PROJECT 1

Updated: 01/04/2025

BUILDING AN ETL PIPELINE IN OCAML

Gabriel Onishi | gabrielhso@al.insper.edu.br

The following is an informal document describing the process of building an ETL (Extract, Transform, Load) Pipeline in OCaml through the lens of a novel in functional programming. It should serve as a guide for building a similar project, as well as a sort of journal illustrating the challenges with dealing with new concepts and tools.

About the Project

This project aims to replicate a real-world scenario: an intern responsible for processing information from a database. In this scenario, the manager provides the intern with two csv files - one representing "items" data and one representing "orders" data - and needs the software to join them by client_id, revealing how much each client spent and paid on taxes.

As a graded academic project, this project had basic requirements:

1. *The project must be implemented in OCaml.*
2. *To compute the output, you must use map, reduce, and filter.*
3. *The code must include functions for reading and writing CSV files, which will introduce impure functions.*
4. *Separate impure functions from pure functions in the project files.*
5. *The input data must be loaded into a list of records.*
6. *The use of helper functions to load fields into a record is mandatory.*
7. *A project report must be written, explaining how each step was implemented. This serves as a guide for anyone who might recreate the project in the future. You must declare whether Generative AI was used in this report.*

This project also implemented the following additional requirements:

1. Read input data from a static file hosted online (exposed via HTTP).
2. Save the output data in an SQLite database.
3. Organize the ETL project using Dune.
4. Document all functions using docstring formatting.
5. Create comprehensive test files for all pure functions.

The full, original description of the project can be found on the project's GitHub.

Infrastructure

The first element of any project involves building its underlying infrastructure. In order to better isolate dependencies and improve developer experience, a Dev Container was used [1]. A Dev Container is a tool created by the Visual Studio Code (VSCode) team to enhance developer experience by allowing Docker Containers to run smoothly with VSCode, encapsulating the setup of extensions, debugging and formatting, among other functionalities. Using it guarantees that everything needed to run the project can be easily and locally configured.

The container was set to use an updated Ubuntu 22.04 image installed with the project dependencies installed - such as Opam, OCaml's package manager. As one of the project requirements dictated, Dune was used as the build system from the start, helping with compiler versioning (OCaml 3.17 was used) and package management.

This first step was not as trivial as it sounds. As the student was not familiar with the tool and didn't have extensive experience with Docker, there was a learning curve until the first version of the Dev Container would work as expected. The biggest challenges faced on this stage had to do with understanding how Opam and Dune worked, specifically about how Opam Switches handle OCaml versions and how to set up automatic code formatting using OCaml's Language Server Protocol (LSP). The Dev Container and Dune files were updated as the project evolved. A step by step tutorial on how to set up the project environment can be found on the repositories README.

Another bit of infrastructure used was setting up a Github Action hook for creating a better Continuous Integration (CI) experience. Although this was not a requirement, it was implemented to make sure the main branch was protected and always running with a buildable and tested codebase.

Project Organization

The project directory tree was organized as follows:

~~~
```
.
├── data
│   ├── prod
│   │   ├── input
│   │   └── output
│   └── test
│       ├── input
│       └── output
├── docs
│   └── images
└── etl
        ├── bin
        ├── lib
        │   ├── impure
        │   └── pure
        └── test
```
~~~

The data directory contains two subfolders: one for production use and another for testing purposes. Each of them contains its own input and output subfolder. Although the project was not used in actual production, this division was created to emulate what the good practices would dictate in a real-world scenario. All data in the "prod" subfolder (except ".gitkeep" files, to keep directory integrity) is ignored by GitHub, ensuring no sensitive files are publicly available. Test files, on the other hand, are randomized and are used by the CI pipeline, so they can be committed to the repository.

The "docs" directory contains all documentation, including this paper and the original project description.

Finally, the "etl" directory is the one encapsulating the app. It was created by Dune at the start of the project development and holds executables ("bin" subdirectory), libraries ("lib" subdirectory) and tests ("test" subdirectory). The lib subdirectory contains all the programming logic to be consumed by the executable, being divided into "impure" and "pure" modules. This segmentation is extremely useful for testing purposes. Pure functions are deterministic, meaning that the same input will always render the same output. This means that they are completely reliable once tested, leaving any errors that may appear almost exclusively to the modules found on the "impure" subfolder.

Development Steps

The first version of the project consists on a CSV to CSV transformation - that is, the document reads a local spreadsheet, executes data transformation and outputs another local spreadsheet.

The first steps involved had to do with understanding how OCaml and Dune work together with downloadable modules. The “csv” module was used to read and write data safely from and to files. As these functions interact with files outside of the code, the file involved with handling reading and writing (“io.ml”) was put under the impure directory.

The result is the representation of the csv as lists of lists of strings - each sublist represents a row and each value represents the value, with its position on the list representing its column. A sanity check was made when reading the file to make sure each row has the expected number of elements.

The next step involved parsing the elements of the lists as records. The record schemas, “order” and “item” are stored in “schemas.ml”, whilst the parsing logic is located on “csv_parser.ml”. Functions were created to iterate over the lists converting each value to their specific type, finally being combined on a “order_item” type - the result of the inner join of the two types by their “order_id”.

With the list of “order_items” set, it was time to group them by client. The logic for this part is stored under “pure/process.ml”. The result of the group by function is of type “order_summary”, also defined at “schemas.ml”.

The only thing left was to filter the resulting list by “status” and “order”. For such, the “Sys” module was used to capture the status and order from the arguments for the project execution on the terminal. The logic for checking if the argument is valid was also added to the “io.ml” file, as it interacts with an external component (terminal).

With the basic program up and running, it was time to make sure it worked. At first, inline tests were used, but to make sure all lines were covered, the test suite was later changed to Alcotest and bisect_ppx. The latter is a code coverage tool for OCaml, while the former is the framework for running unit tests that bisect_ppx depends on.

With the pure functions tested, it was time to work on the additional requirements. Docstrings, outputting to a SQLite database and reading from a static file hosted online were added in that order. The most challenging of those three was working with the “sqlite3” module, specially understanding how SQLite statements were written and could be reused.

Use of GenAI

While GenAI was not used in creating the program logic, it helped in many steps of the way. LLMs were used to:

- Understand concepts and answer questions about documentation
- Creating tests
- Writing docstrings
- Debugging

References and Resources

[Online]. Available: https://dune.readthedocs.io/en/stable/. Accessed on: Apr 3, 2025.
[Online]. Available: https://ocaml.org/. Accessed on: Apr 3, 2025.
[Online]. Available: https://ocaml.org/p/sqlite3/5.1.0/doc/Sqlite3/index.html. Accessed on: Apr 3, 2025.

[Online]. Available: https://ocaml.org/p/csv/latest/doc/Csv/index.html. Accessed on: Apr 3, 2025.

[Online]. Available: https://dev.realworldocaml.org/. Accessed on: Apr 3, 2025.