How to build Functional-ETL

Overview

This guide provides step-by-step instructions for building the Functional-ETL project, a functional ETL pipeline in OCaml that processes order data from CSV files hosted online using map, reduce, and filter. The project produces aggregated results and stores them in a database, showcasing functional programming principles and leveraging OCaml's capabilities.

Project Structure

The project uses Dune as a build system, organized into modular components:

▼ Bin

- **main.ml**: Contains the main application logic, orchestrating the ETL process.

▼ Lib

- **types.ml**: Defines OCaml record types for representing orders and order items.
- **extract.ml**: Extracts CSV data from the web and handles.
- **parse.ml**: Handles CSV parsing, transforming raw data into records.
- **transform.ml**: Applies data transformations using parse, map, filter, and reduce.
- **load.ml**: Reads data from CSV files and writes processed results.

▼ Test

- **test_parse.ml**: Unit tests for parsing logic.
- **test_transform.ml**: Unit tests for transformation logic.

▼ Building basic requirements

Step 1: Project Initialization

1. Create a new Dune project with the command:

```
dune init project functional-etl
```

2. Configure the project structure as described above.

Step 2: Data Extraction and Parsing

1. Read CSV data from local sources using `lib/extract.ml`.
2. Parse the data, transforming it into OCaml records with helper functions defined in `lib/parse.ml`.
3. Ensure type safety and readability by mapping CSV fields to record attributes.

Step 3: Computing Order Totals

1. Implement transformations in `lib/transform.ml` using map, filter, and reduce.
2. Calculate the total amount and taxes for each order. It can be done by iterating over each order and filtering out the items that do not correspond to current order.

Step 4: Writing Output to CSV

1. In `lib/load.ml`, create functions to write processed data to an output CSV file.

Step 5: Filtering Orders

1. Add a function to `lib/extract.ml` to handle user input from the CLI and apply filtering logic to `lib/transform.ml` based on order status and origin values selected by user.
2. Ensure the output meets the requirements specified by the user.

Step 6: Codebase Refactoring with Grok.AI

1. Use Grok.AI to refactor the codebase, focusing on functional decomposition, reducing duplication, and improving clarity.
2. Verify that the refactoring does not alter the intended behavior of the application.

▼ Building optional requirements

Step 1: Database Integration with SQLite3

1. Install the `sqlite3` library.
2. Create a database and insert order_totals data into it from `lib/load.ml`.

Step 2: Data Extraction from the Web

1. Extract data from the web (hosted on GitHub) using `lib/extract.ml` and Cohttp lib.
2. Load this data locally to leverage previous logic to read from local CSV files.

Step 3: Joining Orders and Order Items

1. Perform an inner join between orders and order items using a `filter_map` within a `concat_map` in `lib/transform.ml`.
2. Create the `item_joined_order` type in `lib/types.ml`.
3. Use the `Map` module to create a map with `order_id` as the key, iterating over each order item to update `total_amount` and `total_tax`.

Step 4: Computing Financial Records

1. Update the `order` type to include `order_date` as a `YYYY-MM` string.
2. Parse the order date and implement logic to compute financial records.
3. Store financial records in the database and CSV output using `lib/load.ml`.

Step 5: Refactoring with Monadic Error Handling

1. Refactor the codebase to use `Result` and `Option` OCaml features for error handling in a monadic way.

Step 6: Writing Docstrings with Grok.AI

1. Add docstrings to all functions following the OCamlDocs style using Grok.AI or other LLMs with OCaml knowledge.

Step 7: Writing Unit Tests

1. Write unit tests for all pure functions in `test/test_parse.ml` and `test/test_transform.ml`.

Step 8: Final Refactoring for Clarity

1. Refactor the entire codebase to focus on functional decomposition, reducing duplication, and enhancing clarity, just like the previous refactoring step.

Use of Generative AI

Both refactoring steps utilized Grok.AI to enhance code readability and maintainability. No generative AI was used in writing the initial implementation of the ETL pipeline. Ounit2 tests were developed with the assistance of Grok, doing tasks like defining the error messages and improving coverage.