Table of Contents

Introduction - Octopus challenge

Overview

The Electronic Data Interchange (EDI) project aims to facilitate the seamless exchange of structured business data between trading partners. EDI is widely used in various industries to automate and standardize data transmission, improving efficiency and reducing errors in business processes.

The aim of the challenge Octopus challenge is to create a Python software with a command line interface (CLI) that can be called with the path to an EDI 867_02 file (or files). The data for each reading in the file shall be extracted and stored in a database. The information available for each reading shall include usefull data that for a retail electric provider to find useful.

Objectives

- Parsing and Storing EDI Data: The project involves developing a system to parse incoming EDI messages and store the extracted data in a structured format.
- Database Integration: Implementing a database schema to efficiently store and manage EDI-related information, including files, segments, element details, and element data.
- Configuration Management: Loading and managing configuration files to customize the interpretation of different types of EDI messages.
- Logging and Error Handling: Implementing a robust logging system to track the processing of EDI messages and handling errors effectively.
- Testing and Validation: Incorporating testing mechanisms to validate the correctness of EDI parsing and database operations.
- Command-Line Interface: Developing a user-friendly command-line interface to initiate EDI parsing and interact with the system.

Project structure

The project consists of several key components:

- **File Handling**: Processing incoming EDI files and managing file-related information.
- **Segment and Element Details**: Structuring the database to represent the hierarchical nature of EDI messages, including segments and their element details.
- **Configuration Loading**: Loading and applying configuration files to adapt to different EDI standards and message types.
- **Logging Mechanism**: Implementing a comprehensive logging system to record processing details and errors.
- **Command-Line Interface**: Providing a user-friendly interface for users to interact with the system.

```
└── 🗁 octopus-challenge
    └── 🗁 edi_reader
        └── 🗁 conf
            └── 🗁 conf_model
                └── elements.json
                └── model.json
            └── edi_config.py
            └── __init__.py
        └── database.py
        └── edi_parser.py
        └── 🗁 exception
            └── error.py
            └── __init__.py
        └── 🗁 input
            └── 867_02_ex_01.edi
            └── 867_02_ex_02.edi
            └── 867_02_ex_03.edi
            └── 867_02_ex_04.edi
        └── logger.py
        └── main.py
        └── __init__.py
        └── __main__.py
    └── README.md
    └── 🗁 tests
        └── 🗁 input
            └── 🗁 conf
                └── elements.json
                └── model.json
            └── 🗁 input
                └── 867_02_ex_01.edi
                └── 867_02_ex_02.edi
                └── 867_02_ex_03.edi
                └── 867_02_ex_04.edi
                └── bad_id.edi
                └── incorrect_length.edi
                └── wrong_seg.edi
            └── 🗁 wrong_conf
                └── elements.json
```

```
            └── model.json
        └── test_edi_parser.py
        └── __init__.py
```

Database Table Creation SQL

This section outlines the SQL statements for creating database tables for an EDI (Electronic Data Interchange) system. The tables are designed to store information related to files, segments, element details, and element data.

File Table

```sql
CREATE TABLE IF NOT EXISTS FILE (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);
```

- `id`: Primary key for the file table.
- `name`: File name, must not be null.

Segment Table

```sql
CREATE TABLE IF NOT EXISTS SEGMENT (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type TEXT NOT NULL,
    name TEXT NOT NULL,
    req TEXT NOT NULL,
    max_uses INTEGER NOT NULL,
    notes TEXT,
    hierarchical_level INTEGER NOT NULL,
    file_id INTEGER,  -- Foreign key column referencing FILE.id
    FOREIGN KEY (file_id) REFERENCES FILE(id)
);
```

- `id`: Primary key for the segment table.
- `type`: Type of the segment.
- `name`: Name of the segment.
- `req`: Requirement status.
- `max_uses`: Maximum number of uses.
- `notes`: Additional notes (from specification documentation).
- `hierarchical_level`: Level in the hierarchical structure.
- `file_id`: Foreign key referencing FILE.id

Element Detail Table

```
CREATE TABLE IF NOT EXISTS ELEMENT_DETAIL (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type TEXT NOT NULL,
    name TEXT NOT NULL,
    req TEXT NOT NULL,
    data_type TEXT NOT NULL,
    length_min INTEGER NOT NULL,
    length_max INTEGER NOT NULL,
    segment_id INTEGER,  -- Foreign key column referencing SEGMENT.id
    FOREIGN KEY (segment_id) REFERENCES SEGMENT(id)
);
```

- `id`: Primary key for the element detail table.
- `type`: Type of the element detail.
- `name`: Name of the element detail.
- `req`: Requirement status.
- `data_type`: Data type of the element detail.
- `length_min`: Minimum data field length.
- `length_max`: Maximum data field length.
- `segment_id`: Foreign key referencing SEGMENT.id.

Element Data Table

```
CREATE TABLE IF NOT EXISTS ELEMENT_DATA (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type TEXT NOT NULL,
    name TEXT NOT NULL,
    req TEXT NOT NULL,
    data_type TEXT NOT NULL,
    length_min INTEGER NOT NULL,
    length_max INTEGER NOT NULL,
    element_detail_id INTEGER,  -- Foreign key column referencing
ELEMENT_DETAIL.id
    FOREIGN KEY (element_detail_id) REFERENCES ELEMENT_DETAIL(id)
);
```

- `id`: Primary key for the element data table.
- `type`: Type of the element data.
- `name`: Name of the element data.
- `req`: Requirement status.
- `data_type`: Data type of the element data.
- `length_min`: Minimum length.
- `length_max`: Maximum length.
- `element_detail_id`: Foreign key referencing ELEMENT_DETAIL.id.

These SQL statements define the structure of the database tables for storing information related to files, segments, element details, and element data in an EDI system.

Future Enhancements

- **EDI Standards Support**: Expanding support for additional EDI standards and accommodating industry-specific variations.
- **User Interface**: Developing a graphical user interface for easier interaction and monitoring.
- **Advanced Validation**: Implementing advanced validation checks for EDI data integrity.
- **API Integration**: Enabling integration with external systems through APIs for seamless data exchange.

Getting Started

In this chapter, we will guide you through the necessary steps to get started with the EDI parser.

Installation

To install the EDI parser on your system, follow these steps:

Prerequisites

Before proceeding with the installation, ensure that you have the following prerequisites installed on your system:

- Python ^3.10
- Poetry (Python dependency management)

Installation Steps

1. Navigate to the project directory:

   ```
   cd octopus-challenge
   ```

2. Install Dependencies:

   Run the following command to install the project dependencies specified in **edi_reader.toml**:

   ```
   poetry install
   ```

3. Running the Project

   Once you have installed the dependencies, you can run project using the Poetry command:

   ```
   poetry run python edi_reader <edi_config> <file_or_folder_path>
   ```

For example :

```
poetry run python edi_reader edi_reader\conf\conf_model edi_reader\input
```