Development of a GIS Plugin for Interactive Building Energy Data Management and Visualization: Integration of RESTful APIs with QGIS for Urban Energy Planning.

Table of contents

1. Abstract

With the increasing urgency of climate action and energy transition in the built environment, municipalities require efficient tools to assess, monitor, and improve building energy performance. The developed Energy Plugin for QGIS is a comprehensive QGIS extension designed to streamline the management, visualization, and analysis of building energy efficiency data across German municipalities and it relies of the data available to the GIS World backend.

It implements client-server architecture using PyQt and the QGIS API to integrate building energy data from a remote backend through JWT-authenticated REST API calls. The plugin supports interactive attribute editing through metadata-driven forms that synchronize changes via HTTP PUT requests, automatically updating both the backend database and local map visualization in QGIS project.

It serves multiple stakeholder groups including urban planners, energy consultants, municipal authorities and researchers to respectively identify energy renovation priorities and assess district-level energy efficiency patterns, access up-to-date building energy data for analysis and reporting, monitor progress toward climate neutrality goals and plan targeted interventions, and analyze spatial patterns of energy consumption and renovation effects.

The Energy Plugin is freely available for installation on QGIS 3.22 and higher. The plugin requires no additional dependencies beyond the standard QGIS installation.

For future enhancements, it would be interesting to deploy the same functionalities on an augmented reality (AR) device and mobile app for 3D visualization and editing.

2. Introduction

2.1. Background and Context

The accelerating transition toward sustainable urban development and climate neutrality has elevated building energy efficiency to a critical research and policy priority for municipalities throughout Germany [1]. Buildings currently account for approximately 40% of total energy consumption and 36% of greenhouse gas emissions within the European Union, necessitating comprehensive, spatially explicit tools for the management and analysis of building energy data [2], [3].

Municipal authorities, urban planners, and energy consultants require robust methodologies to access, visualize, and update building energy information within geospatial frameworks. Conventional approaches relying on fragmented databases, manual data extraction protocols, and static visualization techniques have demonstrated insufficient capacity to address the dynamic requirements of energy transition planning. The technical challenge transcends basic data accessibility, demanding integrated systems that synthesize spatial analysis functionality with real-time data management and intuitive cartographic representation [4].

die STEG Stadtentwicklung GmbH identified a critical architectural gap in existing geospatial toolchains: while enterprise-grade backend systems maintain comprehensive building energy repositories and advanced Geographic Information System (GIS) platforms provide sophisticated spatial analysis capabilities, no seamless interoperability framework existed to bridge these distinct technological domains. Energy consultants consequently navigated inefficient workflows characterized by multi-platform software dependencies, manual data transfer operations, and extensive visualization configuration procedures.

This implementation report presents the methodology underlying the Energy Plugin for QGIS, a domain-specific solution designed to address these systemic inefficiencies by enabling direct integration of building energy data within standardized GIS environments.

2.2. Objectives

The Energy Plugin was developed with the following primary objectives: (1) create a direct connection between the GISWorld backend API

and QGIS, eliminating manual data transfer steps and ensuring users always work with current information; (2) Design a user interface that guides users from authentication through data discovery, download, visualization, and editing in a logical, efficient sequence; (3) Implement intelligent symbology that automatically represents energy efficiency classes through standardized color coding, enabling immediate visual assessment of building performance; (4) Enable users to update building energy attributes directly within QGIS, with changes immediately synchronized to the backend database and reflected in map visualizations.

3. Methodology

The development of the Energy Plugin for QGIS follows a client-server architecture that integrates building energy data from a remote backend API into a desktop GIS environment. The plugin is implemented in Python using the PyQt framework and QGIS API version 3.x. The methodology encompasses three primary components: user authentication and data acquisition, spatial data visualization, and interactive building attribute editing.

The authentication workflow begins when users access the plugin through a tabbed dialog interface. Users authenticate via JWT (JSON Web Token) bearer authentication by providing their credentials to the GISWorld backend API endpoint. Upon successful authentication, the system stores the access token locally using Qt's QSettings mechanism.

Data acquisition follows a multi-step filtering process designed to minimize network overhead and provide granular spatial control. Users first search for municipalities by name or official community key or id (AGS - Amtlicher Gemeindeschlüssel), with the interface providing real-time autocomplete suggestions through debounced API queries. Once a municipality is selected, the system optionally retrieves cadastral districts (Gemarkungen) to enable fine-grained spatial filtering. Users then select desired datasets (buildings and land parcels; or pipes) before initiating download requests to the backend API, which returns GeoJSON-formatted vector data with embedded building energy attributes.

The downloaded building data undergoes automatic coordinate reference system (CRS) assignment to EPSG:25832 (ETRS89 / UTM zone 32N), the standard projection for German geospatial data. The plugin then applies categorical symbology based on energy efficiency classifications embedded in the dataset. Each building receives a color code corresponding to its specific energy demand (kWh/m²a), ranging from class A (≤25 kWh/m²a or no data, rendered in gray) through class K (>250 kWh/m²a, rendered in dark red).

For interactive data editing, the plugin implements an advanced editing module that synchronizes with the remote backend through RESTful API operations. When users select a building feature in QGIS, the system retrieves detailed energy attributes via HTTP GET requests, presenting them in a structured table interface organized into logical sections (e.g., general information, pre-renovation state, post-renovation state). Form controls are dynamically generated based on field metadata: categorical attributes render as dropdown menus with human-readable labels, numeric fields with defined ranges become spinboxes or double-precision spinners, and free-text attributes appear as editable text cells.

Attribute modifications are committed through HTTP PUT requests that merge user edits with the complete feature attribute set to satisfy backend validation requirements. The plugin converts QGIS-specific data types (QVariant objects) to JSON-serializable Python primitives before transmission. Upon successful update, the system synchronizes the local QGIS layer by updating both the user-modified attributes and any server-calculated fields (such as recalculated energy efficiency colors), then triggers a layer repaint to immediately reflect changes in the map visualization.

The plugin architecture employs asynchronous network operations using Qt's QNetworkAccessManager to prevent UI blocking during API communication. Signal-slot mechanisms coordinate data flow between the network layer, business logic, and user interface components [5], [6]. The modular design separates concerns into distinct classes: ApiClient handles all HTTP communication, SmartDownloaderDialog manages the user interface, and AdvancedEditingMixin extends the dialog with editing capabilities through multiple inheritance patterns.

4. System Architecture

4.1. User Interface Design

4.1.1. Tab 1: Configuration

The purpose of this tab is to allow the user to authenticate before accessing the data and

managing sessions. The user enters the credentials (Email/Username input field - Password input field (masked) - “Sign In” button - or “Open Sign Up Page” if the user does not yet have an account - “Skip (already signed in)”), the plugin sends POST request to /api/token/, the JWT token is stored in QSettings for persistent sessions and finally the user is automatically redirected to Filter Data tab on success.

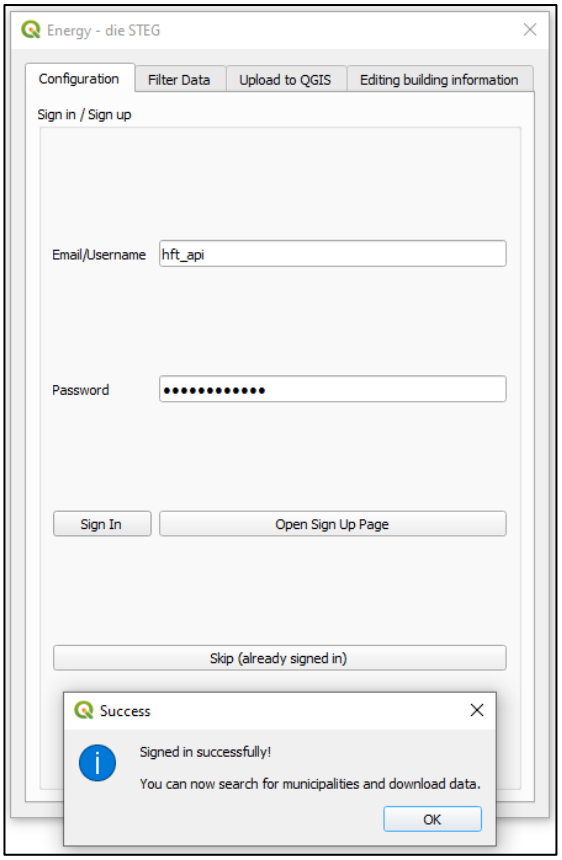

Figure 1.: Sign in and configuration tab of the developped plugin

4.1.2. Tab 2: Filter Data

The Filter Data tab enables users to precisely select a geographic area before downloading energy datasets. It provides autocomplete search fields for both municipality name and municipality ID, with live, debounced suggestions updating as the user types. Users can search by name or ID simultaneously, and once a municipality is selected, an optional Gemarkung dropdown becomes available for hierarchical filtering. After defining the desired area, users proceed using the Download Data & Continue button

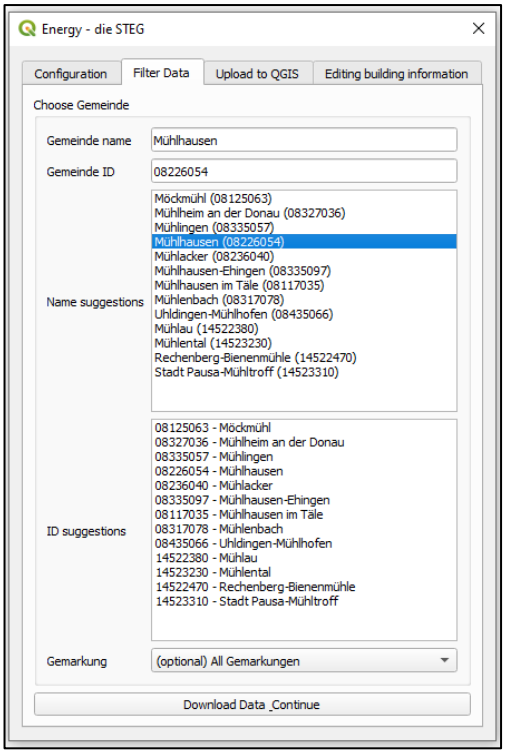

Figure 2.: Selecting the Gemeinde and optionally the Germakung

4.1.3. Tab 3: Upload to QGIS

The Upload to QGIS tab allows users to select which datasets to load into the QGIS project. Users can choose buildings and/or land parcels via checkboxes and then load the selected layers using the Upload to QGIS button. The plugin automatically assigns the EPSG:25832 coordinate reference system, applies predefined energy-related symbology to building layers, handles GeoJSON inputs, and manages all files as temporary layers within QGIS.

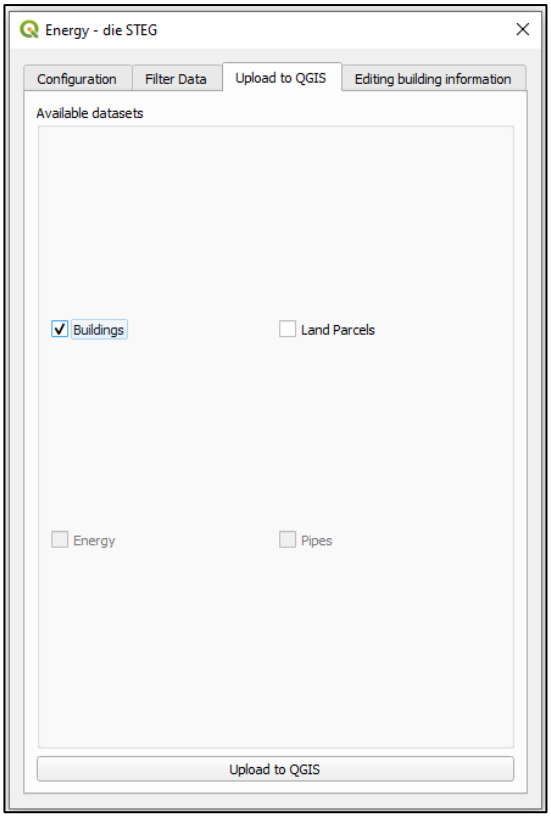

Figure 3.: Choosing the data to upload to QGIS

4.1.4. Tab 4: Editing building information

The Editing Building Information tab provides an interactive interface for modifying building attributes. It displays a structured two-column table of attributes and values, organized with clearly highlighted section headers. Attribute values are edited using context-appropriate widgets such as combo boxes for categorical fields, spin boxes for numeric inputs, and text fields for free text. When a building is selected, its attributes are automatically loaded and the interface switches to this tab. All edits are validated against predefined options and synchronized in real time with both the backend and the QGIS map, with options to save or cancel changes.

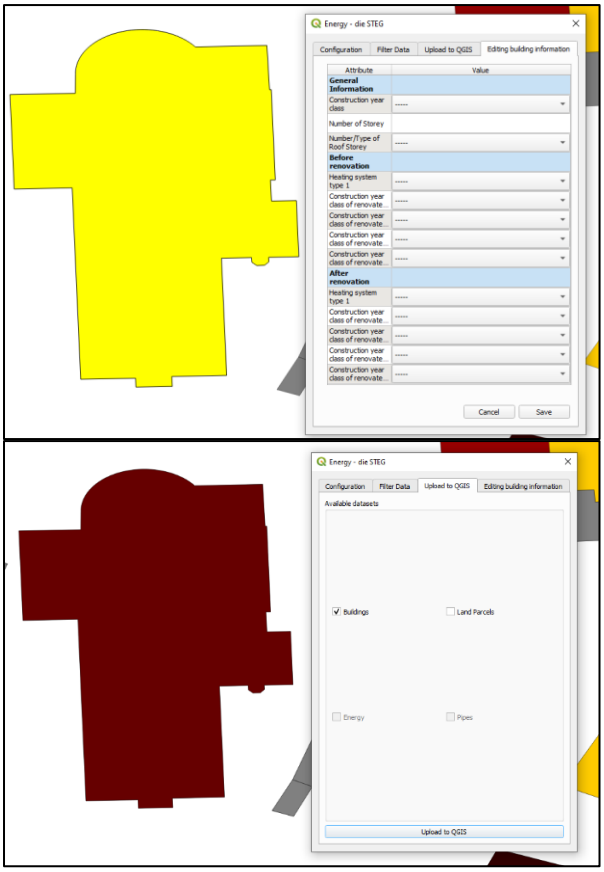

Figure 4.: Building attributes edition and automatic colorization in QGIS

5. Core functionalities

5.1. Authentication system

The authentication system uses a token-based a pproach to securely access the API. Users authe nticate via a POST request to the /api/token/ end point by providing their email and password, and the server responds with a JWT access token. Th is token is securely stored in the plugin settings a nd automatically attached to all subsequent API r equests through the HTTP Authorization header using the Bearer scheme, enabling authenticated communication without repeated logins.

5.2. Community Search

The Community Search functionality retrieves municipalities through the GET /geospatial/communities/?search={query} endpoint. It supports case-insensitive and partial text matching, allowing users to find communities even with incomplete input. Search queries are URL-encoded to handle special characters correctly, and the API returns matching results, such as municipality key, name, and AGS, sorted by relevance to provide accurate and responsive autocomplete suggestions.

5.3. Building Data Download

The building data with symbology is downloaded using the GET API /geospatial/communities/{community_key}/buildings/?include_energy_colors=true.
The 'include_energy_colors=true' parameter ensures color fields are included. This API returns GeoJSON FeatureCollection. For this data, each feature contains a geometry (polygon), properties (all building attributes) and energy color fields.

5.4. Building Attribute Retrieval and update

To retrieve the building attribute before editing, the GET API /geospatial/buildings-energy/{building_key}/?community_id={id}&field_type=basic (or field_type=advanced for more details) is used. It fetches detailed, editable attributes for a selected building. For saving the edits, the PUT API /geospatial/buildings-energy/{building_key}/?community_id={id}&field_type=basic was used.

6. Energy efficiency visualization

The energy classes and the color codes as defined under the Gebäudeenergiegesetz (GEG), the German consumer advice centre. These colors are automatically calculated according to the energy demand and they are streamed and used for symbology in QGIS (schema 9).

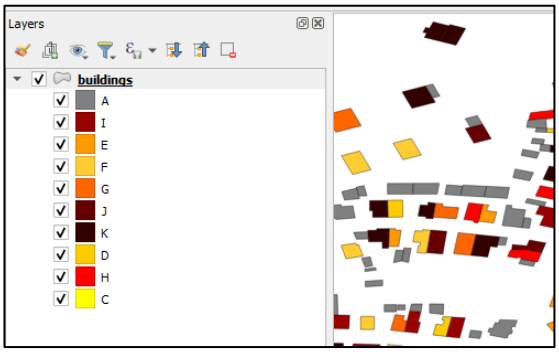

Figure 5.: Upload data and automatic symbology in QGIS project

7. Technical challenges and solutions

The user had to reload the plugin after selecting a building for it to be detected and its attributes to be retrieved for editing. This was time consuming and not appropriate for a semi-automatic plugin. To solve this, the building selection auto-detector (schema 8.) was implemented and it allowed the user to retrieve automatically the attributes loaded in the plugin form without any manual refresh needed.

A key technical challenge was handling JSON serialization of QVariant objects. QGIS layer attributes are returned as QVariant, which cannot be directly serialized, resulting in a TypeError. To resolve this, a conversion utility was implemented to transform QVariant and related QGIS date/time types into Python-native representations, while safely handling null values and nested structures. This ensured all attribute data could be serialized into valid JSON, enabling reliable PUT requests and seamless communication with the backend API.

Another challenge involved handling array-based choice values returned by the API. Certain attributes, such as `roof_storey`, were provided as arrays containing both the internal code and its human-readable label. When these arrays were sent back unchanged in update requests, the API rejected them because only the coded value was considered valid. This issue was resolved by extracting and sending only the first element of the array (the valid choice key), ensuring successful validation during data updates.

The Color Field Synchronization challenge arose when edited building attributes did not immediately update the map visualization. This occurred because color-related fields were not written back to the local QGIS layer and the layer symbology was not refreshed after updates. The solution involved extracting updated color values from the API response, applying them directly to the corresponding layer attributes, re-applying the predefined energy symbology, and triggering a layer repaint. This approach ensured real-time visual feedback on the map whenever a building's energy classification changed.

The Missing Color Fields in Downloaded Data challenge occurred because initial building downloads did not include energy color attributes, preventing symbology from being applied. The issue was resolved by adding a query parameter (include_energy_colors=true) to the building download URL, ensuring these fields are included in the GeoJSON. Verification involved checking the downloaded features for color-related fields, confirming their presence. This fix allows energy-based symbology to be applied immediately, enabling accurate map visualization right after data download.

8. Conclusion

The Energy Plugin provides a complete solution for managing building energy data within QGIS. It features seamless integration with the GISWorld backend API, an intuitive four-tab interface, real-time attribute editing with instant map feedback, automatic energy class visualization, and robust error handling with production-ready code.

For die STEG, this plugin reduces 80% of manual data entry. For the users, it is a user-friendly, intuitive interface which provides fast handling and editing of energy data.

9. Future enhancements

For future enhancements, it would be interesting to deploy the same functionalities on an augmented reality (AR) device for 3D visualization and editing. Also, for many users, it would be interesting to deploy it as a mobile app where the users would be filling in the username and password, then the community id and the asset id from cesium ion. These two improvements would allow users to connect 2D to 3D data and to edit 3D attributes on their own smartphones and visualize automatically the impact in CO2 symbolization.

10. References

[1] European Commission, “Energy performance of buildings,” 2023. [Online]. Available: https://energy.ec.europa.eu/topics/energy-efficiency/energy-performance-buildings_en

[2] European Environment Agency, “Final energy consumption by sector,” 2022. [Online]. Available: https://www.eea.europa.eu

[3] IPCC, “AR6 Climate Change Mitigation,” 2022. [Online]. Available: https://www.ipcc.ch/report/ar6/wg3/

[4] R. Tomlinson, *Thinking About GIS*. Esri Press, 2013.

[5] QGIS Development Team, “PyQGIS Developer Cookbook,” 2024. [Online]. Available: https://docs.qgis.org

[6] Qt Company, “Qt Network Module Documentation,” 2023. [Online]. Available: https://doc.qt.io/qt-5/qtnetwork-index.html

11. Appendices

Here are some code blocks of the most relevant functionalities.

Schema 1.: JSON like structure for the Building energy information

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[...]]]
      },
      "properties": {
        "gml_id": "DEBWL001000hyHDD",
        "community_id": "08417008",
        "construction_year_class": "D",
        "begin_energy_demand_specific": 150,
        "begin_energy_demand_specific_color": "#ff9900",
        "end_energy_demand_specific_color": "#ffcc00",
        ...
      }
    }
  ]
}
```

Schema 2.: Code block adding query parameter to building download URL for solving the missing color code in the API response

```
url = f"{self.base_url}/geospatial/communities/{community_key}/buildings/"
url += "?include_energy_colors=true"
```

Verification:

```
# Log fields in downloaded GeoJSON
if "features" in geojson_data:
    sample_props = geojson_data["features"][0].get("properties", {})
    field_names = list(sample_props.keys())
    color_fields = [f for f in field_names if 'color' in f.lower()]
    print(f"[SmartDL] Found color fields: {color_fields}")
```

Schema 3.: Update the edited building color strategy

```
def _adv_on_update_finished(self, data, error):
    # 1. Extract color fields from API response
    color_updates = extract_color_fields(data)

    # 2. Update local layer
```

```
    for field_name, color_value in color_updates.ite
ms():
        field_idx = lyr.fields().indexOf(field_name)
        lyr.changeAttributeValue(self._adv_fid, field_i
dx, color_value)

    # 3. Re-apply symbology
    self._apply_energy_symbology(lyr)

    # 4. Trigger repaint
    lyr.triggerRepaint()
```

Schema 4.: API response returning choice values as arrays

```
{
  "roof_storey": [0, "Flat Roof"]
}
```

Sending this array back caused validation errors:

```
"[0, 'Flat Roof'] is not a valid choice"
```

Schema 5.: Extract only the code (first element) or label (second element) from the returned array.

```
# In _extract_choice_list - for display
if isinstance(entry, (list, tuple)) and len(entry) >= 2
:
    code = entry[0]
    label = str(entry[1])
    combo.addItem(label, code)  # Show label, store
code
# In _adv_collect_current_values - for submission
idx = editor.currentIndex()
val_to_send = editor.itemData(idx)  # Get stored c
ode, not display label
```

Schema 6.: JSON Serializer of QVariant

```
def _to_python_native(value):
    from qgis.PyQt.QtCore import QVariant, QDate,
QDateTime, QTime

    if isinstance(value, QVariant):
        if value.isNull():
            return None
        value = value.value()
    if isinstance(value, QDateTime):
        return value.toString(QtCore.Qt.ISODate)
    elif isinstance(value, QDate):
        return value.toString(QtCore.Qt.ISODate)
    elif isinstance(value, QTime):
        return value.toString(QtCore.Qt.ISODate)

    if isinstance(value, (str, int, float, bool)):
        return value
    elif isinstance(value, (list, tuple)):
        return [_to_python_native(item) for item in v
alue]
    elif isinstance(value, dict):
        return {k: _to_python_native(v) for k, v in val
ue.items()}
    else:
        return str(value)
```

Schema 7.: Building selection auto-detector

```
def _adv_on_active_layer_changed(self, layer):
    # Disconnect old layer
    if self._adv_layer:
        self._adv_layer.selectionChanged.disconnect(
)
    # Connect new layer
    self._adv_layer = layer
    if self._adv_layer:
        self._adv_layer.selectionChanged.connect(sel
f._adv_on_selection_changed)

def _adv_on_selection_changed(self):
    # Load building attributes
    self._adv_load_selected()
    # Auto-switch to editing tab
    self.tabWidget.setCurrentIndex(3)
```

Schema 8.: JSON schema of submitted attributes updates to the database

```
{
  "construction_year_class": "D",
  "roof_storey": "0",
  "begin_heating_system_type_1": "3",
  "begin_renovated_window_class_year": "D"
}
```

Schema 9.: JSON from the API request, supposed to populate the attributes editing form

```
{
  "general_information": {
    "label": "General Information",
    "fields": {
      "construction_year_class": {
        "label": "Construction Year",
        "value": "D",
        "display": "D- 1949 - 1957",
        "choices": [
          ["A", "A- before 1859"],
          ["B", "B- 1860 - 1918"],
          ["C", "C- 1919 - 1948"],
          ["D", "D- 1949 - 1957"]
        ]
      }
    }
  },
  "before_renovation": {
    "label": "Before Renovation",
    "fields": {...}
```

```
  },
  "after_renovation": {
    "label": "After Renovation",
    "fields": {...}
  }
}
```