the use case, a downstream stakeholder might require additional aesthetics.

In terms of the app's data usability, I would add features to allow for the reading and writing of additional input data file types (other than .csv) and consider integrations of input data sources like databases, cloud storage, and servers.

Additional documentation like a process flow describing the transformation, as well as a detailed sequence diagram would help in making the web app easier to support/extend/troubleshoot. In terms of production support, both a runbook and a

failure modes and effects analysis (FMEA) would serve to provide details about common functionality, troubleshooting, and document known/expected failures.

Finally, I would clean up the app code base by integrating proper Flask exception handling. A simplified approach was taken given my experience, but best practices could be reached with additional time.