Contest Log Analyzer - Programmer's Guide

Version 0.22.0-Beta

Date: 2025-07-29

1. Introduction

This guide is for developers who want to understand, maintain, or extend the Contest Log Analyzer. It provides an overview of the project's architecture, explains the core data flow, and gives step-by-step instructions for common development tasks like adding new reports, contest definitions, and scoring logic.

2. Project Architecture & Directory Structure

The project is designed to be highly modular, separating data, processing, and presentation. The recommended directory structure is as follows:

- Contest-Log-Analyzer/ (Project Root)
    - main_cli.py: The main command-line interface (CLI) script and the entry point for the application.
    - contest_tools/: The core Python application package containing all the processing logic.
        - cabrillo_parser.py: Contains low-level functions for reading and parsing the standard Cabrillo log file format.
        - contest_log.py: Defines the ContestLog class, the central object that holds all data and metadata for a single, fully processed log.
        - log_manager.py: Defines the LogManager class, which handles loading and managing one or more ContestLog instances for comparative analysis.
        - contest_definitions/: A data-driven package containing JSON files that define the rules, multipliers, and exchange formats for each supported contest.
        - core_annotations/: Contains modules for universal data enrichment that applies to most contests, such as country lookup (get_cty.py) and Run/S&P classification (run_s_p.py).

        - contest_specific_annotations/: Contains modules with logic unique to a specific contest, such as the scoring rules for CQ WW or CQ WPX.
        - reports/: The "plug-and-play" reporting system. Each Python file in this directory is a self-contained report generator that is automatically discovered by the program.
    - Logs/ (Recommended User Directory)
        - This directory is the recommended location for storing your raw Cabrillo log files. A good practice is to organize them by year and then by contest name.
        - YYYY/
            - Contest-Name/ (e.g., cq-ww-cw/)
                - k3lr.log
                - kc1xx.log
    - reports_output/ (Generated Directory)
        - This directory is automatically created by the program to store all generated reports and charts. The structure is organized to keep results from different contests separate.
        - YYYY/
            - Contest_Name/ (e.g., CQ-WW-CW/)
                - text/: Contains all generated text reports (.txt).
                - plots/: Contains all generated line graphs (.png).
                - charts/: Contains all generated bar charts (.png).

3. Core Concepts & Data Flow

Understanding the data flow is key to working with the project. The process follows a clear pipeline:

1. Loading: The LogManager is called by main_cli.py with a path to a raw Cabrillo log.

2. Definition: The manager reads the CONTEST: header from the file to identify the contest (e.g., "CQ-WW-CW"). It then loads the corresponding JSON file (e.g., cq_ww_cw.json) into a ContestDefinition object.
3. Parsing: The cabrillo_parser uses the rules from the ContestDefinition object to parse the raw text file into a structured pandas DataFrame and a metadata dictionary.
4. Instantiation: A ContestLog object is created to hold the parsed DataFrame and metadata.
5. Annotation: The ContestLog object's apply_annotations() method is called. This is a crucial step where the raw data is enriched:
    - Core Annotations are applied first (Country/Zone lookup, Run/S&P classification).
    - Contest-Specific Annotations are applied next (QSO point calculation, multiplier identification).
6. Reporting: The final, fully-annotated ContestLog object(s) are passed to the requested report generator, which performs its analysis and saves the output.

4. Extending the Analyzer

How to Add a New Report

The reporting system is designed to be "plug-and-play." To add a new report, you simply create a new Python file in the contest_tools/reports/ directory. The system will automatically discover it.

The Report Interface

Every report file must contain a class named Report that inherits from ContestReport. This base class ensures a consistent structure. You must implement the following:

- report_id (property): A unique, machine-readable string for your report (e.g., band_summary). This is what the user types on the command line.
- report_name (property): A human-readable name for your report (e.g., "QSOs per Band Summary").
- report_type (property): Must be one of text, plot, or chart. This determines the output subdirectory.

- generate(self, output_path: str, **kwargs) (method): This is where your main logic goes. It receives the base output path and a dictionary of optional arguments. It is responsible for saving its own output file(s) and must return a string confirmation message.

Step-by-Step Guide

1. Create Your Report File: In the contest_tools/reports/ directory, create a new Python file (e.g., text_my_new_report.py).
2. Use a Template: Copy the contents of an existing report file (like text_summary.py) into your new file to get the correct structure.
3. Customize Your Report Class:
    - Update report_id and report_name.
    - Set the report_type.
    - Write your analysis logic in the generate method.
        - Access the fully processed logs via self.logs.
        - Get the pandas DataFrame with log.get_processed_data().
        - Get the header data with log.get_metadata().
        - Safely access optional arguments via kwargs.get('arg_name', default_value).
4. Run It! No other files need to be edited. The system will automatically discover your report. You can see it listed by running python main_cli.py --report and generate it with python main_cli.py --report your_new_id ....

How to Add a New Contest Definition

If you want to add support for a contest not currently defined, you only need to create a new JSON file.

1. Create JSON File: In contest_tools/contest_definitions/, create a new file (e.g., arrl_dx_cw.json).
2. Define contest_name: Add the exact name from the Cabrillo CONTEST: header (e.g., "contest_name": "ARRL-DX-CW",).

3. Define Exchange Parsing: Under exchange_parsing_rules, create an entry for the contest name. Provide a regex to capture the exchange fields and a list of groups to name them.
4. Define Multipliers: Add a multiplier_rules list to define the multipliers for the contest (e.g., states, countries). Specify the source of the multiplier data.
5. (Optional) Add Scoring: If the contest requires custom scoring, see the next section.

How to Add New Contest-Specific Scoring

The system can dynamically load scoring logic for any contest.

1. Create Scoring File: In contest_tools/contest_specific_annotations/, create a new Python file whose name matches the contest's JSON file (e.g., arrl_dx_scoring.py).
2. Implement calculate_points: The file must contain a function with the signature calculate_points(df: pd.DataFrame, my_call_info: Dict[str, Any]) -> pd.Series.
3. Write Logic: Inside this function, write the logic to calculate the point value for each QSO in the input DataFrame (df). The my_call_info dictionary provides the operator's own location data, which is often needed for scoring.
4. Automatic Discovery: The ContestLog class will automatically find and execute this function during the annotation process based on the contest name.