

We used this server and client architecture to create our DSS application.

**Server:**

The server basically initialises an SEMPR core first as we are using the already proposed SEMR architecture as a backbone in our application. Semantic mapping framework as a tool to create a general environment model from which different representations can be drawn on demand.

## **Imports and Dependencies**

The application relies on several external libraries and custom modules:

* **Flask**: Web framework for handling HTTP requests.
* **semprpy**: Python bindings for the SEMPR reasoning engine.
* **json, re, os, glob**: Standard libraries for data handling.
* **datetime, timedelta, date**: Date and time manipulation.
* **pandas**: Data manipulation and analysis.
* **Custom Modules**:
  + - weather: Fetches weather data from an API.
    - weat1: Additional weather data processing.
    - read: Reads and preprocesses weather data.

**Pipeline Flow**

### **1. Initialize SEMPR Core**

* **Purpose**: Initialize the SEMPR reasoning core with a specified database directory where the field information has been stored ("db").
* **Actions**:
  + - Creates a Core object, which manages entities and reasoning processes.
    - Loads necessary plugins for the reasoning engine.

### **2. Load Reasoning Rules**

* **Purpose**: Load reasoning rules from a file and add them to the SEMPR core.
* **Actions**:
  + - Reads the rules from "rules/gerrit.rules".
    - Adds the rules to the core using core.addRules().
    - Performs an initial inference to process the rules.

### **3. Extract Area Codes**

#### **Function: extract\_area\_codes\_from\_folder(folder\_path)**

* **Purpose**: Extract area codes from JSON files (field files) in the specified folder.
* **Actions**:
  + - Iterates over all .json files in the "db/" directory.
    - For each file, extracts the agnw:Areacode value.
    - Compiles a list of area codes.

### **4. Fetch and Process Weather Data**

* **Purpose**: For each area code, fetch and preprocess weather data.
* **Actions**:
  + - Fetches the past two days weather data for the current date and area code using weather.station\_example().
    - Fetches the next seven days weather data for the current date and area code using weat1.station\_example().
    - Reads and process the weather data recived using read.read\_weather().

### **5. Create Weather Data Entities**

* **Purpose**: Convert weather data into entities that can be processed by SEMPR.
* **Actions**:
  + - Reads the weather data from 'input\_weather\_tm.csv'.
    - Filters data within a date range (from three days ago to seven days later).
    - For each record, creates a sempr.Entity() and a TriplePropertyMap (tpm1).
    - Sets properties like area code, date, temperature, rain, and sunshine.
    - Adds each entity to the SEMPR core.

### **6. Create Date Entity**

* **Purpose**: Create an entity representing the current date.
* **Actions**:
  + - Creates a Date entity with the type agnw:Date.
    - Sets the property agnw:Today to the current date.
    - Adds the entity to the SEMPR core.

### **7. Perform Inference**

* **Purpose**: Run the reasoning engine to process the new entities and generate inferences.
* **Actions**:
  + - Executes the reasoning process based on the entities and rules added to the core.
    - Updates the inference state with new WMEs.

### **8. Clean Up Temporary Files**

* **Purpose**: Remove temporary files generated during the process to clean up the workspace.
* **Actions**:
  + - Identifies all files in the "db/" directory that start with 'Entity'.
    - Deletes each identified file.

### **9. Retrieve and Filter Inferred WMEs**

* **Purpose**: Extract relevant inference results and present them in a user-friendly format.
* **Actions**:
  + - Retrieves all WMEs from the inference state.
    - Filters WMEs that match specific patterns (agricultural recommendations that we want to show in the home page).
    - Cleans up the strings for readability.
    - Saves the filtered WMEs to 'wmes1.json'.
    - Returns the filtered WMEs as a JSON response.

## **SERVER API Endpoints**

### **GET /**

* **Description**: Triggers the entire pipeline flow, from initializing the core to returning filtered inference results.
* **Response**: JSON array of filtered WMEs representing recommendations or significant inferences.

### **GET/POST /fwmes**

* **Description**: Returns all WMEs from the inference state without filtering.
* **Response**: JSON array of all WMEs.

### **GET/POST /explain**

* **Description**: Provides a detailed explanation for a specific inference result.
* **Request Parameter**: row - the specific WME string to explain.
* **Response**: JSON object containing the explanation of how the inference was derived.

**CLIENT**

This PyQt5 application serves as a graphical user interface (GUI) for the agricultural support system. It allows users to:

* View and interact with maps of agricultural fields.
* Fetch and display recommendations from a backend Flask server.
* Add new fields by drawing polygons on a map.
* Manage field attributes and values.
* Visualize explanations for recommendations using sunburst plots.
* Edit and save configuration files.

**Imports and Dependencies**

The application relies on several external libraries and custom modules:

* **Standard Libraries**:
  + - os, sys, re, json, io, threading, glob
* **Networking**:
  + - requests: For HTTP requests to the backend server.
* **PyQt5 Modules**:
  + - QtWidgets, QtGui, QtCore, QtWebEngineWidgets
* **Folium**:
  + - folium, folium.plugins.Draw: For map creation and drawing tools.
* **Custom Modules**:
  + - haxx: For generating sunburst plots (for creating plots).
    - create\_field: For adding fields to the database (for creating a new field).

## **Pipeline Flow**

### **1. Main Components**

* **MainWindow Class**: Central class that handles the GUI, user interactions, and data fetching.
* **Buttons and Navigation**: Buttons connected to methods that switch between different pages.
* **Map Integration**: Uses Folium and QWebEngineView to display and interact with maps.
* **Backend Communication**: Fetches data from a Flask server running on http://127.0.0.1:5000/.
* **Data Visualization**: Generates sunburst plots for explanations using the haxx module.

### **2. Initialization**

* **Load UI**: The UI layout is loaded from an external .ui file (interface.ui).
* **Set Initial Page**: The application starts on the home page.
* **Set Initial Button Style**: The home button is highlighted to indicate the current page.
* **Connect Button Signals**: Navigation buttons are connected to their respective

### **3. User Interface Setup**

* **Initialize Buttons and Widgets**: Set up buttons, text browsers, labels, and other widgets.
* **Set Button Styles**: Customize the appearance of buttons using stylesheets.
* **Initialize Table Widgets**: Set up tables to display data fetched from the backend.

### **4. Map Integration**

* **Load Existing Fields**: Read JSON files from the db directory to retrieve existing field polygons.
* **Create Folium Map**: Initialize a Folium map centered on a default location.  
  python
* **Add Polygons to Map**: For each field, extract coordinates and add a polygon to the map.
* **Save Map with Global Variable**: Modify the HTML to include a global map variable, enabling interaction via JavaScript.
* **Display Map in Application**: Use QWebEngineView to load the generated map.html file.
* **Add Field Buttons**: Create buttons for each field to allow focusing on specific polygons.

### **5. Fetching Data from Backend Server**

* **Fetch WMEs Data**: Retrieve filtered working memory elements from the backend server.
* **Fetch Full WMEs Data**: Retrieve all working memory elements from the backend server.

### **6. Populating Tables with Data**

* **Populate Table with Filtered WMEs**: Display the filtered WMEs in a table widget.  
  **Populate Table with Full WMEs**: Display all WMEs in another table widget.

**7. Field Management**

* + **Initialize Field Addition Interface**: Set up the interface for adding new fields.
  + **Add Attributes and Values**: Allow users to select attributes and enter corresponding values for a new field.
  + **Draw Polygon on Map**: Enable users to draw a polygon representing the new field.
  + **Capture Polygon Coordinates**: Extract the coordinates of the drawn polygon.
  + **Update Database**: Send the new field data to the backend for updating the database.
  + **Refresh Map and Data**: After adding a new field, refresh the map and fetch updated data.

### **8. Explanation and Visualization**

* **Handle Cell Clicks**: When a cell in the WMEs table is clicked, show the "Explain" button.
* **Fetch Explanation**: Send a request to the backend server to get an explanation for the selected WME.
* **Generate Sunburst Plot**: Use the haxx module to generate a sunburst plot based on the explanation.
* **Display Plot**: Open a new window to display the generated plot.
* **Show HTML Plot in QWebEngineView**: Load the HTML plot in a QWebEngineView.

### **9. Settings and Configuration**

* **List Configuration Files**: Display a list of files in the db directory for editing.
* **Edit Selected File**: Load the content of the selected file into a text editor.
* **Save Edited File**: Save changes made to the file back to the disk.
* **Refresh Data After Saving**: Fetch updated data from the backend after saving the file.

### **10. Event Handling and Navigation**

* + - **Toggle Left Menu**: Show or hide the left navigation menu.  
      Python code  
      self.menuBtn.clicked.connect(self.toggle\_left\_menu)
    - **Animate Menu Button**: Provide visual feedback when the menu button is clicked.  
      Python code  
      self.menuBtn.clicked.connect(self.animate\_button)
    - **Navigate Between Pages**: Methods to switch the main content area to different pages.  
      Python code  
      def go\_to\_map(self):

self.mainPages.setCurrentWidget(self.mapPage)# Similar methods for other pages

* + - **Update Button Styles**: Highlight the active button to indicate the current page.  
      Python code  
      def button\_clicked(self, button):

# Reset styles for all buttons and Apply active style to the clicked button

* + - **Handle Button Clicks**: Connect buttons to their respective methods for functionality.

## **Detailed Function Explanations**

### **save\_map\_with\_global\_variable(map\_object, file\_path)**

* **Purpose**: Saves a Folium map to an HTML file and modifies it to include a global map variable.
* **Actions**:
  + Saves the map to an in-memory buffer.
  + Modifies the HTML content to declare map as a global variable.
  + Writes the modified HTML to the specified file.

### **add\_polygon\_to\_map(map\_html, coordinates)**

* **Purpose**: Adds a new polygon to an existing map and saves the updated map.
* **Actions**:
  + Reads the existing map's HTML content.
  + Creates a new Folium map and adds the existing content.
  + Adds the new polygon based on provided coordinates.
  + Saves the updated map to map1.html.

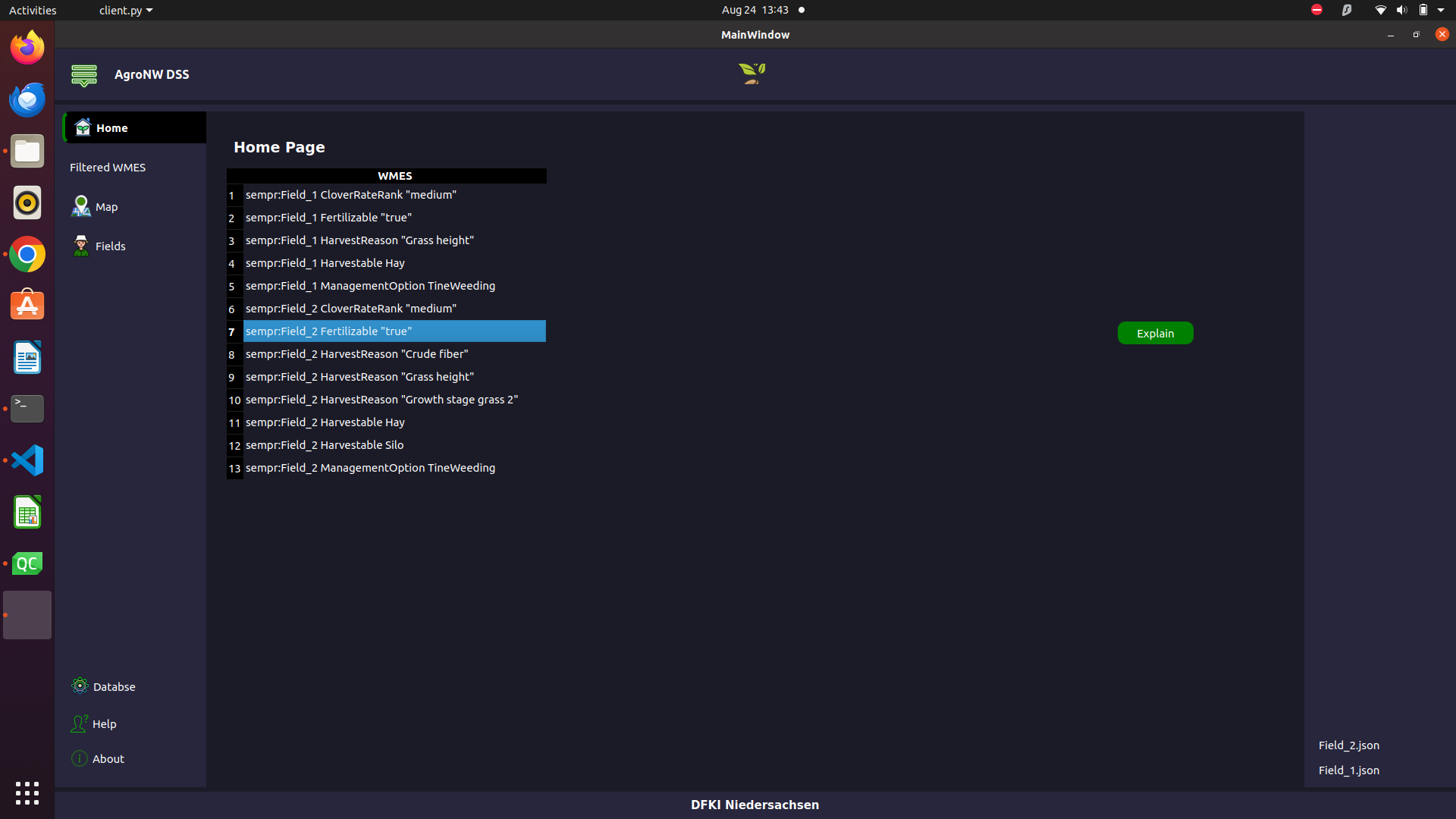
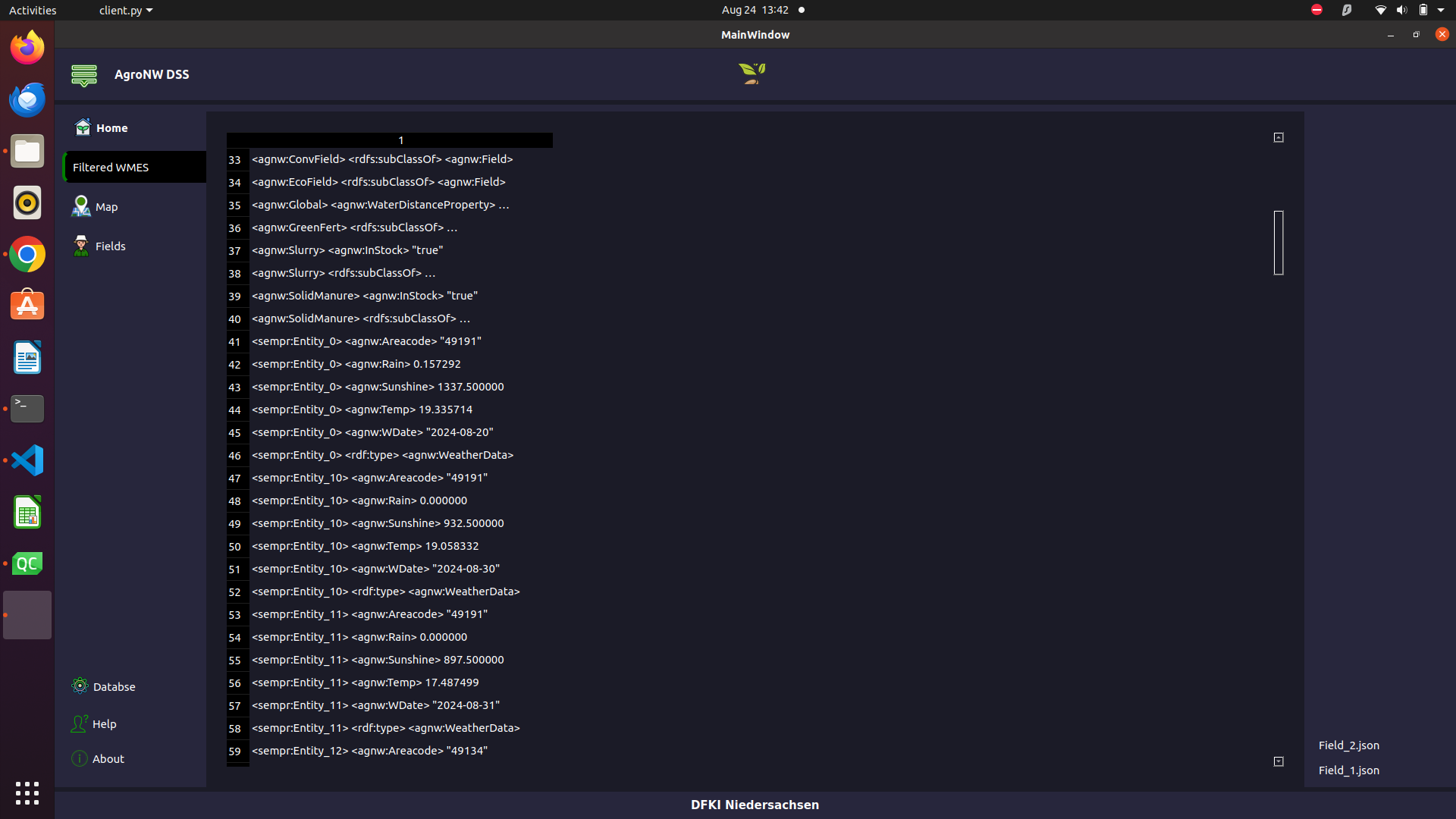
### **MainWindow Class**

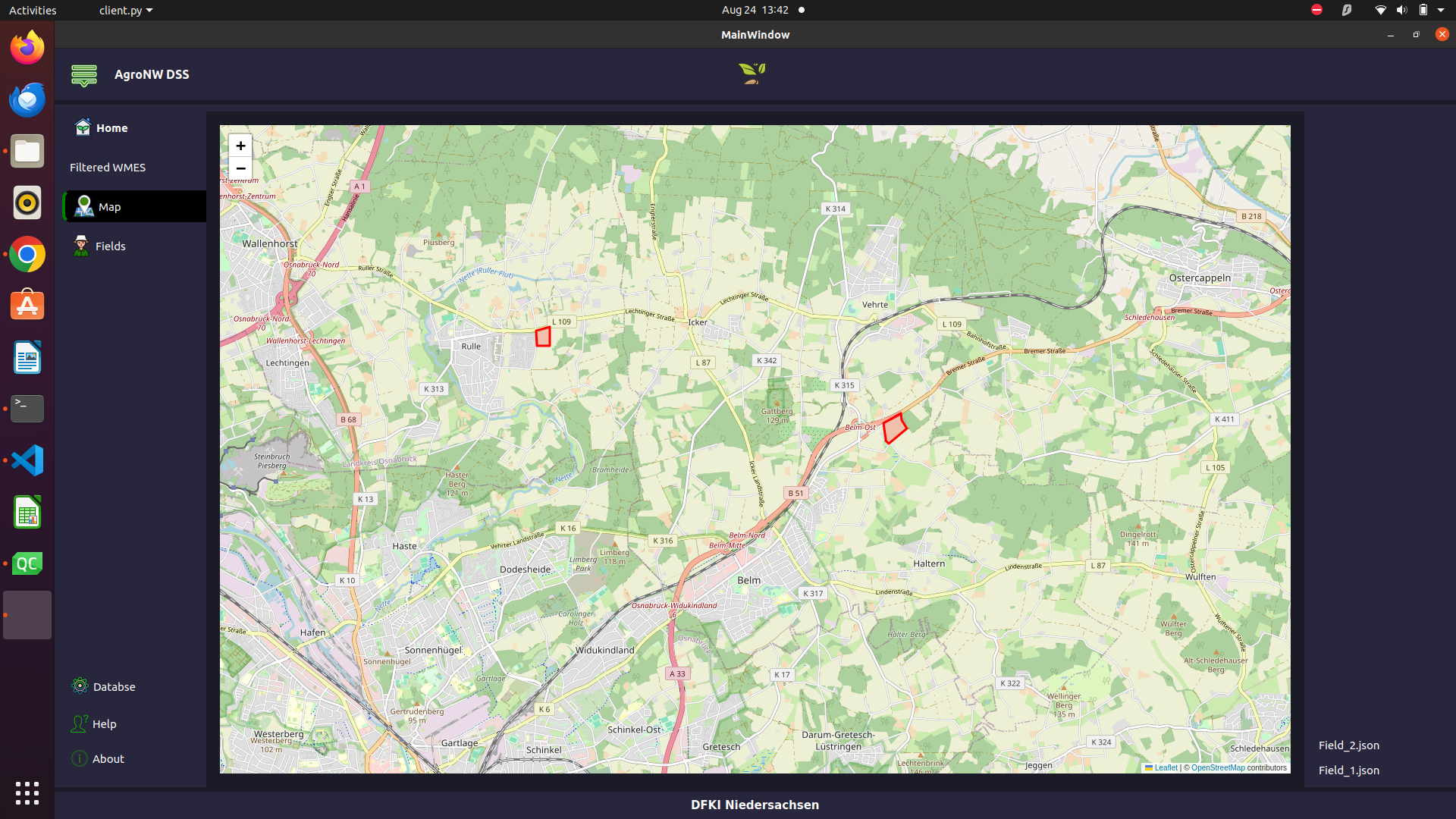
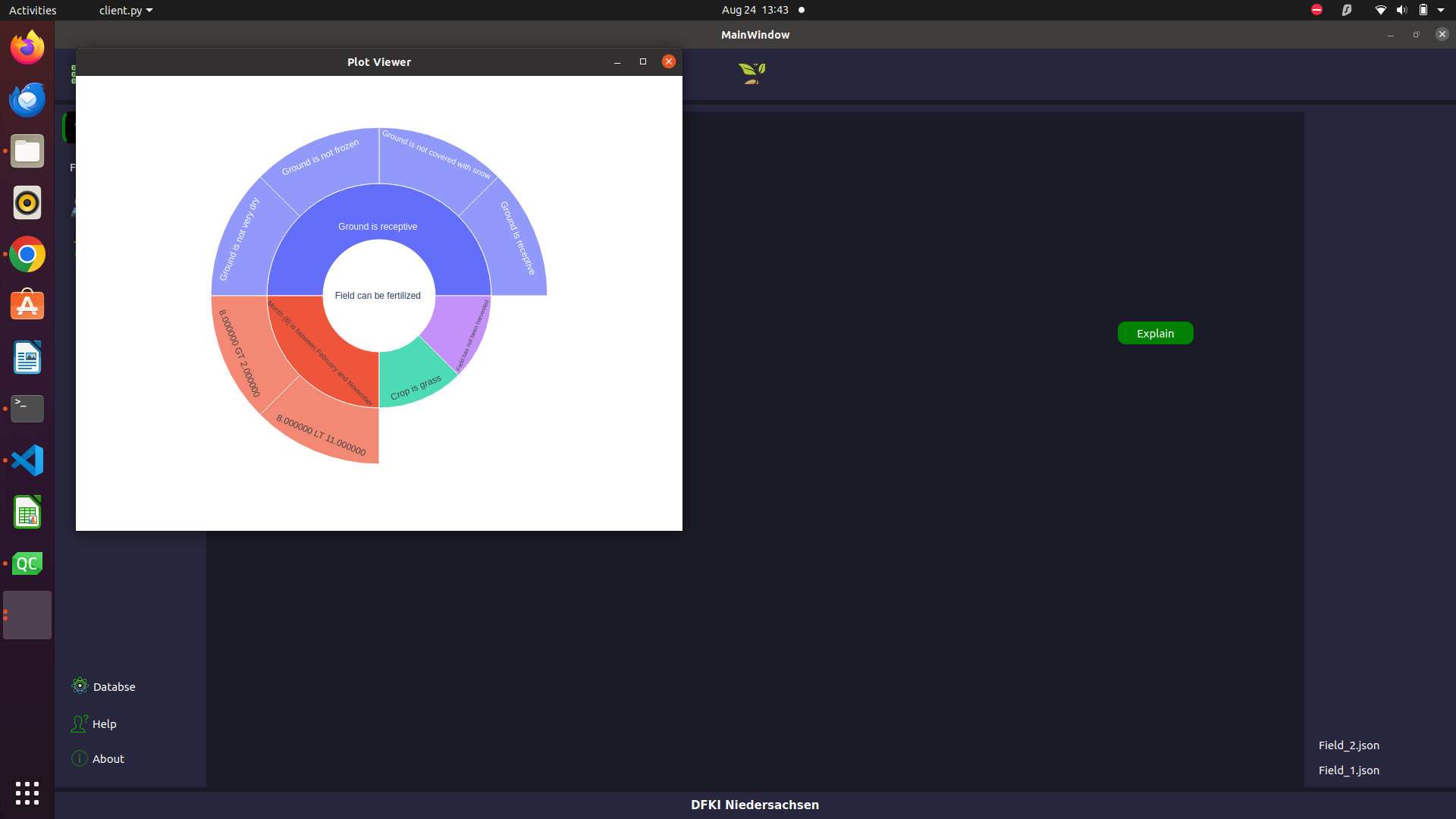
* **Initialization**:
  + Loads the UI from interface.ui.
  + Sets up initial styles and connects signals to slots.
* **add\_new\_field\_polygoan Method**:
  + Loads existing fields from JSON files.
  + Adds polygons to the map for each field.
  + Updates the map display in the application.
* **fetch\_wmes\_data and fetch\_fwmes\_data Methods**:
  + Fetch data from the backend server.
  + Populate tables with the retrieved data.
* **on\_cellClicked Method**:
  + Displays the "Explain" button when a table cell is clicked.
  + Positions the button next to the clicked cell.
  + Connects the button to the explain\_clicked method with the cell's content.
* **explain\_clicked Method**:
  + Sends a request to the backend to get an explanation for a selected WME.
  + Generates a sunburst plot using the explanation.
  + Displays the plot in a new window.
* **init\_map Method**:
  + Sets up the interface for adding a new field.
  + Initializes a Folium map with drawing capabilities.
  + Adds event handlers for adding attributes and updating the database.
* **add\_attribute\_value Method**:
  + Adds the selected attribute and entered value to a temporary list.
  + Displays the added attribute-value pair.
* **update\_database Method**:
  + Sends the new field data to the backend to update the database.
  + Refreshes the map and data after updating.
* **get\_polygon\_coordinates and extract\_coordinates Methods**:
  + Retrieves the coordinates of the drawn polygon from the map.
  + Adds the coordinates to the temporary data list.
* **Navigation Methods** (go\_to\_map, go\_to\_home, etc.):
  + Switches the main content area to the specified page.
  + Updates the visibility of field buttons accordingly.
* **button\_clicked Method**:
  + Resets the styles of all navigation buttons.
  + Highlights the clicked button.

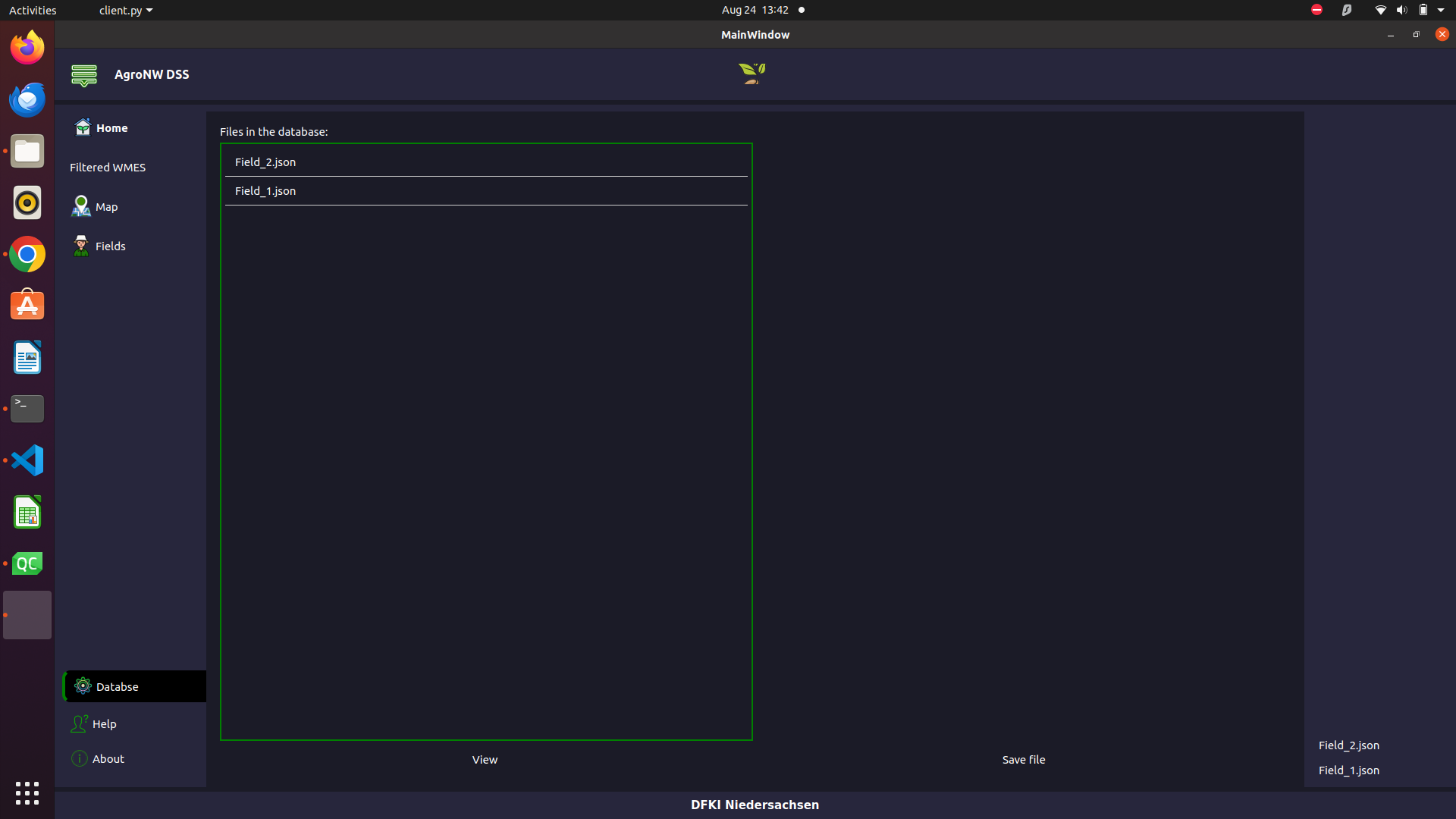
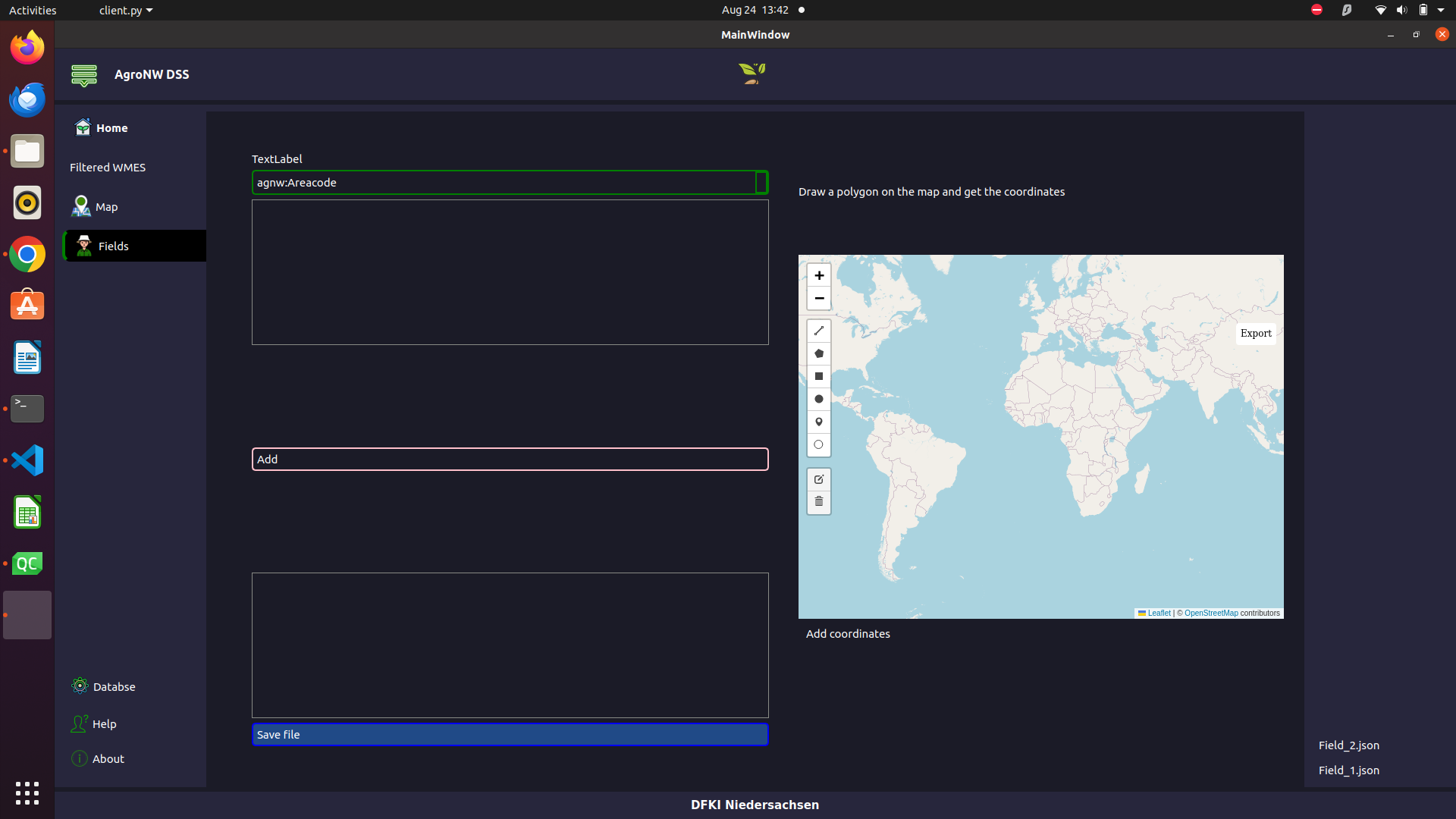
### **Other Utility Functions**

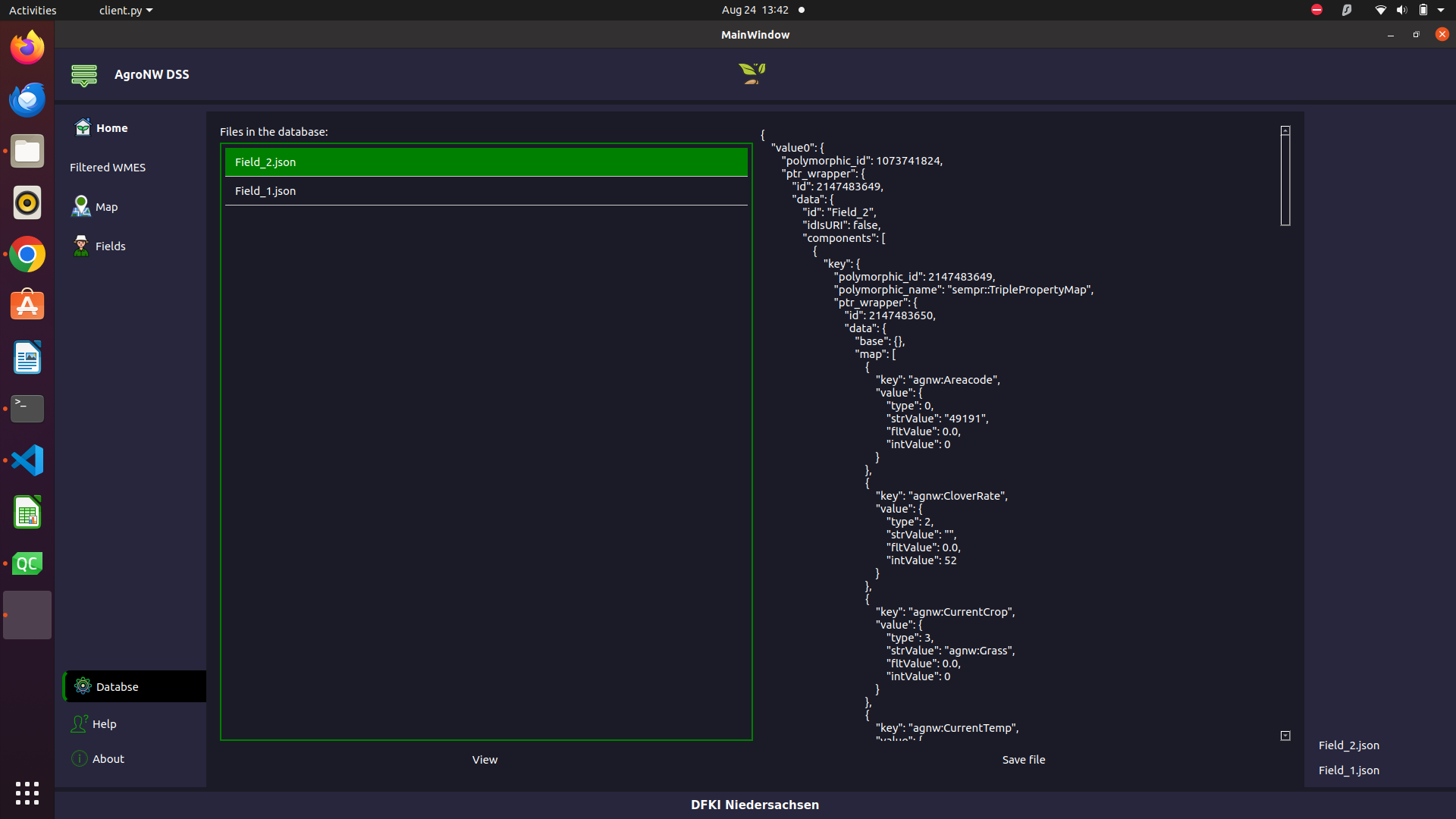
* **read\_existing\_map(file\_path) and load\_existing\_map(file\_path)**:
  + Reads the content of an existing map HTML file.
* **focus\_on\_polygon(polygon)**:
  + Centers the map view on the centroid of a given polygon.
* **show\_html\_plot(html\_file)**:
  + Opens a new window to display an HTML plot using QWebEngineView.
* **clear\_layout(layout)**:
  + Removes all widgets from a given layout.

**Some working pictures of the application:**

****

****

****

****