Technical Documentation: we_mechload_viewer Architecture

Introduction

This document provides a deep dive into the internal architecture of the we_mechload_viewer application (modular_version_v2). It details the responsibility of each module and, crucially, maps out the **signal and slot communication network** that allows the application's components to interact.

The architecture follows a **Model-View-Controller (MVC)** pattern. Think of it as a well-organized system with specialized roles: the **Model** manages the data, the **View** displays it, and the **Controller** acts as the go-between. Understanding this separation is key to understanding the codebase.

a Model-View-Controller diagram resmi

Part 1: Application Bootstrap

main.py

- **Role:** The single entry point for the application. It's the "general contractor" that assembles everything.
- **Responsibilities:**
  1. Initializes the QApplication.
  2. Instantiates all the core components:
     - MainWindow (the main View).
     - DataManager, ConfigManager (the Model).
     - ActionHandler, PlotController (the Controllers).
  3. **Dependency Injection:** It passes instances of the Model and View to the Controllers. This is a critical step that **decouples** the components; the controllers don't create their dependencies, they receive them.
  4. Starts the Qt event loop.

```
# main.py - Simplified
app = QApplication(sys.argv)

# 1. Instantiate all components
main_win = MainWindow()
data_manager = DataManager()
config_manager = ConfigManager(config_path)
plotter = main_win.plotter # Get plotter from MainWindow
```

```
# 2. Instantiate controllers and inject dependencies
action_handler = ActionHandler(main_win, data_manager, config_manager)
plot_controller = PlotController(main_win, data_manager, plotter)

# 3. Show the main window and run
main_win.show()
sys.exit(app.exec_())
```

Part 2: The Model Layer (Data and Logic)

The Model layer is responsible for managing data and business logic. It **knows nothing about the UI**.

app/data_manager.py

- **Role:** The central repository for application data. It's the **single source of truth**.
- **Key Attributes:** self.df (the main pandas DataFrame), self.df_compare (the second DataFrame for comparisons).
- **Key Methods:** load_data(), get_dataframe(), set_dataframe().
- **Signals:** It defines a custom signal, data_loaded, which it emits after a new file has been successfully loaded into the DataFrame. This allows other components (like controllers) to react to new data becoming available.

app/config_manager.py

- **Role:** Handles saving and loading user settings and application configuration from a JSON file.

app/analysis/data_processing.py

- **Role:** A library of **pure, stateless functions** for all numerical analysis. This is where the "heavy lifting" happens.
- **Responsibilities:** Contains all calculation logic, such as:
  - calculate_fft()
  - calculate_rolling_envelope()
  - apply_butterworth_filter()
- **Interaction:** These functions are called exclusively by the **Controllers**. They take data as input and return results, without modifying any application state directly.

app/analysis/ansys_exporter.py

- **Role:** A specialized module for exporting data to the Ansys APDL format.
- **Interaction:** Called by the ActionHandler controller when the user triggers an export action.

Part 3: The View Layer (UI)

The View layer is responsible for everything the user sees. Its components are designed to be "dumb"—they display information and **emit signals** when the user interacts with them, but they don't contain processing logic.

app/main_window.py

- **Role:** The top-level QMainWindow that acts as a container for all other UI elements.
- **Responsibilities:**
  1. Initializes and lays out the main UI structure (docks, tabs, menu bar).
  2. Instantiates all the individual UI tabs from the app/ui/ directory.
  3. Instantiates the Plotter widget.
  4. Provides accessors (.get_tab_single_data(), etc.) so the controllers can access the UI components to connect signals.

app/plotting/plotter.py

- **Role:** A specialized QWidget whose only job is to render a Plotly figure.
- **Responsibilities:** It has one primary slot/method: plot(fig), which takes a Plotly figure object, saves it to a temporary HTML file, and loads that file into its web browser view.

app/ui/*.py (The Tabs)

These modules are the primary source of user-interaction **signals**.

- **tab_single_data.py**, **tab_compare_data.py**:
  - **Purpose:** Display data statistics and allow column selection.
  - **Signals Emitted:**
    - self.column_selector.currentTextChanged
    - self.plot_checkbox.stateChanged
    - self.filter_checkbox.stateChanged
    - self.butterworth_order.valueChanged
    - self.butterworth_cutoff.valueChanged
- **tab_time_domain_represent.py**:
  - **Purpose:** Configure time-domain plot representations like rolling FFTs and envelopes.
  - **Signals Emitted:**
    - self.checkbox_roll_fft.stateChanged
    - self.spin_box_roll_fft_window.valueChanged
    - self.spin_box_roll_fft_freq.valueChanged
    - self.checkbox_roll_env.stateChanged
    - self.spin_box_roll_env.valueChanged
- **directory_tree_dock.py**:
  - **Purpose:** A file browser dock.

- **Signals Emitted:** self.tree.doubleClicked (emits a QModelIndex).

Part 4: The Controller Layer (The "Glue")

The Controllers are the heart of the application's interactive logic, connecting the View's signals to the Model's functions. They **listen for user actions** and orchestrate the application's response.

app/controllers/action_handler.py

- **Role:** Manages general, non-plotting actions like file I/O and settings.
- **Signal-Slot Connections (in __init__)**:
  - main_win.open_action.triggered **connects to** self.handle_open_file
  - main_win.open_folder_action.triggered **connects to** self.handle_open_folder
  - main_win.directory_dock.tree.doubleClicked **connects to** self.handle_tree_selection
  - data_manager.data_loaded **connects to** main_win.update_ui_after_data_load *(Controller connects a Model signal to a View slot)*
  - main_win.settings_tab.save_settings_button.clicked **connects to** self.handle_save_settings
  - main_win.interface_data_tab.export_button.clicked **connects to** self.handle_export_to_ansys

app/controllers/plot_controller.py

- **Role:** Manages all logic related to updating the plot. This is the most complex controller because many different UI inputs can trigger a plot refresh.
- **Signal-Slot Connections (in __init__)**:
  - **From tab_single_data & tab_compare_data**:
    - tab.column_selector.currentTextChanged **connects to** self.update_plot
    - tab.plot_checkbox.stateChanged **connects to** self.update_plot
    - tab.filter_checkbox.stateChanged **connects to** self.update_plot
    - tab.butterworth_order.valueChanged **connects to** self.update_plot
    - tab.butterworth_cutoff.valueChanged **connects to** self.update_plot
  - **From tab_time_domain_represent**:
    - tab_time.checkbox_roll_fft.stateChanged **connects to** self.update_plot
    - tab_time.spin_box_roll_fft_window.valueChanged **connects to** self.update_plot
    - tab_time.spin_box_roll_fft_freq.valueChanged **connects to** self.update_plot
    - tab_time.checkbox_roll_env.stateChanged **connects to** self.update_plot
    - tab_time.spin_box_roll_env.valueChanged **connects to** self.update_plot
  - **From data_manager (Model)**:
    - self.data_manager.data_loaded **connects to** self.update_plot
- **Core Logic (update_plot slot)**: This central slot is the destination for almost every plot-related signal. When triggered, it executes the following sequence:
  1. Gathers the current configuration from **all relevant UI tabs** (which columns are selected, are filters active, what are the FFT settings, etc.).

2. Gets the raw DataFrame from the DataManager.
3. If required, it calls functions in data_processing.py to perform filtering, FFT calculations, etc., on the raw data.
4. Constructs a Plotly Figure object with the final data traces.
5. Calls self.plotter.plot(fig) to render the result.