DERLA Documentation

DERLA is an interdisciplinary documentation and mediation project of the Center for Jewish Studies (CJS), the Center for Information Modeling (ZIM), the Department of History Didactics of Karl-Franzens-University Graz and _erinnern.at_ - National Socialism and the Holocaust: Memory and the Present. In addition to the documentation of all memorial sites and signs for the victims as well as the sites of terror of National Socialism in Austria, it sets itself the goal of developing digital mediation offers.

At the time of writing this documentation, data from 4 federal states of Austria are already part of the project data. These are subject to regular updates. The process of data updates makes up the first part of this document. At some point more or even all federal states might be added to the project, adding to the project's data pools. This process is handled in the second part of this document.

Data Updates

The following part of documentation refers to updates of the existing data pools (federal states of Austria) in DERLA. No extensions are made in the process.
In some cases, the data basis for the individual subsections of the project must be generated on a state-specific basis and then ingested.
This applies to:

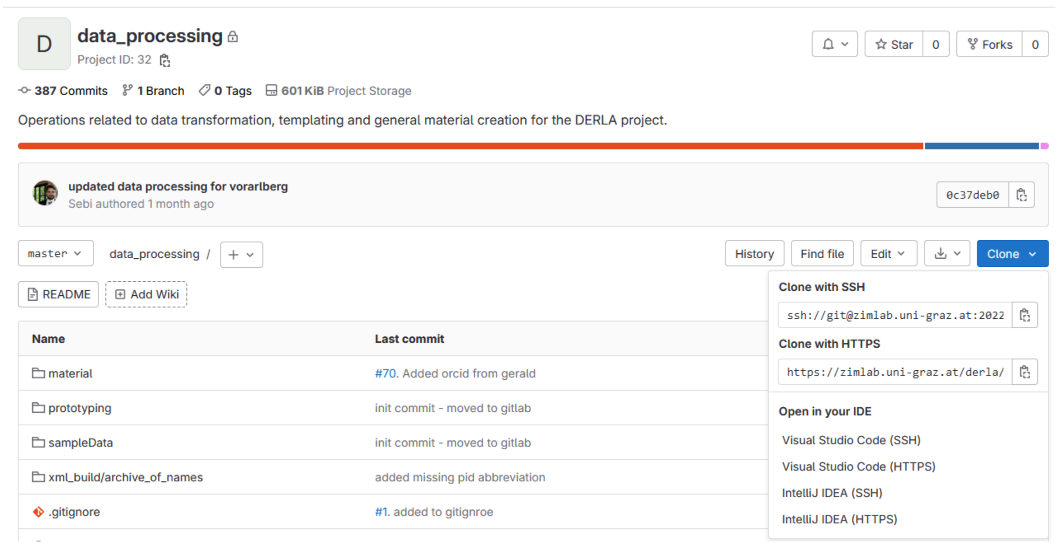
- Orte der Erinnerung/Places of Remembrance
- Archiv der Namen/Archive of the Names
- Fixierte Vermittlungsangebote/Fixed Mediation Offers

The following subsections of data are not state-specific:

- Didaktisches Glossar/Didactical Glossary
- Dynamische Vermittlungsangebote/Dynamic Mediation Offers
- Historischer Thesaurus/Historical Thesaurus
- Karte der Erinnerung/Map of Remembrance
- Schlagwortregister/Keywords
- Wege der Erinnerung/Paths of Remembrance

Data Creation

The necessary scripts for data creation can be found at https://zimlab.uni-graz.at/derla in the "data processing" subfolder. You will need to clone the repository to your local computer:


D data_processing
Project ID: 32

387 Commits 1 Branch 0 Tags 601 KiB Project Storage

Operations related to data transformation, templating and general material creation for the DERLA project.

updated data processing for vorarlberg
Sebi authored 1 month ago
0c37deb0

master data_processing / +

History Find file Edit Clone

README Add Wiki

Name	Last commit
material	#70. Added orcid from gerald
prototyping	init commit - moved to gitlab
sampleData	init commit - moved to gitlab
xml_build/archive_of_names	added missing pid abbreviation
.gitignore	#1. added to gitignroe

Clone with SSH
ssh://git@zimlab.uni-graz.at:2022
Clone with HTTPS
https://zimlab.uni-graz.at/derla/
Open in your IDE
Visual Studio Code (SSH)
Visual Studio Code (HTTPS)
IntelliJ IDEA (SSH)
IntelliJ IDEA (HTTPS)

To clone the repository via HTTPS, copy the corresponding link. Then open your terminal and navigate to the directory, where you would like to clone the repository (you might want to create a parent folder for DERLA first).
Type the command “git clone” and the copied link from before, then press Enter:

```
git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
```

It is necessary to set up a virtual environment, which contains a particular installed version of Python and probably a number of additional packages in particular versions. This ensures that no complications arise from interfering versions on your local computer and the required ones to update the DERLA data pool. The Python model, which is needed to manage and create virtual environments, is called venv. Also refer to the chapter on “Virtual Environments and Packages” of the Python documentation: https://docs.python.org/3/tutorial/venv.html.

Create your virtual environment in the root of your cloned “data_processing” repository:

```
python -m venv tutorial-env
```

This creates the “tutorial-env” directory and all needed supporting files as well as the Python interpreter. You might want to call the directory of your venv “.venv” so that it might be hidden in your folder structure and won’t interfere with “.env”.

After creation, you will need to activate it by running

```
tutorial-env\Scripts\activate
```

on Windows, or

```
source tutorial-env/bin/activate
```

on Unix or MacOS. To deactivate the virtual environment, simply type

```
deactivate
```

To set up all necessary packages, you use the provided list “requirements.txt”, installing them via pip:

```
python -m pip install -r requirements.txt
```

Tip: Check your env variables in env.py to make sure the links are pointing to GAMS as your production environment.

The script “erla_xml_builder.py” handles the creation of the various TEI-XML files and the folder structure. To make sure, your system is able to access the necessary data in the Google Sheets, you will need to adapt the variable “data_path” in the script to the location where it’s able to access the Google Drive data.

```
data_path = f"G:\\Meine
Ablage\\01_Berufliches\\01_Projekte\\02_digitale_erinnerungslandschaft\\
03_data\\"
```

For further documentation and code description of erla_xml_builder.py, please refer to the chapter *Code and Script Description*.

Places of Remembrance

1. In your virtual environment via console access data_processing:

```
python main.py
```

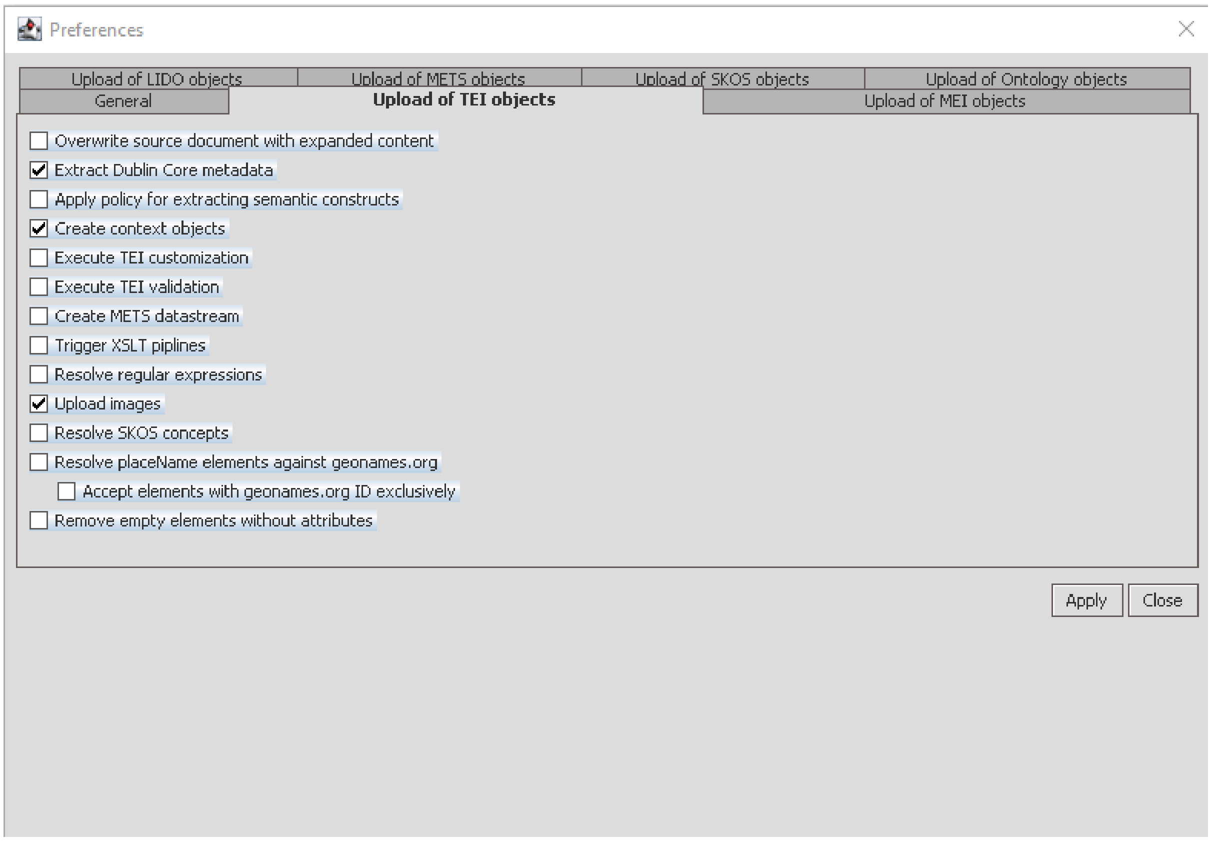
Tip: During the creation of the TEI-XML files the folder "results" is created. This folder should be emptied before each data generation to avoid complications. Also do not ingest directly from the “results” folder. Instead create a separate “ingest” folder to store all TEI-XMLs and needed images for ingest.

The script creates TEIs + one GML for Places of Remembrance, but is also able to create the list (TEI) of Archive of the names, if needed. Adapt the code according to your demands.

```
import gSheetHandler as g_handler
import erla_xml_builder as erla_xml_builder
from env import *
import logging
import os

def main():
    # at file import create log folder if not exist
    try:
      folder_path_name = "." + os.path.sep + "log"
      os.mkdir("./log")
    except FileExistsError:
      pass
    # set logging folder and level

logging.basicConfig(filename='log/erla_xml_builder.log',level=logging.DE
BUG, filemode="w")
    logging.debug("Applying log file -- main.py called. Starting
program")

    # at last start actual process
    # (data processing for DERLA-project)
    create_ingest_material()


def create_ingest_material():
  #####
  # Data generation for styria
  # g_data: [[str]] =
g_handler.get_data(sheet_id="1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z83AyjjR
M")
  # # g_handler.back_up_data(g_data, "./backups/steiermark/styr")

  # # Getting the person data for Styria
  # persons_sty: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d
/e/2PACX-1vTRFAMnUQ7YC9HZwwSrw55tXhopXcXrGaVl2CMGd8JMRMYiZn7lmDG-pCU2SmD
SuIR0-6vzgbTD29lS/pub?gid=1356804042&single=true&output=csv")
  # # print(persons_sty)

  # # build ingest files
  # styr_pid_base: str = f"o:{PROJECT_ABBR}.sty" # only 3 letters
allowed
  # erla_xml_builder.build_ingest_files(g_data,
pid_name=styr_pid_base,person_table=persons_sty, build_max_rows = 541)
  # erla_xml_builder.build_ingest_gml_files(g_data,
pid_name=styr_pid_base, build_max_rows = 1000)

  # #####
  # # Data generation for vorarlberg
  g_data: [[str]] =
g_handler.get_data(sheet_id="1hFJprVGgTUOBlHoW8RdzmYTGDzek_DebpOutQRdeZ_
k")
  # # g_handler.back_up_data(g_data, "./backups/vorarlberg/vor")

  # # Getting the person data for Vorarlberg
  persons_vor: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d
/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zqVqpFHI6iyL_pnzfmO5WKDTefozICa8qng
to4CB2PpdWYOxPBe0/pub?gid=2079639611&single=true&output=csv")
  # # print(persons_vor)

  # # build ingest files
  vor_pid_base: str = f"o:{PROJECT_ABBR}.vor"
  erla_xml_builder.build_ingest_files(g_data, pid_name=vor_pid_base,
person_table=persons_vor, build_max_rows = 250)

  # # print("Vorarlberg: Temporarily skipping build of gml files -> need
to adapt table model processing!")
  erla_xml_builder.build_ingest_gml_files(g_data, pid_name=vor_pid_base,
build_max_rows = 1000)


  ####
  # Data generation Tyrol
  # g_data: [[str]] =
g_handler.get_data(sheet_id="1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZn
c")
  # g_handler.back_up_data(g_data, "./backups/tirol/tir")

  # Getting the person data for Tyrol
  # persons_tir: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d
/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc/export?format=csv&gid=1986
675455")

  # build ingest files
  # tir_pid_base: str = f"o:{PROJECT_ABBR}.tir"
  # erla_xml_builder.build_ingest_files(g_data, pid_name=tir_pid_base,
person_table=persons_tir, build_max_rows = 234)
  # erla_xml_builder.build_ingest_gml_files(g_data,
pid_name=tir_pid_base, build_max_rows = 1000)


  ####
  # Data generation Carinthia
  # g_data: [[str]] =
g_handler.get_data(sheet_id="13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL
0")
  # g_handler.back_up_data(g_data, "./backups/tirol/tir")

  # Getting the person data for Carinthia
  # persons_car: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d
/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0/export?format=csv&gid=1026
060477")

  # build ingest files
  # car_pid_base: str = f"o:{PROJECT_ABBR}.car"
  # erla_xml_builder.build_ingest_files(g_data, pid_name=car_pid_base,
person_table=persons_car, build_max_rows = 231)
  # erla_xml_builder.build_ingest_gml_files(g_data,
pid_name=car_pid_base, build_max_rows = 1000)



if __name__ == '__main__':
    main()
```

For further documentation and code description of main.py, please refer to the chapter *Code and Script Description*.

2. Creation of the Submission Information Packages: TEI's + images in one ingest folder
   Be aware that the structure of image folders needs to be followed, otherwise the automatic generation will not be handled correctly. Sometime project partners are not aware.
3. If required, change Cirilo Extras/Preferences:



- [ ] Overwrite source document with expanded content
- [x] Extract Dublin Core metadata
- [ ] Apply policy for extracting semantic constructs
- [x] Create context objects
- [ ] Execute TEI customization
- [ ] Execute TEI validation
- [ ] Create METS datastream
- [ ] Trigger XSLT piplines
- [ ] Resolve regular expressions
- [x] Upload images
- [ ] Resolve SKOS concepts
- [ ] Resolve placeName elements against geonames.org
  - [ ] Accept elements with geonames.org ID exclusively
- [ ] Remove empty elements without attributes

4. Ingest in Cirilo (Glossa)
5. Check the result
6. Ingest in Cirilo (Gams)

Archive of the Names, Fixed Mediation Offers, Keywords and Didactical Glossary

The script xml_build/archive_of_names/__main__.py handles:

- the creation of the list (XML file) of person names for the Archive of the Names
- the individual XML file for each Fixed Mediation Offer
- a list of all Fixed Mediation Offers per state
- the Keyword XML file
- the Didactical Glossary XML file

You can either create all state specific files (Fixed Mediation Offers) or choose individual code blocks, which target single states. This is marked by comments in the script.
You will need to change the value of the env variable “origin” to either “glossa” or “gams” depending on the intended ingest.
There might be prototype data generated, so always check the ingest material beforehand and delete if necessary.
Update references in Cirilo (compare to existing objects):

- Replace Stylesheets for dissemination
- Context Relationships
- Dublin Core

```
# Author: Sebastian
#
from DERLAPersListHandler import DERLAPersListHandler
from DERLAKeyWordsHandler import DERLAKeyWordsHandler
from DERLATeiLoHandler import DERLATeiLoHandler
from DERLALoListHandler import DERLALoListHandler
from DERLADidacticGlossary import DERLADidacticsGlossary


def build_fixed_lo_xmls(env):
      """
      Method handles the building of educational related xml files.
      Individual TEI XML objects per educational material.
      """

      # sty fixed
      # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFK
Pl_z83AyjjRM/pub?gid=1790748251&single=true&output=csv"
      # env["pid_var"] = "sty"
      # derla_vor_lo_builder = DERLATeiLoHandler(env)

      # vor fixed (created by hand - skipped)

      # tir fixed
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEj
OGgDLDFjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc&gid=263259825"
      env["pid_var"] = "tir"
      derla_vor_lo_builder = DERLATeiLoHandler(env)

      # car fixed
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAa
Z6ns51xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0&gid=740894826"
      env["pid_var"] = "car"
      derla_vor_lo_builder = DERLATeiLoHandler(env)


def build_lo_list(env):
      """
      Method handles construction of los as lists
      (for the fixierte vermittlungsangebote).
      """

      # for styria
      # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFK
Pl_z83AyjjRM/pub?gid=1790748251&single=true&output=csv"
      # env["pid_var"] = "sty"
      # DERLALoListHandler(env)

      # vor list
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEj
OGgDLDFjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc&gid=263259825"
      env["pid_var"] = "tir"
      DERLALoListHandler(env)

      # tir list
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAa
Z6ns51xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0&gid=740894826"
      env["pid_var"] = "car"
      DERLALoListHandler(env)

def build_perslists(env):
      """
      Method to handle construction of perslist xmls needed in DERLA.
      """

      # vor persons
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8tw
vZS5zqVqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639
611&single=true&output=csv"
      env["pid_var"] = "vor"
      DERLAPersListHandler(env)

      # sty persons
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vTRFAMnUQ7YC9HZwwSrw55t
XhopXcXrGaVl2CMGd8JMRMYiZn7lmDG-pCU2SmDSuIR0-6vzgbTD29lS/pub?gid=1356804
042&single=true&output=csv"
      env["pid_var"] = "sty"
      DERLAPersListHandler(env)

      # tir persons
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEj
OGgDLDFjGZnc/export?format=csv&gid=1986675455"
      env["pid_var"] = "tir"
      DERLAPersListHandler(env)

      # car persons
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAa
Z6ns51xLzFL0/export?format=csv&gid=1026060477"
      env["pid_var"] = "car"
      DERLAPersListHandler(env)


def build_keywords_tei(env):
      """
      Handles building of keywords TEI.
      """
      # needed env setup
      env["get_param"] =
"https://docs.google.com/spreadsheets/d/1j3GeeoIwC72tlSToz8yrEgKo845gQ_y
VMDymcVhvNus/pub?gid=0&single=true&output=csv"
      env["pid_var"] = ""
      DERLAKeyWordsHandler(env)

def build_didactical_glossary(env):
      """
      Handles building of the didactical glossary for DERLA
      """

      DERLADidacticsGlossary(env)


if __name__ == "__main__":

      # Define env variables here as dictionary
      # with basic setting
      env = {
      "log_level": "DEBUG",
      "origin": "https://glossa.uni-graz.at",
      "pid_static": "o:derla.",
      "pid_var": "vor",
      "get_param":
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8tw
vZS5zqVqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639
611&single=true&output=csv"
      }

      # # instantiate your custom class here
      # derla_builder = DERLAPersListHandler(env)
      # build_perslists(env)

      # keywords tei
      # build_keywords_tei(env)

      # perslists
      # build_perslists(env)

      # build educational xml files
      # build_fixed_lo_xmls(env)

      #####
      # LO lists
      # build_lo_list(env)

      ## didactical glossary
      build_didactical_glossary(env)
```

Dynamic Mediation Offers

Dynamic Mediation Offers are created entirely manually. To complete, simply start with an existing one and swap all content accordingly. When ingesting, be sure to check the Dublin Core entries/complete them manually.

Paths of Remembrance

Paths of Remembrance are done entirely manually. Open an existing file (all are saved in Y:/data/projekte/derla/data) and change the contents accordingly. Be sure to save the new files in the folder as well. Create a new story object for each Path object separately. Ignore the Error message that pops up in Cirilo. Change the PID in the XML to whatever Cirilo assigned to the object. Change the text of the <text> element in the first <slides> element to the description text contained in Google Sheets. The <headline> is the title also contained in Google Sheets. As the values <lon> and <lat> elements in the first <slides> element assign the lon and lat values of the first place visited during the Path of Remembrance.
Don't forget to add the Story objects to the derlastories context. (Replace/Relations in Cirilo)
For the thumbnail will also need to add the IMG_1 datastream and the image per hand. Download the image of the first visited place and add it to the datastream. When creating the datastream, make sure to use Managed Content.

Static Project Texts

Static texts like the Imprint are part of the context datastream and are stored in Y:derla. All changes should be done there and then transferred to Cirilo.

Project Extension

The following documentation refers to the concrete extension of the DERLA project by further functions or data pools (e.g. further federal states).

Expansion of the categories of the Map of Remembrance

Changes to "react_map_app" in Zimlab are required to extend the map categories (https://zimlab.uni-graz.at/derla/reactmapapp).
After the repository has been cloned, the node environment must be run.
In reactmap/src/components/App/DERLAControl/index.tsx the new category to be introduced must be added as a DERLACheckbox element:

```
<AccordionItemPanel>
    <DERLACheckbox
    label="Alliierte"
      color="yellow"
      onClick={() => toggleLayer("Alliierte Soldaten")}
      ></DERLACheckbox>
```

```
        <DERLACheckbox
        label="Deserteure"
        color="yellow"
        onClick={() => toggleLayer("Soldaten - Deserteure")}
        ></DERLACheckbox>

        <DERLACheckbox
        label="Kriegsgefangene"
        color="yellow"
        onClick={() => toggleLayer("Soldaten - Kriegsgefangene")}
        ></DERLACheckbox>

        <DERLACheckbox
        label="Wehrmacht"
        color="yellow"
        onClick={() => toggleLayer("Soldaten - Wehrmacht")}
        ></DERLACheckbox>

</AccordionItemPanel>
```

After adding the new category/categories, the change is deployed. The content of the resulting file (copymetoyourURI.tsx) replaces the already existing one in thela-context-geo.xsl, thela-gml.xsl and thela-object.xsl.

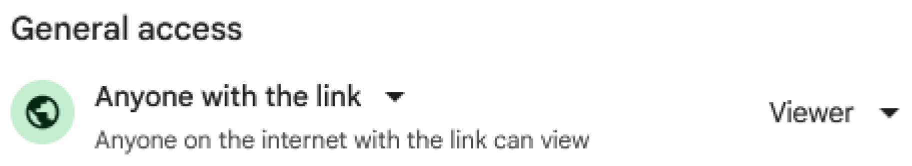
Extension by Federal States

Adding Google Sheets and image folders

To add one or more federal states to DERLA, Google Sheets, which are used to generate the XML files, must be created, as well as image folders in Google Drive at the same level. The naming convention for Google Sheet provides the name of the federal state (e.g. "Tirol"), the associated image folder is named "bundesland_bilder" (e.g. "tirol_bilder").
A Google Sheet consists of the worksheets "Orte", "Personen", "Fixierte Angebote", "Routen/Wege" and "Validierung". The respective title row from an already existing table is copied into the individual worksheets. In almost every worksheet the first line is a mock line to display the way every line should be filled.
The worksheet "Validierung" is copied completely.
In columns where categories are entered during filling (e.g. "Kategorie, Gruppe", "Räumliche Kategorie", ...) the dropdown selection must be inserted. To do this, select the entire column (be careful that the title cell is not selected) and select "Data/Data Validation" in the Google Sheets menu:

- Apply to range: check if column/cell selection is correct
- Criteria: in the "Validierung" worksheet, select the appropriate column
- Done

When selecting between two values (e.g. "ja" and "nein"), these are entered manually. For this purpose, "Dropdown" is selected under "Criteria" and the necessary values are entered. This is necessary, for example, for the columns "Frauen" or "Jugendliche".
After completion of the table, it is set to publicly visible under "Share":


General access

Anyone with the link

Anyone on the internet with the link can view

Viewer

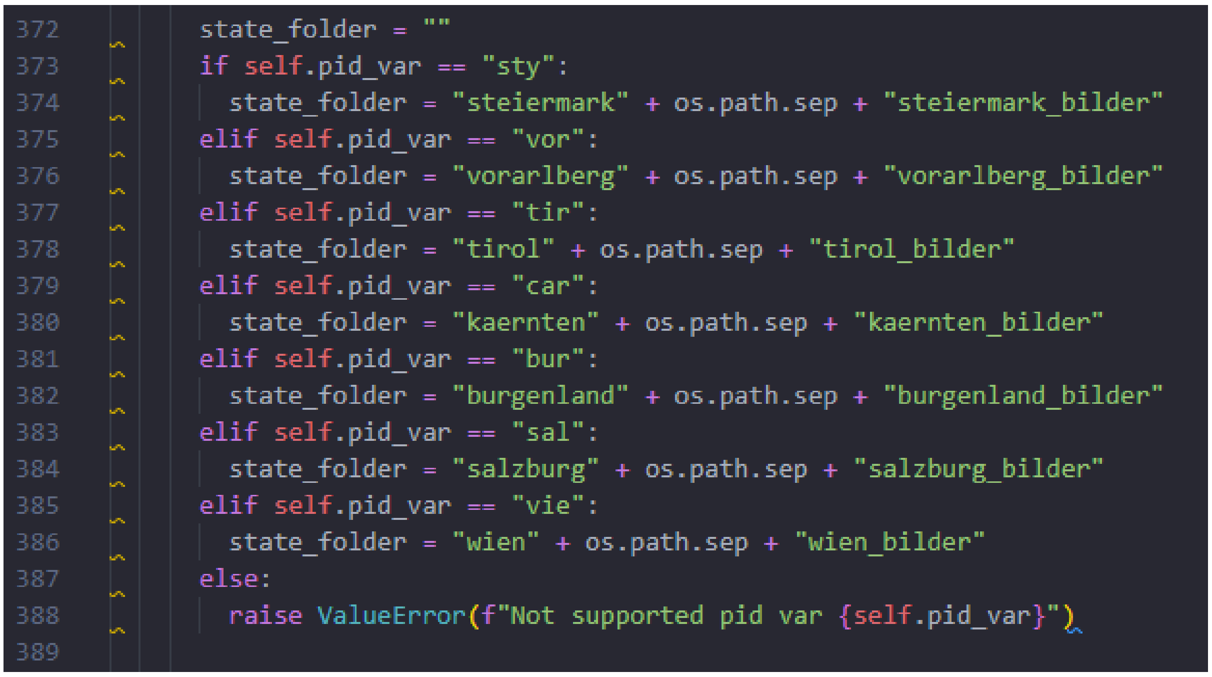
Then the links to the Google Sheets and the image folders on Trello are added to the others in "Bundeslandspezifische Tabellen". For the image folder, only the folder link itself is added, Access is requested from the editors.

The XSLT stylesheets reference the states from time to time (e.g. subpage Paths of Remembrance). Be sure to adapt the code accordingly to include the new categories.

Adding federal state abbreviations in the code

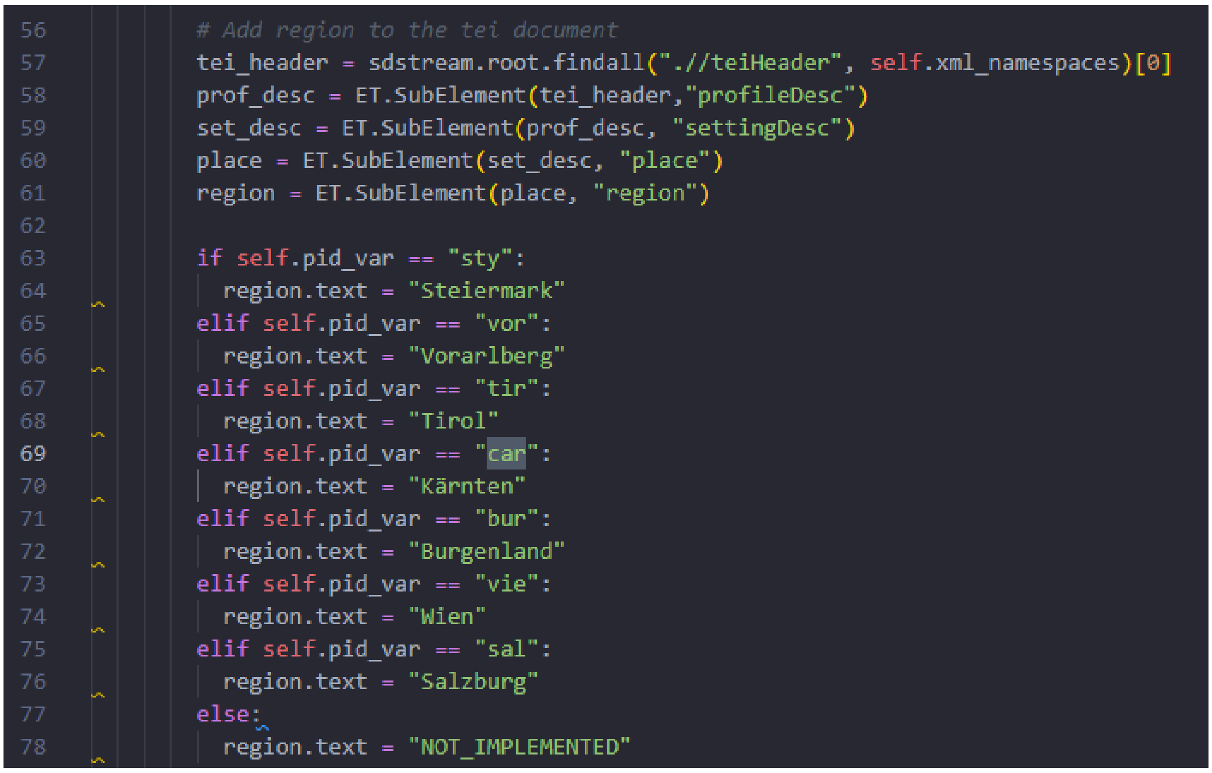
To enable automated data generation for all federal states, the abbreviations and/or corresponding lines of code must be added to the Python scripts in "data_processing" for the new federal states.

DERLATeiLoHandler.py:



```python
state_folder = ""
if self.pid_var == "sty":
    state_folder = "steiermark" + os.path.sep + "steiermark_bilder"
elif self.pid_var == "vor":
    state_folder = "vorarlberg" + os.path.sep + "vorarlberg_bilder"
elif self.pid_var == "tir":
    state_folder = "tirol" + os.path.sep + "tirol_bilder"
elif self.pid_var == "car":
    state_folder = "kaernten" + os.path.sep + "kaernten_bilder"
elif self.pid_var == "bur":
    state_folder = "burgenland" + os.path.sep + "burgenland_bilder"
elif self.pid_var == "sal":
    state_folder = "salzburg" + os.path.sep + "salzburg_bilder"
elif self.pid_var == "vie":
    state_folder = "wien" + os.path.sep + "wien_bilder"
else:
    raise ValueError(f"Not supported pid var {self.pid_var}")

```

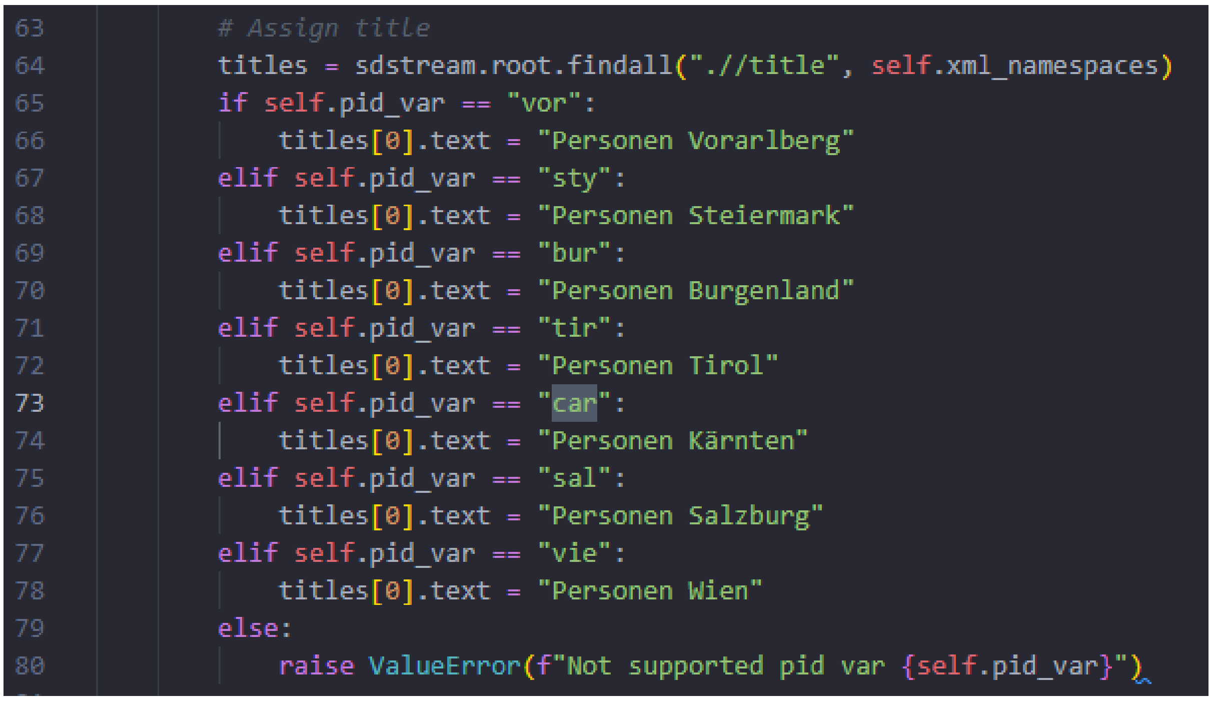
DERLALoListHandler.py:



```
        # Add region to the tei document
        tei_header = sdstream.root.findall("./teiHeader", self.xml_namespaces)[0]
        prof_desc = ET.SubElement(tei_header,"profileDesc")
        set_desc = ET.SubElement(prof_desc, "settingDesc")
        place = ET.SubElement(set_desc, "place")
        region = ET.SubElement(place, "region")

        if self.pid_var == "sty":
            region.text = "Steiermark"
        elif self.pid_var == "vor":
            region.text = "Vorarlberg"
        elif self.pid_var == "tir":
            region.text = "Tirol"
        elif self.pid_var == "car":
            region.text = "Kärnten"
        elif self.pid_var == "bur":
            region.text = "Burgenland"
        elif self.pid_var == "vie":
            region.text = "Wien"
        elif self.pid_var == "sal":
            region.text = "Salzburg"
        else:
            region.text = "NOT_IMPLEMENTED"
```

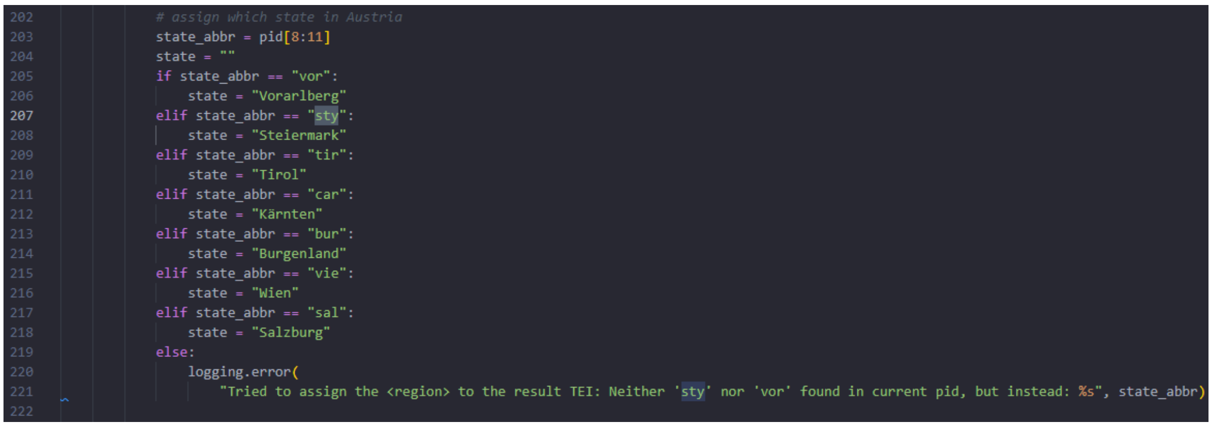
DERLAPersListHandler.py:

Person List



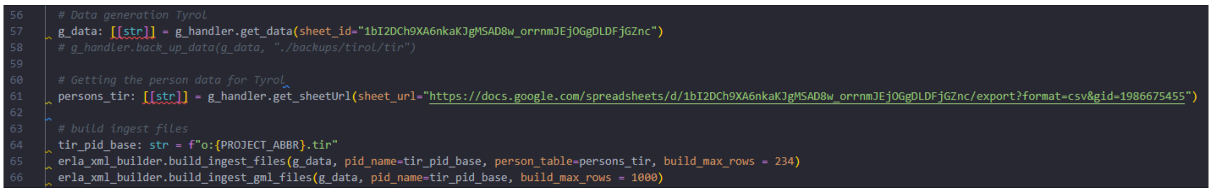
```python
        # Assign title
        titles = sdstream.root.findall(".//title", self.xml_namespaces)
        if self.pid_var == "vor":
            titles[0].text = "Personen Vorarlberg"
        elif self.pid_var == "sty":
            titles[0].text = "Personen Steiermark"
        elif self.pid_var == "bur":
            titles[0].text = "Personen Burgenland"
        elif self.pid_var == "tir":
            titles[0].text = "Personen Tirol"
        elif self.pid_var == "car":
            titles[0].text = "Personen Kärnten"
        elif self.pid_var == "sal":
            titles[0].text = "Personen Salzburg"
        elif self.pid_var == "vie":
            titles[0].text = "Personen Wien"
        else:
            raise ValueError(f"Not supported pid var {self.pid_var}")
```

erla_xml_builder.py:



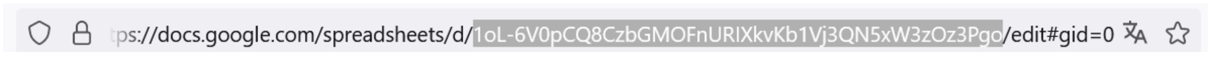
```python
            # assign which state in Austria
            state_abbr = pid[8:11]
            state = ""
            if state_abbr == "vor":
                state = "Vorarlberg"
            elif state_abbr == "sty":
                state = "Steiermark"
            elif state_abbr == "tir":
                state = "Tirol"
            elif state_abbr == "car":
                state = "Kärnten"
            elif state_abbr == "bur":
                state = "Burgenland"
            elif state_abbr == "vie":
                state = "Wien"
            elif state_abbr == "sal":
                state = "Salzburg"
            else:
                logging.error(
                    "Tried to assign the <region> to the result TEI: Neither 'sty' nor 'vor' found in current pid, but instead: %s", state_abbr)

```

main.py:



```python
    # Data generation Tyrol
    g_data: [[str]] = g_handler.get_data(sheet_id="1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc")
    # g_handler.back_up_data(g_data, "./backups/tirol/tir")

    # Getting the person data for Tyrol
    persons_tir: [[str]] = g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc/export?format=csv&gid=1986675455")

    # build ingest files
    tir_pid_base: str = f"o:{PROJECT_ABBR}.tir"
    erla_xml_builder.build_ingest_files(g_data, pid_name=tir_pid_base, person_table=persons_tir, build_max_rows = 234)
    erla_xml_builder.build_ingest_gml_files(g_data, pid_name=tir_pid_base, build_max_rows = 1000)
```

Copy existing code and change the abbreviations to the correct state. At this point it is especially important to adjust the sheet_id variable correctly. Copy it from your browser’s address line:

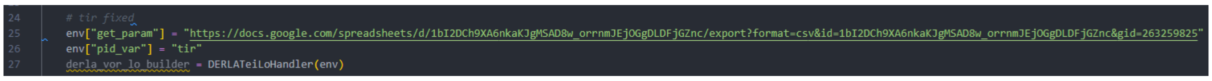


ps://docs.google.com/spreadsheets/d/1oL-6V0pCQ8CzbGMOFnURIXkvKb1Vj3QN5xW3zOz3Pgo/edit#gid=0

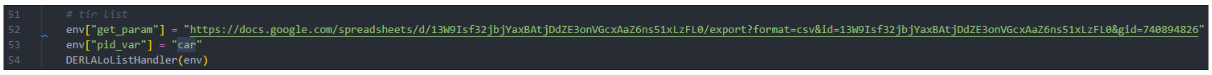
Ingest the GML object also generated separately.

The GML object must be assigned to the state context in Cirilo.

xml_build/archive_of_names/main.py:
Fixed Mediation Offers



```
    # tir fixed
    env["get_param"] = "https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc&gid=263259825"
    env["pid_var"] = "tir"
    derla_vor_lo_builder = DERLATeiLoHandler(env)
```



```
    # tir list
    env["get_param"] = "https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0&gid=740894826"
    env["pid_var"] = "car"
    DERLALoListHandler(env)
```

xml_build/archive_of_names/main.py:
Person List

```
# car persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0/export?format=csv&gid=1026060477"
    env["pid_var"] = "car"
    DERLAPersListHandler(env)
```

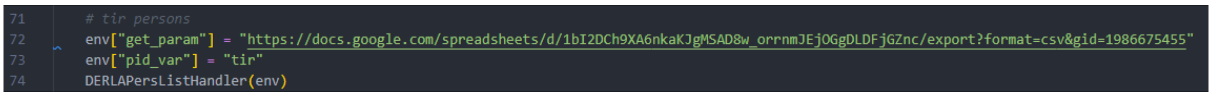
Paths of Remembrance

Adapt the Stylesheet to include the new state in the modes. Be sure to complete the Dublin Core for the new Paths of Remembrance.

Bugs

Image Thumbnails

Some unknown issue leads to the thumbnails being displayed with the wrong orientation (e.g. sideways). To correct this, the JPEG of the image needs to be opened in Photoshop and either exported or saved as a JPEG. No changes to the picture are needed, the process of exporting/saving itself is enough to correct the issue. Don’t forget to rename the images to 1.jpeg before adding them to the datastream “IMAGE.1”.



```
    # tir persons
    env["get_param"] = "https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc/export?format=csv&gid=1986675455"
    env["pid_var"] = "tir"
    DERLAPersListHandler(env)
```

Script and Code Description

erla_xml_builder.py

```
import xml_operations as xml_ops
import erla_templates as erla_templates
import xml.etree.ElementTree as ET
```

```python
import pathlib
import os
import FileReader
from env import *
from shutil import copy
from pathlib import Path
import logging
from xml.dom import minidom


def build_ingest_files(csv_list: [[str]], pid_name: str, person_table:
[[str]], build_max_rows: int = 9999999999):
      """
      'Main' routine that handles the creation of the different
ingestable TEI files
      with according folder + folder-structures.
      :param csv_list Table data which the Digital Object creation
process is dependent on.
      :param pid_name first-part of the "pid" that should be applied.
Param is concatinated to the start if the pid-string.
      :build_max_rows limits the number of digital objects that are
createed according to the row-count of given table.

"""

      logging.debug(
      "Start building TEI based ingest files for: '%s'. Only considering
the first %s rows.", pid_name, build_max_rows)

      row_nmbr = 0
      for row in csv_list:
      # skip xml building at specific number.
      if row_nmbr > build_max_rows:
            logging.info(
                  "Ending build of TEI-ingest files at row_count: %s",
row_nmbr)
            return
      # build xml files according to rows in csv
      # (but skip first 2 because are example values)
      if row_nmbr > 1:
            cur_pid = pid_name + str(row_nmbr + 1)
            # pids start with 1! --> because first row is table header
            csv_build_tei_folder(row, cur_pid, person_table)
            logging.info(
                  "Finished building of TEI-ingest files for: %s",
cur_pid)
```

```
        # lastly increase counter
        row_nmbr = row_nmbr + 1
```

The first function, build_ingest_files, is designed to create ingestable TEI files based on the four input parameters: csv_list, pid_name, person_table, and build_max_rows.
The csv_list parameter is a two-dimensional list (a list of lists) that represents table data, which makes up the content of the TEI. The pid_name parameter is a string that represents the persistent identifier of a given TEI. It is used in the function to write easily understandable logging messages.
The person_table parameter is another two-dimensional list that presumably contains information about persons. The build_max_rows parameter is an integer that limits the number of rows that are iterated. By default, this parameter is set to a very large number (9999999999).
The function starts by logging a debug message indicating the start of the TEI file building process. It then initializes a counter, row_nmbr, to 0. This counter is used to keep track of the current row number in the csv_list.
The function then enters a loop where it iterates over each row in the csv_list. If the row_nmbr exceeds build_max_rows, the function logs an info message and returns, effectively ending the function. If the row_nmbr is greater than 1, the function proceeds to build the XML file according to the rows in the CSV. The first row is skipped because it contains the headings of each row.
Within the loop, the function constructs the current pid by concatenating pid_name and the string representation of row_nmbr + 1. It then calls the csv_build_tei_folder function with the current row, the current pid, and the person_table as arguments. After this, it logs an info message indicating the completion of the TEI-ingest file for the current pid.
Finally, the function increments the row_nmbr counter by 1 before the next iteration of the loop. This ensures that each row in the csv_list is processed in order.

```
def csv_build_tei_folder(csv_row, pid: str, person_table: [[str]]):
    """
    Creates an ingestable TEI-folder for gams for every list entry it gets
    and enriches it with related person data.
    :param csv_row list of string that should be turned into a TEI file.
    :param pid Current pid for the generated TEI file.
    :param person_table Table that should be used for the person-data
enrichment process.
    """
    logging.debug("Start building TEI ingest-files for: %s", pid)

    # first parse the TEI-template for derla (with needed namespaces)
    root = xml_ops.parse_xml(erla_templates.OBJECT_TEI, TEI_NAMESPACES)
    # further down the elements inside the TEI are filled with the
    # data inside given tables.

    # write pid into xml
    root.findall(
        ".//t:idno", {"t": "http://www.tei-c.org/ns/1.0"})[5].text = pid

    # write linked SKOS into keywords element
    key_word_tag = root.findall(
        ".//t:keywords", {"t": "http://www.tei-c.org/ns/1.0"})[0]

    # list_person used multiple times (refernce saved here)
    tei_profile_desc = root.findall(
        ".//t:profileDesc", {"t": "http://www.tei-c.org/ns/1.0"})[0]

    root.find(".//t:geogFeat", {"t": "http://www.tei-c.org/ns/1.0"}).set(
        "ref",
f"{SERVER_ADDRESS}/archive/objects/o:{PROJECT_ABBR}.{pid[8:11]}#{pid[8:]}")

    # main graphic
    # need to save reference here
    main_graphic = None
    sec_graphic = None

    col_nmbr = 0
    for col in csv_row:
        col = col.strip()  # removing leading and ending whitespace
        if col == "":
            logging.warning(
                "No value defined for col with number '%s' at pid '%s'. (Not
even explicit none via '-')", col_nmbr, pid)

        if col_nmbr == 0:
            # skip
            if (col == "") or (col == "-"):
                logging.info("Encountered empty title cell in places table -
skipping generation of ingest files for pid (WILL NOT BE CREATED): " + pid)
                return

            # Titles from table
            root.find("./t:teiHeader/t:fileDesc/t:titleStmt/t:title",
                      {"t": "http://www.tei-c.org/ns/1.0"}).text = col

        if col_nmbr == 1:
            # x coordinate
            validate_coord_length(col, pid)
            root.findall(
                ".//t:geo", {"t": "http://www.tei-c.org/ns/1.0"})[0].text = col

        if col_nmbr == 2:
            # y coordinate
            validate_coord_length(col, pid)
            root.findall(
                ".//t:geo", {"t": "http://www.tei-c.org/ns/1.0"})[1].text = col

        if col_nmbr == 3:
            # nominatim stuff
            idnos = root.findall(
                ".//t:idno", {"t": "http://www.tei-c.org/ns/1.0"})

            # do not consider nominatim link -> too unstable / just use osm number.
            # idno for osm number
            osm_idno = idnos[8]

            # splitting data from the table
            col_split: [str] = col.split(',')

            if len(col_split) > 1:
                osm_nmbr = col_split[1].strip()
                osm_idno.text = osm_nmbr
            elif len(col_split) > 2:
                logging.warning(
                    "cell-value mismatch in nominatim cell: The split size of the 'nominatim-link,osm-number' cell is greater than 2. (Maybe there is another ',' added) For pid: %s", pid)
            else:
                logging.warn(
                    "Cell-value mismatch in nominatim cell: The splitted string has neither length 2 nor greater than 2. The osm-number or the nominatim link might be missing.")

        if col_nmbr == 4:
            # persecution source 01
            kword_id, kword_label = resolve_keywords_id(col)
            ET.SubElement(key_word_tag, "term",
                          ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text =
kword_label


        if (col_nmbr == 5) and (col != "-"):
            if col != "":
                # persecution source 02
                kword_id, kword_label = resolve_keywords_id(col)
                ET.SubElement(key_word_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
            else:
                logging.error("Second persecution source cannot be just an
empty string. Make sure to set it to a value or '-' instead! Skipped
generation of keyword element")

        if (col_nmbr == 6) and (col != "-") and (col != ""):
            # persecution source 03
            if col != "":
                kword_id, kword_label = resolve_keywords_id(col)
                ET.SubElement(key_word_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
            else:
                logging.error("Second persecution source cannot be just an
empty string. Make sure to set it to a value or '-' instead! Skipped
generation of keyword element")


        if col_nmbr == 7:
            # Handling the date of "sign building"
            if(col != '-'):
                # if only year
                if len(col) == 4:
                    geog_feat_tag = root.findall(
                    ".//t:geogFeat", {"t": "http://www.tei-c.org/ns/1.0"})[0]
                    ET.SubElement(geog_feat_tag, "date", type="creation").text
= col
                # if full date take just the year
                else:
                    geog_feat_tag = root.findall(
                    ".//t:geogFeat", {"t": "http://www.tei-c.org/ns/1.0"})[0]
                    ET.SubElement(geog_feat_tag, "date", type="creation").text
= col[-4:]

        if col_nmbr == 8:
            # publicity statement "ja" or "nein"
            if col == "ja":
                location_tag = root.findall(
                ".//t:location", {"t": "http://www.tei-c.org/ns/1.0"})[0]
                desc_tag = ET.SubElement(location_tag, "desc")

                kword_id, kword_label =
resolve_keywords_id("öffentlich_zugänglich")
                ET.SubElement(desc_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
            elif col != "nein":
                logging.warning(
                    "Detected value mismatch in the publicity statement. Got
value: %s. Given value is neither 'ja' nor 'nein'", col)

        if col_nmbr == 9:
            # Spatial Category
            object_type_tag = root.findall(
                ".//t:objectType", {"t": "http://www.tei-c.org/ns/1.0"})[0]
            kword_id, kword_label = resolve_keywords_id(col)
            ET.SubElement(object_type_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
            logging.debug(f"Succesfully created term for spatial category:
{kword_label}")


        if col_nmbr == 10:
            # address
            address_tei = root.findall(
                ".//t:address", {"t": "http://www.tei-c.org/ns/1.0"})[1]
            if col != "-":
                ET.SubElement(address_tei, "addName").text = col

            # assign which state in Austria
            state_abbr = pid[8:11]
            state = ""
            if state_abbr == "vor":
                state = "Vorarlberg"
            elif state_abbr == "sty":
                state = "Steiermark"
            elif state_abbr == "tir":
                state = "Tirol"
            elif state_abbr == "car":
                state = "Kärnten"
            elif state_abbr == "bur":
                state = "Burgenland"
            elif state_abbr == "vie":
                state = "Wien"
            elif state_abbr == "sal":
                state = "Salzburg"
            else:
                logging.error(
                    "Tried to assign the <region> to the result TEI: Neither
'sty' nor 'vor' found in current pid, but instead: %s", state_abbr)

            ET.SubElement(address_tei, "region").text = state

        if col_nmbr == 11:
            # Initiator / Stifter
            # root.findall(".//t:name", {"t":
"http://www.tei-c.org/ns/1.0"})[0].text = col
            if col != "-":
                cur_listPerson = get_tei_listPerson(root, "creators")
                tei_person = ET.SubElement(
                    cur_listPerson, "person", role="initiator")
                ET.SubElement(tei_person, "name").text = col

        if col_nmbr == 12:
            # Künstler
            # root.findall(".//t:name", {"t":
"http://www.tei-c.org/ns/1.0"})[1].text = col
            if col != "-":
                cur_listPerson = get_tei_listPerson(root, "creators")
                tei_person = ET.SubElement(
                    cur_listPerson, "person", role="artist")
                ET.SubElement(tei_person, "name").text = col

        if col_nmbr == 13:
            # description text
            root.findall(".//t:body//t:div//t:p",
                         {"t": "http://www.tei-c.org/ns/1.0"})[0].text = col

        if col_nmbr == 14:
            # optional engraving text
            # root.findall(".//t:body//t:div//t:p", {"t":
"http://www.tei-c.org/ns/1.0"})[1].text = col
            if col != "-" and col != "":
                body = root.findall(
                    ".//t:body", {"t": "http://www.tei-c.org/ns/1.0"})[0]
                inscr_div = ET.SubElement(body, "div", type="inscription")
                ET.SubElement(inscr_div, "p").text = col

        if col_nmbr == 15:
            # women involved
            if col == "ja":
                kword_id, kword_label = resolve_keywords_id("frauen")
                ET.SubElement(key_word_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
                logging.debug("Succesfully added women mentioned term")
            elif col != "nein":
                logging.error(f"Neither 'ja' or 'nein' set for women mentioned
sheet value. Got value: {col}")

        if col_nmbr == 16:
            # Adolescents
            if col == "ja":
                kword_id, kword_label = resolve_keywords_id("jugendliche")
                ET.SubElement(key_word_tag, "term",
ref=f"{DERLA_KEYWORDS_LINK}{kword_id}").text = kword_label
                logging.debug("Succesfully added adolescents mentioned term")
            elif col != "nein":
                logging.error(f"Neither 'ja' or 'nein' set for adolescents
mentioned sheet value. Got value: {col}")

        if col_nmbr == 17:
            # Image description
            facs = root.find(
                "./t:facsimile", {"t": "http://www.tei-c.org/ns/1.0"})
            graphic = ET.SubElement(
                facs, "graphic", mimeType="image/jpeg", url="file:///1.JPG")
            # first image is popup / main image!
            graphic.set("xml:id", "IMAGE.1")
            ET.SubElement(graphic, "desc").text = col
            main_graphic = graphic  # save reference for next iteration in
outer scope

            # second image
            graphic02 = graphic = ET.SubElement(
                facs, "graphic", mimeType="image/jpeg", url="file:///2.JPG")
            # first image is popup / main image!
            graphic02.set("xml:id", "IMAGE.2")
            ET.SubElement(graphic, "desc").text = col
            sec_graphic = graphic02

        if col_nmbr == 18:
            # Main Image orientation
            main_graphic.set("style", col)

        if col_nmbr == 19:
            # Sec image orientation
            sec_graphic.set("style", col)

        col_nmbr = col_nmbr + 1

    logging.debug("Finished building tei-xml for place: %s", pid)

    # Enrich TEI with person data.
    apply_victims_to_tei(pid, root, person_table)

    # Adding links to linked Vermittlungsangebote
    apply_fixed_los_to_tei(pid, root)

    # writing xml files in correct folders
    # first make folders per place
    object_name = str(pid).replace(f"o:{PROJECT_ABBR}.", "")
    results_path = str(os.getcwd()) + "/results/"
    object_folder_path = results_path + object_name
    pathlib.Path(object_folder_path).mkdir(parents=True, exist_ok=True)
    # then write tei-xml into related folder.
    tei_path = "results/" + object_name + "/" + object_name + ".xml"
    xml_to_file(tei_path, root)

    logging.info(
        "Wrote TEI with pid '%s' to the filesystem at path: %s ", pid,
tei_path)

    # copy img files into related tei-place-folder
    handle_graphics(pid)
```
The function csv_build_tei_folder, is used to create an ingestable TEI folder for GAMS. The function takes in a row from a CSV file, a PID (Persistent Identifier), and a table of person data. It then uses this data to populate an XML template with the necessary information.
The function starts by parsing the TEI template and then proceeds to fill in the elements with data from the provided tables. The PID is written into the XML, and linked SKOS (Simple Knowledge Organization System) are written into the keywords element.
The function then iterates over each column in the CSV row, performing different actions based on the column number. These actions include validating coordinates, handling dates, assigning states, and adding terms to the keywords element.
After all the columns have been processed, the function enriches the TEI with person data, adds links to linked Mediation Offers, and writes the XML files in the correct folders. It also handles the copying of image files into the related TEI-place-folder.
The function uses logging to provide information about its progress and any issues that arise.

```
def handle_graphics(pid: str):
    """
    Searches files in sampleData directory and copies it to the folders
containing the tei-xml.
    :param pid: pid of current object to generate.
    :return: nothing.
    """

    working_dir_path = str(os.getcwd())
    home_path = str(Path.home())
    data_path = f"G:\\Meine
Ablage\\01_Berufliches\\01_Projekte\\02_digitale_erinnerungslandschaft\\03_dat
a\\"
    # specify folder for output.
    results_path = working_dir_path + "\\results\\"
    fr = FileReader.FileReader()
    logging.debug("Copying main image file from folder '%s' to '%s'",
                  data_path, results_path)

    state_abbr = pid[8:11]
    if len(state_abbr) != 3:
        logging.error(
            "The state abbreviation in the pid should exactly have 3
characters. But got: '%s'", state_abbr)
```

```
    pid_num = pid[11:]
    state_and_num = pid[8:]

    # set according to given pid.
    cur_img_path = ""
    if state_abbr == "sty":
        cur_img_path = data_path + "steiermark\\steiermark_bilder\\" + pid_num
+ "\\img\\"
    elif state_abbr == "vor":
        cur_img_path = data_path + "vorarlberg\\vorarlberg_bilder\\" + pid_num
+ "\\img\\"
    elif state_abbr == "tir":
        cur_img_path = data_path + "tirol\\tirol_bilder\\" + pid_num +
"\\img\\"
    elif state_abbr == "bur":
        cur_img_path = data_path + "burgenland\\burgenland_bilder\\" + pid_num
+ "\\img\\"
    elif state_abbr == "car":
        cur_img_path = data_path + "kaernten\\kaernten_bilder\\" + pid_num +
"\\img\\"
    elif state_abbr == "sal":
        cur_img_path = data_path + "salzburg\\salzburg_bilder\\" + pid_num +
"\\img\\"
    elif state_abbr == "vie":
        cur_img_path = data_path + "wien\\wien_bilder\\" + pid_num + "\\img\\"
    else:
        logging.error(
            "A state abbreviation aside from vor,sty,tir,bur,car,sal, vie is
not supported, but got: '%s'", state_abbr)

    # where to write image
    img_target_path = results_path + state_and_num
    try:
        img_list = fr.listFiles(cur_img_path)
        copy(cur_img_path + "\\1.JPG", img_target_path + "\\1.JPG")
        logging.info("Succesfully copied main image for '%s' to path: '%s'",
pid, img_target_path)
        copy(cur_img_path + "\\2.JPG", img_target_path + "\\2.JPG")
        logging.info("Succesfully copied secondary image for '%s' to path:
'%s'", pid, img_target_path)

    except FileNotFoundError:
        logging.error(
            "There are missing image files inside for pid '%s' inside folder:
%s", pid, cur_img_path)
        return
```
The function handle_graphics is designed to handle image files related to a specific object identified by a PID (Persistent Identifier). The function takes one parameter, pid, which is the identifier of the current object for which the images are to be handled.
The function begins by defining several paths. The working_dir_path is the current working directory of the script, and home_path is the home directory of the user. The data_path is a hard-coded path where the data is stored, and results_path is the directory where the results (i.e., the copied images) will be stored.
A FileReader object is created, which is a custom class used to read files and defined in the file FileReader.py. The function then logs a debug message indicating that it's about to copy the main image file from the data_path to the results_path.
The function extracts the state abbreviation and the number from the pid. The state abbreviation is the substring from the 8th to the 11th character of the pid, and the number is the substring from the 11th character to the end. If the state abbreviation is not exactly 3 characters long, an error message is logged.
The function then determines the current image path (cur_img_path) based on the state abbreviation. If the state abbreviation is not one of the expected values ("sty", "vor", "tir", "bur", "car", "sal", "vie"), an error message is logged.
The target path for the image is then set to be the results_path concatenated with the state and number extracted from the pid.
The function then attempts to list all the files in the cur_img_path and copy the first and second image (named "1.JPG" and "2.JPG") from the cur_img_path to the img_target_path. If the images are successfully copied, an info message is logged for each image. If the images are not found (i.e., a FileNotFoundError is raised), an error message is logged, and the function returns.

```
def build_ingest_gml_files(csv_list, pid_name: str, build_max_rows: int =
9999999999):
    root = xml_ops.parse_xml(erla_templates.OBJECT_GML, GML_NAMESPACES)
    root.find(".//gdas:PID", GML_NAMESPACES).text = pid_name
    row_nmbr = 0
    logging.debug(
        "Start to build GML file for pid '%s'. Using only the first %s rows.",
pid_name, build_max_rows)

    for row in csv_list:
        # first two table entries -> header + example should be skipped
        # also optionally limits the number of assigned features into the GML.
        if (row_nmbr > 1) and (build_max_rows > row_nmbr):
            cur_pid = pid_name + str(row_nmbr + 1)
            # pids start with 2! --> because first row is table header
            build_gml_feature(row, cur_pid, root)
        row_nmbr = row_nmbr + 1

    # writing xml files in correct folders
    object_name = str(pid_name).replace(f"o:{PROJECT_ABBR}.", "") + "_gml"
    results_path = str(os.getcwd()) + "/results/"
    object_folder_path = results_path
    pathlib.Path(object_folder_path).mkdir(parents=True, exist_ok=True)

    # tree = ET.ElementTree(root)
    gml_file_name = "results/" + object_name + ".xml"
    xml_to_file(gml_file_name, root)
    logging.info("Wrote GML file for '%s' to '%s'", pid_name, gml_file_name)
```

The build_ingest_gml_files function is used to create GML files from a given CSV list. The function takes three parameters: csv_list, pid_name, and build_max_rows. The csv_list is a list of rows from a CSV file, pid_name is a string that represents the Persistent Identifier, and build_max_rows is an optional parameter that limits the number of rows from the CSV file to be processed.
The function begins by parsing an XML template and setting the PID in the root element of the parsed XML. It then iterates over each row in the csv_list. The first row is skipped, and the function also checks if the current row number exceeds the build_max_rows limit. If the row number is within the limit, the function calls build_gml_feature to build a GML feature for the current row and appends it to the root element of the XML.
The build_gml_feature function takes a row from the CSV file, a PID, and the root element of an XML document as parameters. It adds a feature to the GML document according to the data in the CSV row. The function iterates over each column in the CSV row and performs different actions based on the column number. These actions include setting the title, coordinates, nominatim link, persecution causes, date, publicity, spatial category, address, description, and information about women and adolescents involved.
After all the rows have been processed, the function writes the XML document to a file in the 'results' directory. The filename is derived from the pid_name and appended with '_gml'. The function uses the xml_to_file function to write the XML document to the file. The xml_to_file function takes a filename and the root element of an XML document as parameters. It converts the XML document to a string, formats it with indents for readability, and writes it to the file.

```
def build_gml_feature(csv_row, pid: str, root):
    """
    Adds to given gml document features according to rows of given csv and pid
    given as parameter.
    :param csv_row: csv list of the places
    :param pid: pid to assign to each place
    :param root: root of gml document to create.
    :return:
    """
    logging.debug("Building GML features for: %s", pid)

    if(csv_row[0] == "-") or (csv_row[0] == ""):
        logging.info("Ignoring GML featureMemeber generation (title cell is
empty or '-') for pid " + pid)
        return

    # for every row create a feature Member
    feature_member = ET.SubElement(root, "gml:featureMember", fid=pid[8:])
    derla_place = ET.SubElement(
        feature_member, f"{PROJECT_ABBR}:placeOfRemembrance")
    derla_geom = ET.SubElement(derla_place, "gdas:geometryProperty")
    gml_point = ET.SubElement(derla_geom, "gml:Point", srsName="EPSG:3857")
    gdas_keywords = ET.SubElement(derla_place, "gdas:keywords")

    # first assign pid
    ET.SubElement(derla_place, 'gdas:ref').text = pid

    col_nmbr = 0
    for col in csv_row:
        col = col.strip()  # removing leading and ending whitespace
        if col == "":
            logging.warn(
                "GML: No value defined for col with number '%s' at pid '%s'.
(Not even explicit none via '-')", col_nmbr, pid)

        if col_nmbr == 0:
            # Titles from table
            ET.SubElement(derla_place, "gdas:title").text = col

        if col_nmbr == 1:
            # x coordinate
            gml_point.text = col

        if col_nmbr == 2:
            # y coordinate
            gml_point.text = gml_point.text + " " + str(col)

        if col_nmbr == 3:
            # nominatim stuff
            if col != "" and col != '-':
                split = col.split(", ")
                if len(split) > 1:
                    ET.SubElement(derla_place, "gdas:osm").text = split[1]
                else:
                    logging.error(f"Malformed nominatim link - osm number 
combination detected, at pid {pid}. Got col value: {str(split)}")

        if col_nmbr == 4:
            # handling persecution causes as categories
            # col_split: [str] = col.split(',')
            # category_name = col_split[0].strip()
            # ET.SubElement(derla_place, "gdas:layer").text = 
resolve_category_label(category_name)

            # if len(col_split) > 1:
            #     group = col_split[1].strip()
            #     ET.SubElement(derla_place, "gdas:filterTerm").text = 
resolve_category_label(group)
            # elif len(col_split) > 2:
            #     logging.warning("GML: PersecutionSource - Category-Mismatch? 
Column split into more than 2 separate values. Pid: %s", pid)

            # fill gdas layer property
            handle_gdas_layer(col, derla_place)
            # create gdas keywords property
            handleGdasKeywords(col, derla_place, gdas_keywords)

        if col_nmbr == 5:
            # create gdas keywords property
            if (col != "-") and (col != ""):
                handleGdasKeywords(col, derla_place, gdas_keywords)
            else:
                logging.error(f"Malformed entry at third prosecution type. Got 
value: {col}")

        if col_nmbr == 6:
            # create gdas keywords property
            if (col != "-") and (col != ""):
                handleGdasKeywords(col, derla_place, gdas_keywords)
            else:
                logging.error(f"Malformed entry at third prosecution type. Got 
value: {col}")
```

```
        if col_nmbr == 7:
            # Handling the date
            if(len(col) == 4) or (col == "-"):
                ET.SubElement(derla_place, "gdas:date").text = col
            else:
                ET.SubElement(derla_place, "gdas:date").text = col[-4:]

        if col_nmbr == 8:
            # publicity
            if col == "ja":
                kword_id, kword_label =
resolve_keywords_id("öffentlich_zugänglich")
                link = DERLA_KEYWORDS_LINK
                ET.SubElement(
                    derla_place, "gdas:publicAvailable",
ref=f"{link}{kword_id}").text = kword_label
            elif col == "nein":
                pass
            else:
                logging.warning(
                    "Detected value mismatch in the publicity statement for
place: %s. Given value is neither 'ja' nor 'nein'", )

        if col_nmbr == 9:
            # Spatial Category
            kword_id, kword_label = resolve_keywords_id(col)
            link = DERLA_KEYWORDS_LINK
            ET.SubElement(
                derla_place, "gdas:type", ref=f"{link}{kword_id}").text =
kword_label

        if col_nmbr == 10:
            # address
            ET.SubElement(derla_place, "gdas:address").text = col

        # if col_nmbr == 11:
        #     # Initiator / Stifter

        # if col_nmbr == 12:
        #     # artist

        if col_nmbr == 13:
            # Text / description of
            ET.SubElement(derla_place, "gdas:desc").text = col

        # if col_nmbr == 14:
        #     # Inschrift

        if col_nmbr == 15:
            # Frauen
            # women involved ?
            if col == "ja":
                kword_id, kword_label = resolve_keywords_id("frauen")
                link = DERLA_KEYWORDS_LINK
                ET.SubElement(
                    gdas_keywords, "gdas:keyword", type="women",
ref=f"{link}{kword_id}").text = kword_label
            elif col != "nein":
                logging.error(f"Value mismatch at women involved stmt. Expeted
'ja' or 'no' but got: {col}")

        if col_nmbr == 16:
            # Adolescents
            if col == "ja":
                kword_id, kword_label = resolve_keywords_id("jugendliche")
                link = DERLA_KEYWORDS_LINK
                ET.SubElement(
                    gdas_keywords, "gdas:keyword", type="adolescents",
ref=f"{link}{kword_id}").text = kword_label
            elif col != "nein":
                logging.error(f"Value mismatch at adolescents involved stmt.
Expected 'ja' or 'no' but got: {col}")

        # if col_nmbr == 17:
        #     # Hauptbildunterschrift

        # if col_nmbr == 18:
        #     # Hauptbildorientierung

        col_nmbr = col_nmbr + 1
```
The build_gml_feature function in Python is designed to add features to a given GML document according to the rows of a given CSV and a PID provided as a parameter. The function takes three parameters: csv_row, pid, and root. csv_row is a list representing a row

from a CSV file, pid is a string representing the PID to assign to each place, and root is the root of the GML document to be created.
The function starts by logging a debug message indicating that it's building GML features for the given PID. It then checks if the first cell of the csv_row is either empty or contains a dash ("-"). If it does, the function logs an info message stating that it's ignoring the GML feature member generation for the given PID and returns.
If the first cell of the csv_row is not empty or a dash, the function proceeds to create a feature member for each row. It creates several XML sub-elements under the root and the feature member, including a gml:Point for the coordinates and a gdas:keywords element for the keywords.
The function then iterates over each column in the csv_row, performing different actions based on the column number. These actions include setting the title, coordinates, nominatim link, persecution causes, date, publicity, spatial category, address, description, and information about women and adolescents involved. The function uses helper functions like handle_gdas_layer and handleGdasKeywords to handle specific columns.
If a column is empty, the function logs a warning message. If a column contains a value that doesn't match the expected format or values, the function logs an error or warning message.
After processing all the columns, the function increments the column number and continues with the next column.

```python
def resolve_category_label(key: str) -> str:
    """
    Looks up given value inside a local dictionary and returns the
    related corrected label.
    :param key Key that should be looked ip
    :returns Label related to key.
    """

    value_dict = {
        "widerstand": "Widerstand",
        "politisch": "Politisch",
        "religiös": "Religiös",
        "individuell": "Individuell",
        "jüdischeopfer": "Jüdische Opfer",
        "gemeindeeinrichtung": "Jüdische Gemeinde",
        "sammelwohnung": "Sammelwohnungen",
        "kz": "KZ",
        "ikg": "als Jude verfolgt",
        "todesmarsch": "Todesmarsch",
        "roma": "Roma/Romnija und Sinti/Sintize und andere Fahrende wie
Lovara",
        "euthanasie": "NS-Euthanasie",
        "zwangsarbeiter": "ZwangsarbeiterInnen",
        "terror": "Orte des Terrors",
        "gestapossnsdap": "Gestapo/SS/NSDAP",
        "hinrichtungsstätten": "Hinrichtungsstätten",
        "gefängnis": "Gefängnis",
        "soldaten": "Soldaten",
        "wehrmacht": "Wehrmacht",
        "deserteure": "Deserteure",
        "alliierte": "Alliierte Soldaten",
        "zivileopfer": "Zivile Opfer",
        "homosexuelle": "Homosexuelle Opfer",
        "kollektiv": "Kollektiv"
    }

    return_val = ""
    try:
        return_val = value_dict[key]
    except KeyError:
        logging.error(
            "Persecution category not found in dictionary, therefore can also
not resolve it's label. Given category: '%s'. Applying an empty string
instead", key)
        return_val = ""

    return return_val
```

The function resolve_category_label is designed to map a given key to a corresponding label.
The function takes one parameter, key, which is the key that should be looked up in the dictionary. The dictionary, value_dict, is defined within the function and maps keys to their corresponding labels. The keys are in lowercase and without spaces, while the labels are more human-readable with proper capitalization and spaces.
The function initializes a variable, return_val, to an empty string. It then tries to look up the key in the value_dict. If the key is found in the dictionary, the corresponding label is assigned to return_val.
If the key is not found in the dictionary (which would raise a KeyError), the function logs an error message indicating that the persecution category was not found in the dictionary and that it cannot resolve its label. In this case, return_val remains an empty string.
Finally, the function returns return_val, which will be the label corresponding to the given key if the key was found in the dictionary, or an empty string if the key was not found.

```
def resolve_placetype_label(key: str) -> str:
    """
    Resolves the name of the placeytpe used in the DERLA project.
    :param key name to be resolved
    :returns label of placename
    """
    placetypes_dict = {
        "ohnezeichen": "Ohne Zeichen",
        "passage": "Passage",
        "platz": "Platz",
        "strasse": "Straße",
        "brücke": "Brücke",
        "gasse": "Gasse",
        "verkehrsfläche": "Verkehrsfläche",
        "sammelwohnung": "Sammelwohnung",
        "gebäude": "Gebäude",
        "stolperstein": "Stolperstein",
        "grabanlage": "Grabanlage",
        "grab": "Grab",
        "denkmal": "Denkmal",
        "freimonument": "Freistehendes Monument",
        "skulptur": "Skulptur",
        "gedenkstein": "Gedenkstein",
        "statue": "Statue",
        "glasfenster": "Glasfenster",
        "schriftskulptur": "Schriftskulptur",
        "kriegerdenkmal": "Kriegerdenkmal",
        "gedenktafel": "Gedenktafel",
        "intervention": "Intervention",
        "siedlung": "Siedlung"
    }

    return_val = ""
    try:
        return_val = placetypes_dict[key]
    except KeyError:
        logging.error(
            "Persecution group not found in dictionary, therefore can also not
resolve it's label. Given category: %s. Applying an emtpy string instead",
key)
        return_val = ""

    return return_val
```
The function resolve_placetype_label is designed to map a given key to a corresponding label and has a similar purpose to the last function.

The function initializes a variable, return_val, to an empty string. It then tries to look up the key in the placetypes_dict. If the key is found in the dictionary, the corresponding label is assigned to return_val.
If the key is not found in the dictionary (which would raise a KeyError), the function logs an error message indicating that the persecution group was not found in the dictionary and that it cannot resolve its label. In this case, return_val remains an empty string.
Finally, the function returns return_val, which will be the label corresponding to the given key if the key was found in the dictionary, or an empty string if the key was not found.

```python
def apply_victims_to_tei(pid: str, root: ET.Element, victim_table: [[str]]):
    logging.debug("Start victim enrichment process for pid: %s", pid)
    pid_num = pid[11:]

    # partic desc used for victims
    # tei_partic_desc = None

    # used per place
    tei_victims_list = None

    # loop through victims
    for victim_index, victim_row in enumerate(victim_table):
        # TODO error handling -> more robust against nonsense!
        place_refs = victim_row[2].split(";")
        logging.debug("Split victim row into: %s", str(place_refs))

        # skip first two rows (are example rows)
        if victim_index < 2:
            continue

        # remove whitespaces
        if " " in victim_row[2]:
            victim_row[2] = victim_row[2].replace(" ", "")

        # apply data only if current place is referenced
        # in specific column.
        if str(pid_num) in place_refs:
            logging.info("Found matching place reference in current victim row
for pid: '%s'. %s",
                         pid, "Adding now vicim data to the TEI.")

            # create victims list if not available
            if not tei_victims_list:
                logging.debug(
                    "No tei:listPerson found for current person and therefore 
added one.")
                tei_victims_list = get_tei_listPerson(root, "victims")

            # add basic person data (that everyone must have at least)
            tei_cur_person = ET.SubElement(tei_victims_list, "person")
            tei_cur_pers_name = ET.SubElement(tei_cur_person, "persName", ref
= f"{SERVER_ADDRESS}/o:derla.pers{pid[8:11]}#pers{pid[8:11]}{victim_index +
1}")



            # tei_cur_list_event = ET.SubElement(tei_cur_person, "listEvent")

            # assign data from table
            col_nmbr = 0
            for victim_col in victim_row:
                if victim_col == "":
                    logging.warn(
                        "TEI-Person_Processing: No value defined for col with
number '%s' at pid '%s'. (Not even explicit none via '-')", col_nmbr, pid)

                # surname
                if col_nmbr == 0:
                    ET.SubElement(tei_cur_pers_name,
                                  "surname").text = victim_col
                    logging.debug("Added person's surname: %s", victim_col)

                # forename
                if col_nmbr == 1:
                    ET.SubElement(tei_cur_pers_name,
                                  "forename").text = victim_col
                    logging.debug("Added person's forename: %s", victim_col)

                # linked places (not used here / only needed to "know"
                # about linked references)
                if col_nmbr == 2:
                    pass

                # Mädchenname
                if col_nmbr == 3:
                    if(victim_col != '-'):
                        ET.SubElement(tei_cur_pers_name,
                                      "addName").text = victim_col

                # alternative Schreibweise
                if col_nmbr == 4:
                    if(victim_col != '-'):
                        ET.SubElement(tei_cur_pers_name, "addName",
                                      type="alternative_surname").text =
victim_col


                # Freitext
                if col_nmbr == 5:
                    pass
                    # tei_list_event_head = ET.SubElement(
                    #     tei_cur_list_event, "head")
                    # ET.SubElement(tei_list_event_head,
                    #               "desc").text = victim_col

                # Kommentar intern
                # nice
                # nice 02
                if col_nmbr == 6:
                    pass

                # GND
                # moved to later
                if col_nmbr == 7:
                    pass

                # VIAF
                if col_nmbr == 8:
                    pass

                # Wiki-Data
                if col_nmbr == 9:
                    pass

                # category + group

                if col_nmbr == 10:
                    pass
                    # if(victim_col != '-'):
```

```
                    #     col_split: [str] = victim_col.split(',')
                    #     category_name = col_split[0].strip()

                    #     cur_event = ET.SubElement(tei_cur_list_event,
"event", {
                    #                                 "xml:id":
f"victim_event01_{str(victim_index)}", "type": category_name})
                    #     cur_event_label = ET.SubElement(cur_event, "label")
                    #     ET.SubElement(cur_event_label, "term",
type="category",
                    #
ref=f"{TEI_NAMESPACES['tr']}{category_name}").text =
resolve_category_label(category_name)

                    #     if len(col_split) > 1:
                    #         group = col_split[1].strip()
                    #         group_label = ET.SubElement(cur_event, "label")
                    #         ET.SubElement(group_label, "term", type="group",
                    #
ref=f"{TEI_NAMESPACES['tr']}{group}").text = resolve_category_label(group)
                    #         cur_event.set("subtype", group)
                    #     elif len(col_split) > 2:
                    #         logging.warning(
                    #             "TEI-PersonProcessing - Category-Mismatch?
Column split into more than 2 separate values. Pid: %s", pid)

                # additional category + group
                if col_nmbr == 11:
                    pass

                # weiblich
                if col_nmbr == 12:
                    pass
                    # tei_sex = ET.SubElement(tei_cur_person, "sex")
                    # if victim_col == "ja":
                    #     ET.SubElement(tei_sex, "term", type="skos:Concept",
                    #
ref=f"{TEI_NAMESPACES['tr']}{victim_col}").text = "Frauen explizit erwähnt"
                    # elif victim_col == "nein":
                    #     victim_col = "womenNotMentioned"
                    #     ET.SubElement(tei_sex, "term", type="skos:Concept",
                    #
ref=f"{TEI_NAMESPACES['tr']}{victim_col}").text = "Frauen nicht explizit
erwähnt"

                # jugendlich
                if col_nmbr == 13:
                    pass
                    # tei_age_tag = ET.SubElement(tei_cur_person, "age")
                    # if victim_col == "ja":
                    #     victim_col = "adolescentsMentioned"
                    #     ET.SubElement(tei_age_tag, "term",
type="skos:Concept",
                    #
ref=f"{TEI_NAMESPACES['tr']}{victim_col}").text = "Jugendliche explizit
erwähnt"
                    # elif victim_col == "nein":
                    #     victim_col = "adolescentsNotMentioned"
                    #     ET.SubElement(tei_age_tag, "term",
type="skos:Concept",
                    #
ref=f"{TEI_NAMESPACES['tr']}{victim_col}").text = "Jugendliche nicht explizit
erwähnt"


                # Geburtsdatum
                if col_nmbr == 14:
                    if(victim_col != '-'):
                        birth_tag = ET.SubElement(tei_cur_person, "birth")
                        ET.SubElement(birth_tag, "date").text = victim_col

                # Todesdatum
                if col_nmbr == 15:
                    if(victim_col != '-'):
                        birth_tag = ET.SubElement(tei_cur_person, "death")
                        ET.SubElement(birth_tag, "date").text = victim_col

                # Bildunterschrift
                if col_nmbr == 16:
                    pass

                # Bildorientierung
                if col_nmbr == 17:
                    pass
                if col_nmbr == 18:
                    pass

                col_nmbr = col_nmbr + 1

            logging.debug(
                "Ended victim enrichment process for person belonging to the
pid: %s", pid)
```

The function apply_victims_to_tei is used to enrich a TEI XML document with data about victims. The function takes three parameters: a PID, the root of the XML document, and a table of data about victims.
The function starts by extracting the numeric part of the PID and initializing a variable, tei_victims_list, to None. This variable will later hold the XML element that contains the list of victims.
The function then loops through each row in the table containing the victims. For each row, it splits the third column into a list of place references and removes any whitespace from this column. If the current place is referenced in this column, the function proceeds to add the data about the victims to the TEI.
If tei_victims_list is None, the function calls get_tei_listPerson to create a new listPerson element in the TEI and assigns this element to tei_victims_list.
The function then creates a new person element within tei_victims_list and a persName element within this person element. The ref attribute of the persName element is set to a URL that includes the PID and the index of the current victim.
The function then loops through each column in the victim row and performs different actions based on the column number. These actions include adding the victim's surname and forename, adding an alternative name, and adding birth and death dates. For each of these actions, the function creates a new XML element within the persName or person element and sets its text to the value from the corresponding column in the victim row.
The function also includes several commented-out sections of code that show how additional data could be added to the TEI, such as event lists, sex, age, and categories.
Finally, the function logs a message to indicate that the victim enrichment process has ended for the current person.

```
def validate_coord_length(coord: str, pid):
    if len(coord) > 9:
        logging.warning(
            "Wrong Coordinate: Given coord count is greater than 9. %s", pid)
```

The validate_coord_length function is used to check the length of a given coordinate string. It takes two parameters: coord, which is a string representing a coordinate, and pid. If the length of the coord string is greater than 9, the function logs a warning message indicating that the given coordinate count is greater than 9 and includes the pid in the message. This function is used to ensure that coordinates are in a valid format before they are used in further processing.

```
def get_tei_listPerson(root, listPersType: str):
    """
    Checks if particDesc and listPerson exists. Create particDesc and
listPerson if not
    there and returns the tei_list with given listPersType.
    """

    tei_profile_desc = root.findall(
        ".//t:profileDesc", {"t": "http://www.tei-c.org/ns/1.0"})[0]
    tei_particDesc = None
    try:
        tei_particDesc = root.findall(
            ".//t:particDesc", {"t": "http://www.tei-c.org/ns/1.0"})[0]
    except:
        tei_particDesc = ET.SubElement(tei_profile_desc, "particDesc")

    tei_listPerson = None
    try:
        tei_listPerson = root.findall(
            f".//t:listPerson[@type='{listPersType}']", {"t":
"http://www.tei-c.org/ns/1.0"})[0]
    except:
        tei_listPerson = ET.SubElement(
            tei_particDesc, "listPerson", type=listPersType)

    return tei_listPerson
```

The get_tei_listPerson function is used to retrieve or create a listPerson element in a TEI XML document. The function takes two parameters: root, which is the root of the XML document, and listPersType, which is a string representing the type of the listPerson element.
The function first finds the profileDesc element in the XML document. It then tries to find a particDesc element. If no particDesc element is found, the function creates a new particDesc element as a child of the profileDesc element.
The function then tries to find a listPerson element with the type specified by listPersType. If no such listPerson element is found, the function creates a new listPerson element with the specified type as a child of the particDesc element.
Finally, the function returns the listPerson element. This function is used to ensure that a listPerson element of a specific type exists in the XML document before adding person data to it.

```
def handleGdasKeywords(col: str, derla_place: ET.Element, gdas_keyword:
ET.Element):
    """
    Creates the correct gdas keywords string from gsheets table
    :param col string value of current gsheet cell
    :param derla_place parent element to which the keywords tag should be
added.
    """


    kword_id, kword_label = resolve_keywords_id(col)
    link = DERLA_KEYWORDS_LINK
    ET.SubElement(gdas_keyword, "gdas:keyword", ref=f"{link}{kword_id}",
type="prosecution").text = kword_label
```

The handleGdasKeywords function is used to create a gdas:keyword element in a GDAS XML document. The function takes three parameters: col, which is a string value of the current cell in a Google Sheets table, derla_place, which is the parent XML element to which the gdas:keyword element should be added, and gdas_keyword, which is the gdas:keyword XML element.
The function starts by calling the resolve_keywords_id function (explained later on) to get the ID and label of the keyword corresponding to the value in col. It then creates a new gdas:keyword element as a child of gdas_keyword, sets its ref attribute to a URL that includes the keyword ID, sets its type attribute to "prosecution", and sets its text to the keyword label.

```
def handle_gdas_layer(col: str, derla_place: ET.Element):
    """
    Creates the correct gdas layer string from gsheets table
    :param col string value of current gsheet cell
    :param derla_place parent element to which the keywords tag should be
added.
    """
    # col_refactored: [str] = col.replace(",", " |")
    kword_id, kword_label = resolve_keywords_id(col)
    #col_refactored = res_prosecution_type_label(col_refactored)
    link = DERLA_KEYWORDS_LINK
    ET.SubElement(derla_place, "gdas:layer", ref = f"{link}{kword_id}").text =
kword_label
```

The handle_gdas_layer function is used to create a gdas:layer element in a GDAS XML document. The function takes two parameters: col, which is a string value of the current cell in a Google Sheets table, and derla_place, which is the parent XML element to which the gdas:layer element should be added.
The function starts by calling the resolve_keywords_id function (explained later on) to get the ID and label of the keyword corresponding to the value in col. It then creates a new gdas:layer element as a child of derla_place, sets its ref attribute to a URL that includes the keyword ID, and sets its text to the keyword label.

```
def res_prosecution_type_label(col: str) -> str:
    value_dict = {
        "widerstand": "Widerstand",
        "politisch": "Politisch",
        "religiös": "Religiös",
        "individuell": "Individuell",
        "jüdischeopfer": "Jüdische Opfer",
        "gemeindeeinrichtung": "Gemeindeeinrichtung",
        "sammelwohnung": "Sammelwohnungen",
        "kz": "KZ",
        "ikg": "Israelitische Kultusgemeinde",
        "als Jude verfolgt": "Als Jude oder Jüdin verfolgt",
        "todesmarsch": "Todesmarsch",
        "roma": "Roma/Romnija, Sinti/Sintizze, Lovara/Lovarizza",
        "jenische": "Jenische",
        "euthanasie": "NS-Euthanasie",
        "zwangsarbeiter": "ZwangsarbeiterInnen",
        "terror": "NS-Terror",
        "gestapossnsdap": "Gestapo, SS, NSDAP",
        "hinrichtungsstätten": "Hinrichtungsstätte",
        "gefängnis": "Gefängnis",
        "soldaten": "Soldaten",
        "wehrmacht": "Wehrmacht",
        "deserteure": "Deserteure",
        "alliierte": "Alliierte",
        "zivileopfer": "Zivile Opfer",
        "homosexuelle": "Homosexuelle Opfer",
        "kollektiv": "Kollektive Erinnerungszeichen"
    }

    for key in value_dict.keys():
        if col.find(key) != -1:
            col = col.replace(key, value_dict[key])

    return col
```

The function res_prosecution_type_label returns the human readable counterpart of the keys in a given dictionary within the function, which resemble the labels for different groups of victims. The function iterates through the dictionary and returns the label according to the found key.

```
def xml_to_file(out_file: str, root: ET.Element):
    """
    Takes in a filename (+path) and writes given root to the location.
    Pretty prints given xml.
    :out_file path and filename of xml to be written.
    :root ET root element to be outputted as xml.
    """
    xmlstr = minidom.parseString(ET.tostring(root)).toprettyxml(indent="   ")
    with open(out_file, "w", encoding='utf-8') as f:
        f.write(xmlstr)
```

The provided function xml_to_file is used to write an XML document to a file. The function takes two parameters: out_file, which is a string representing the path and filename of the XML file to be written, and root, which is the root element of the XML document.
The function starts by converting the root element to a string using ET.tostring(root). This string is then parsed into an XML document using minidom.parseString. The toprettyxml method is called on the resulting XML document to convert it into a pretty-printed string with an indent of three spaces. This pretty-printed string is assigned to xmlstr.
The function then opens the file specified by out_file in write mode with UTF-8 encoding. It writes xmlstr to the file, effectively writing the pretty-printed XML document to the file.

```
def resolve_keywords_id(key_sheet_val) -> (str, str):
    """
    Dummy method to potentially resolve new keywords
    XML id is
    NOT UED ATM
    :returns Tuple: first id of item. Second string is label of keyword.
    """
    value_dict = {
        "als Jude verfolgt": "Als Jude oder Jüdin verfolgt (Archiv der
Namen)",
        "Altar": "Altar",
        "Arkade": "Arkade",
        "Bild": "Bild",
        "Brunnen": "Brunnen",
        "brücke": "Brücke",
        "denkmal": "Denkmal",
        "euthanasie": "NS-Euthanasie",
        "frauen": "Frauen explizit erwähnt",
        "freimonument": "freistehendes Monument",
        "gasse": "Gasse",
        "gebäude":  "Gebäude",
        "Gedenkkreuz": "Gedenkkreuz",
        "gedenktafel": "Gedenktafel",
        "gedenkstein": "Gedenkstein",
        "gestapo/ss": "Gestapo/SS/NSDAP (Archiv der Namen)",
        "Gipfelkreuz": "Gipfelkreuz",
        "glasfenster": "Glasfenster",
        "glocke": "Glocke",
        "grab": "Grab",
        "grabanlage": "Grabanlage",
        "homosexuelle": "Homosexuelle Opfer",
        "intervention": "Künstlerische Intervention",
        "jenische": "Jenische",
        "jugendliche": "Jugendliche explizit erwähnt",
        "jüdischeopfer, als Jude verfolgt": "Jüdische Opfer - Als Jude oder
Jüdin verfolgt",
        "jüdischeopfer, gemeindeeinrichtung": "Jüdische Opfer -
Gemeindeeinrichtung",
        "jüdischeopfer, ikg": "Jüdische Opfer - Israelitische Kultusgemeinde",
        "jüdischeopfer, kz": "Jüdische Opfer - KZ",
        "jüdischeopfer, sammelwohnung": "Jüdische Opfer - Sammelwohnung",
        "jüdischeopfer, todesmarsch": "Jüdische Opfer - Todesmarsch",
        "jüdischeopfer, zwangsarbeit": "Jüdische Opfer - Zwangsarbeit",
        "Kapelle": "Kapelle",
        "kärntner_slowenen": "Kärntner SlowenInnen",
        "kollektiv": "Kollektiv",
        "kriegerdenkmal": "Kriegerdenkmal",
        "kz_opfer": "KZ Opfer",
        "ns_justiz_opfer": "Opfer der NS Justiz",
        "Orgel": "Orgel",
        "ohnezeichen": "Ort ohne Zeichen",
        "opfer_faschismus": "Opfer des Faschismus",
        "öffentlich_zugänglich": "Öffentlich zugänglich",
        "passage": "Passage",
        "platz": "Platz",
        "roma": "Roma/Romnija und Sinti/Sintize und andere Fahrende wie
Lovara",
        "sammelwohnung": "Sammelwohnung",
        "schriftskulptur": "Schriftskultpur",
        "siedlung": "Siedlung",
        "skulptur": "Skulptur",
        "soldaten, alliierte": "Alliierte Soldaten",
        "soldaten, deserteure": "Soldaten - Deserteure",
        "soldaten, kriegsgefangene": "Soldaten - Kriegsgefangene",
        "soldaten, wehrmacht": "Soldaten - Wehrmacht",
        "statue": "Statue",
        "stolperstein": "Stolperstein",
        "strasse": "Straße",
        "terror, gefängnis": "Orte des Terrors - Gefängnis",
        "terror, gestapossnsdap": "Orte des Terrors - Gestapo/SS/NSDAP",
        "terror, hinrichtungsstätten": "Orte des Terrors -
Hinrichtungsstätten",
        "terror, justiz": "Orte des Terrors - Justiz",
        "terror, kz": "Orte des Terrors - KZ",
        "verkehrsfläche": "Verkehrsfläche",
        "Weg": "Weg",
        "widerstand, individuell": "Individueller Widerstand",
        "widerstand, politisch": "Politischer Widerstand",
        "widerstand, religiös": "Religiöser Widerstand",
        "widerstand, Kärntner PartisanInnenwiderstand": "Kärntner
PartisanInnenwiderstand",
        "zivileopfer": "Zivile Opfer",
        "zwangsarbeiter": "ZwangsarbeiterInnen",
        "Inschrift":"Inschrift"
    }

    lowered_dict =  {k.lower(): v for k, v in value_dict.items()}
    lowerd_keysheet_val = key_sheet_val.lower()

    key_index = 99999999
    keyword_label: str = ""
    try:
        keyword_label = lowered_dict[lowerd_keysheet_val]
        key_index = list(lowered_dict.keys()).index(lowerd_keysheet_val)
    except (ValueError, KeyError, IndexError):
        logging.error(f"Given keyword from gsheet not found in value dict:
{lowerd_keysheet_val}. Assigning dummy value")
        keyword_label = "NOT_FOUND_ERR"

    return (f"keyword{key_index + 1}", keyword_label)
```

The resolve_keywords_id function is used to map a given key to a corresponding ID and label. The function takes one parameter, key_sheet_val, which is the key that should be looked up in the dictionary. The dictionary, value_dict, is defined within the function and maps keys to their corresponding labels.
The function first converts the keys in value_dict to lowercase and assigns the resulting dictionary to lowered_dict. It then converts key_sheet_val to lowercase and assigns the result to lowerd_keysheet_val.
The function initializes a variable, key_index, to a large number and a variable, keyword_label, to an empty string. It then tries to look up lowerd_keysheet_val in

lowered_dict. If the key is found in the dictionary, the corresponding label is assigned to keyword_label and the index of the key in the dictionary is assigned to key_index.
If the key is not found in the dictionary (which would raise a ValueError, KeyError, or IndexError), the function logs an error message indicating that the given keyword was not found in the dictionary and assigns a dummy value to keyword_label.
Finally, the function returns a tuple containing the keyword ID (which is "keyword" followed by key_index + 1) and keyword_label. This function is used to ensure that keywords are in a valid format before they are used in further processing.

```python
def apply_fixed_los_to_tei(pid: str, root: ET.Element):
    """
    Method adds linked lo if available to the TEI.
    - Based on intern array. Checks if pid in intern array
    """

    # dictionary containing pids that have a linked lo assigned
    fixed_los = [
        "o:derla.sty5",
        "o:derla.sty17",
        "o:derla.sty30",
        "o:derla.sty31",
        "o:derla.sty34",
        "o:derla.sty37",
        "o:derla.sty38",
        "o:derla.sty98",
        "o:derla.sty106",
        "o:derla.sty114",
        "o:derla.sty124",
        "o:derla.sty136",
        "o:derla.sty137",
        "o:derla.sty169",
        "o:derla.sty173",
        "o:derla.sty174",
        "o:derla.sty176",
        "o:derla.sty186",
        "o:derla.sty207",
        "o:derla.sty208",
        "o:derla.sty256",
        "o:derla.sty346",
        "o:derla.sty430",
        "o:derla.vor85",
        "o:derla.vor100",
        "o:derla.vor25",
        "o:derla.vor75",
        "o:derla.vor67"
    ]

    # adding xml structure
    if pid in fixed_los:
        location_elem = root.find(".//t:location", {"t":
"http://www.tei-c.org/ns/1.0"})
        mediation_div = ET.SubElement(location_elem, "desc", type="mediation")
        fixed_lo_pid = pid.replace("o:derla.", "o:derla.fix")
        ET.SubElement(mediation_div, "ref",
target=f"https://gams.uni-graz.at/{fixed_lo_pid}")
        logging.info(f"Added fixed LO for pid: {pid}")
```

The function apply_fixed_los_to_tei is used to add a linked location (LO) to a TEI XML document if the Persistent Identifier of the document is in a predefined list. The function takes two parameters: pid, which is the PID of the document, and root, which is the root element of the XML document.
The function starts by defining a list, fixed_los, which contains the PIDs that have a linked LO assigned. These PIDs are hardcoded into the function.
The function then checks if pid is in fixed_los. If pid is in the list, the function proceeds to add the linked LO to the TEI.
The function first finds the location element in the XML document. It then creates a new desc element with a type attribute of "mediation" as a child of the location element. This desc element is assigned to mediation_div.
The function then replaces "o:derla." in pid with "o:derla.fix" to create the PID of the fixed LO and assigns this to fixed_lo_pid.
The function then creates a new ref element as a child of mediation_div, sets its target attribute to a URL that includes fixed_lo_pid, and adds the ref element to the mediation_div element.
Finally, the function logs a message to indicate that a fixed LO has been added for the PID.

data_processing/main.py

```
import gSheetHandler as g_handler
import erla_xml_builder as erla_xml_builder
from env import *
import logging
import os

def main():
      # at file import create log folder if not exist
      try:
      folder_path_name = "." + os.path.sep + "log"
      os.mkdir("./log")
      except FileExistsError:
```

```
      pass
      # set logging folder and level

logging.basicConfig(filename='log/erla_xml_builder.log',level=logging.DE
BUG, filemode="w")
      logging.debug("Applying log file -- main.py called. Starting
program")

      # at last start actual process
      # (data processing for DERLA-project)
      create_ingest_material()
```

The main() function is the entry point of the script. It starts by creating a directory named "log" if it doesn't already exist. This is done using the os.mkdir() function, and any FileExistsError exceptions (which occur if the directory already exists) are silently ignored. After that, it sets up a logging system using the logging module, specifying a log file and setting the logging level to DEBUG. This means that all messages of level DEBUG and above will be logged. The first log message indicates that the program has started. Finally, it calls the create_ingest_material() function to start the actual data processing.

```
def create_ingest_material():
  #####
  # Data generation for styria
  # g_data: [[str]] =
g_handler.get_data(sheet_id="1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPL_z83AyjjRM")
  # # g_handler.back_up_data(g_data, "./backups/steiermark/styr")

  # # Getting the person data for Styria
  # persons_sty: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d/e/2PA
CX-1vTRFAMnUQ7YC9HZwwSrw55tXhopXcXrGaVl2CMGd8JMRMYiZn7lmDG-pCU2SmDSuIR0-6vzgbT
D29lS/pub?gid=1356804042&single=true&output=csv")
  # # print(persons_sty)

  # # build ingest files
  # styr_pid_base: str = f"o:{PROJECT_ABBR}.sty" # only 3 letters allowed
  # erla_xml_builder.build_ingest_files(g_data,
pid_name=styr_pid_base,person_table=persons_sty, build_max_rows = 541)
  # erla_xml_builder.build_ingest_gml_files(g_data, pid_name=styr_pid_base,
build_max_rows = 1000)

  # #####
  # # Data generation for vorarlberg
  # g_data: [[str]] =
g_handler.get_data(sheet_id="1hFJprVGgTUOBlHoW8RdzmYTGDzek_DebpOutQRdeZ_k")
  # # g_handler.back_up_data(g_data, "./backups/vorarlberg/vor")

  # # Getting the person data for Vorarlberg
  # persons_vor: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d/e/2PA
CX-1vRfam42JhbxFMs8u84im8twvZS5zqVqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYO
xPBe0/pub?gid=2079639611&single=true&output=csv")
  # # print(persons_vor)

  # # build ingest files
  # vor_pid_base: str = f"o:{PROJECT_ABBR}.vor"
  # erla_xml_builder.build_ingest_files(g_data, pid_name=vor_pid_base,
person_table=persons_vor, build_max_rows = 150)

  # # print("Vorarlberg: Temporarily skipping build of gml files -> need to
adapt table model processing!")
  # erla_xml_builder.build_ingest_gml_files(g_data, pid_name=vor_pid_base,
build_max_rows = 1000)


  ####
  # Data generation Tyrol
  g_data: [[str]] =
g_handler.get_data(sheet_id="1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc")
  # g_handler.back_up_data(g_data, "./backups/tirol/tir")

  # Getting the person data for Tyrol
  persons_tir: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d/1bI2D
Ch9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc/export?format=csv&gid=1986675455")

  # build ingest files
  tir_pid_base: str = f"o:{PROJECT_ABBR}.tir"
  erla_xml_builder.build_ingest_files(g_data, pid_name=tir_pid_base,
person_table=persons_tir, build_max_rows = 234)
  erla_xml_builder.build_ingest_gml_files(g_data, pid_name=tir_pid_base,
build_max_rows = 1000)



  ####
  # Data generation Carinthia
  g_data: [[str]] =
g_handler.get_data(sheet_id="13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0")
  # g_handler.back_up_data(g_data, "./backups/tirol/tir")

  # Getting the person data for Carinthia
  persons_car: [[str]] =
g_handler.get_sheetUrl(sheet_url="https://docs.google.com/spreadsheets/d/13W9I
sf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0/export?format=csv&gid=1026060477")

  # build ingest files
  car_pid_base: str = f"o:{PROJECT_ABBR}.car"
  erla_xml_builder.build_ingest_files(g_data, pid_name=car_pid_base,
person_table=persons_car, build_max_rows = 231)
  erla_xml_builder.build_ingest_gml_files(g_data, pid_name=car_pid_base,
build_max_rows = 1000)



if __name__ == '__main__':
    main()
```

This function is responsible for generating and ingesting data for different regions (Styria, Vorarlberg, Tyrol, Carinthia). For each region, it follows a similar process:

1. It retrieves data from a Google Sheets document using the get_data() method of an object g_handler. The specific document is identified by a sheet ID.
2. It retrieves person data from another Google Sheets document using the get_sheetUrl() method of g_handler. The specific document is identified by an URL.
3. It builds ingest files using the build_ingest_files() and build_ingest_gml_files() methods of erla_xml_builder. These methods take several parameters, including the retrieved data, a base PID name, the person data, and a maximum number of rows to build.

The code for Styria and Vorarlberg is commented out, so it will not be executed. Only the code for Tyrol and Carinthia will run. This needs to be adapted according to the desired outcome. Additionally, adding new states requires adapting the code to include them.
The get_data() method is used to fetch data from a Google Sheets document. The ID of the document is passed as an argument to this method. The data fetched is stored in the g_data variable.
The script is then executed.

xml_build/archive_of_names/__main__.py

```
# Author: Sebastian
#
from DERLAPersListHandler import DERLAPersListHandler
```

```python
from DERLAKeyWordsHandler import DERLAKeyWordsHandler
from DERLATeiLoHandler import DERLATeiLoHandler
from DERLALoListHandler import DERLALoListHandler
from DERLADidacticGlossary import DERLADidacticsGlossary


def build_fixed_lo_xmls(env):
    """
    Method handles the building of educational related xml files.
    Individual TEI XML objects per educational material.
    """

    # sty fixed
    # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z83
AyjjRM/pub?gid=1790748251&single=true&output=csv"
    # env["pid_var"] = "sty"
    # derla_vor_lo_builder = DERLATeiLoHandler(env)

    # vor fixed (created by hand - skipped)

    # tir fixed
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc&gid=2
63259825"
    env["pid_var"] = "tir"
    derla_vor_lo_builder = DERLATeiLoHandler(env)

    # car fixed
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0&gid=7
40894826"
    env["pid_var"] = "car"
    derla_vor_lo_builder = DERLATeiLoHandler(env)
```
This function is responsible for building Fixed Mediation Offer XML files. The files are built based on data fetched from various Google Sheets documents.
The function takes one argument, env, which is a dictionary. This dictionary is used to store parameters that are needed to fetch data from Google Sheets and to create XML files.

The section of the function handling Styrian data is commented out. This should be adapted according to the desired state specific data output. Additional states also need to be added here. Each section sets two parameters in the env dictionary: get_param and pid_var.
The get_param parameter is an URL that points to a Google Sheets document. The URL is structured to directly export the data in the document as a CSV file.
The pid_var parameter is a string that identifies the state's DERLA specific abbreviation.
After setting the parameters, the function creates an instance of DERLATeiLoHandler, passing the env dictionary to the constructor. The instance is stored in the variable derla_vor_lo_builder.

```python
def build_lo_list(env):
    """
    Method handles construction of los as lists
    (for the fixierte vermittlungsangebote).
    """

    # for styria
    # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z83
AyjjRM/pub?gid=1790748251&single=true&output=csv"
    # env["pid_var"] = "sty"
    # DERLALoListHandler(env)

    # vor list
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc&gid=2
63259825"
    env["pid_var"] = "tir"
    DERLALoListHandler(env)

    # tir list
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0&gid=7
40894826"
    env["pid_var"] = "car"
    DERLALoListHandler(env)
```

This function is very similar to the one above, as both handle Fixed Mediation Offers. After setting the parameters, the function creates an instance of DERLALoListHandler, passing the env dictionary to the constructor. DERLALoListHandler is a class that handles the actual fetching of data and building of lists.

The DERLALoListHandler class is a subclass of GAMSSOURCEBuilder and CSVHarvester. It fetches data from Google Sheets, backs up the data, applies the data to an XML structure, and writes the XML to a file. The class also includes several methods for handling specific columns in the fetched data.

```python
def build_perslists(env):
    """
    Method to handle construction of perslist xmls needed in DERLA.
    """

    # vor persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zq
VqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639611&single=t
rue&output=csv"
    env["pid_var"] = "vor"
    DERLAPersListHandler(env)

    # sty persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vTRFAMnUQ7YC9HZwwSrw55tXhopXc
XrGaVl2CMGd8JMRMYiZn7lmDG-pCU2SmDSuIR0-6vzgbTD29lS/pub?gid=1356804042&single=t
rue&output=csv"
    env["pid_var"] = "sty"
    DERLAPersListHandler(env)

    # tir persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&gid=1986675455"
    env["pid_var"] = "tir"
    DERLAPersListHandler(env)

    # car persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&gid=1026060477"
    env["pid_var"] = "car"
    DERLAPersListHandler(env)
```

This function is very straightforward and handles the construction of all person lists for all states.

```python
def build_keywords_tei(env):
    """
    Handles building of keywords TEI.
    """
    # needed env setup
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1j3GeeoIwC72tlSToz8yrEgKo845gQ_yVMDymc
VhvNus/pub?gid=0&single=true&output=csv"
    env["pid_var"] = ""
    DERLAKeyWordsHandler(env)
```

This function handles the construction of the keywords file, which is not state-specific. It calls the class DERLAKeyWordsHandler.

```
def build_didactical_glossary(env):
    """
    Handles building of the didactical glossary for DERLA
    """

    DERLADidacticsGlossary(env)
```

This function handles the construction of the keywords file, which is not state-specific. It calls the class DERLADidacticGlossary.

```
if __name__ == "__main__":

    # Define env variables here as dictionary
    # with basic setting
    env = {
        "log_level": "DEBUG",
        "origin": "https://glossa.uni-graz.at",
        "pid_static": "o:derla.",
        "pid_var": "vor",
        "get_param":
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zq
VqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639611&single=t
rue&output=csv"
    }

    # # instantiate your custom class here
    # derla_builder = DERLAPersListHandler(env)
    # build_perslists(env)

    # keywords tei
    # build_keywords_tei(env)
    # perslists
    # build_perslists(env)

    # build educational xml files
    # build_fixed_lo_xmls(env)

    #####
    # LO lists
    # build_lo_list(env)

    ## didactical glossary
    build_didactical_glossary(env)
```

gSheetHandler.py

```
import requests
import csv
import io
from datetime import date


def get_data(sheet_id: str) -> [[str]]:
    """
    Gets-requests the data from a published google spreadsheet.
    :param sheet_id: id of the published google spread sheet.
    :return: List of given table. With a List for each row. Nested
    rows/columns are not recognized.
    """
    sheet_url =
"https://docs.google.com/spreadsheets/d/{0}/export?format=csv".format(sheet_id
)

    r = requests.get(sheet_url)
    r.encoding = 'utf-8'    # Needed to set correct encoding!

    sio = io.StringIO( r.text, newline=None)
    reader = csv.reader(sio)

    g_sheet_data: [[str]] = []
    for row in reader:
        g_sheet_data.append(row)
```

```python
    return g_sheet_data


def get_sheetUrl(sheet_url: str) -> [[str]]:
    """
    Gets-requests the data from a published google spreadsheet.
    :param sheet_url: url to the csv of published google spread sheet (at
least read access).
    e.g.
https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zqV
qpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639611&single=tr
ue&output=csv
    :return: List of given table. With a List for each row. Nested
    rows/columns are not recognized.
    """
    r = requests.get(sheet_url)
    r.encoding = 'utf-8'    # Needed to set correct encoding!

    sio = io.StringIO( r.text, newline=None)
    reader = csv.reader(sio)

    g_sheet_data: [[str]] = []
    for row in reader:
        g_sheet_data.append(row)

    return g_sheet_data


def back_up_data(csv_list:[[str]], filename_path = "myBackup"):
    """
    Takes in the sheet data and generates a backup file. Adds .csv and current
date
    to the filename.
    :param csv_list: List containing each individual row as list.
    :param filename_path: name of the csv - file (date and ".csv" will be
automatically appended.)
    :return:
    """
    today = date.today()
    d1 = today.strftime("%b-%d-%Y")
    filename_path = filename_path + "_" + d1 + ".csv"
    with open(filename_path, 'w', newline='', encoding='utf-8') as myfile:
        wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
        for row in csv_list:
            wr.writerow(row)
```

xml_operations.py

```
import xml.etree.ElementTree as ET


def clean_xml_string(xml: str) -> str:
    """
    Replaces double whitespace, \n \t and whitespace before and after "<" or
"/>"
    :param xml: xml as string
    :return: cleaned xml.
    """
    # xml = xml.replace("  ", "")
    xml = xml.replace("\n", " ")
    xml = xml.replace("\t", " ")
    xml = xml.replace("  ", " ")
    xml = xml.replace("   ", " ")
    xml = xml.replace(" <", "<")
    xml = xml.replace("< ", "<")
    xml = xml.replace(" />", "/>")
    xml = xml.replace("> ", ">")
    return xml


def parse_xml(xml: str, default_namespaces: {} = None) -> ET.Element:
    """
    Parses given string as xml. Needs namespace declarations. For the default
namespace
    just ommit an empty string as key.
    :param xml: xml as string.
    :param default_namespaces: Dictionary of namespaces. Key is the 'shorctut'
and value the full resource-link.
    :return:
    """
    # Register default tei namespace first.
    if default_namespaces:
        for key in default_namespaces:
            ET.register_namespace(key, default_namespaces[key])

    tei_string: str = clean_xml_string(xml)
    root: ET.Element = ET.fromstring(tei_string)
    return root
```

skos_to_xml.py

```
import logging
import gSheetHandler as g_handler
import os
from typing import AnyStr, Callable
from env import SKOS_NAMESPACES

# xml related libs
import erla_templates as erla_templates
import xml.etree.ElementTree as ET
import xml_operations as xml_ops

# write file
import pathlib


def build_skos_xml(csv: [[str]]):
  """
  Main routine to build the SKOS - xml file out
  of the parsed gsheet.

  Args:
    csv: A list of lists representing the CSV data.

  Returns:
    None
  """
  logging.debug("Starting to build the SKOS xml file via build_skos_xml")

  # first parse xml template
  root = xml_ops.parse_xml(erla_templates.OBJECT_SKOS, SKOS_NAMESPACES)

  # loop csv row
  for row_index, row in enumerate(csv):
    # first two rows are just examples.
    if row_index > 1:
      skos_concept_tag = ET.SubElement(root, "skos:Concept",
{'rdf:about':f"https://gams.uni-graz.at/archive/objects/o:derla.terms#term{row
_index + 1}"})
      ET.SubElement(skos_concept_tag, "skos:broader",
{"rdf:resource":"https://gams.uni-graz.at/archive/objects/o:derla.terms#didact
icalTerm"})
      ET.SubElement(skos_concept_tag, "skos:inScheme",
{"rdf:resource":"https://gams.uni-graz.at/archive/objects/o:derla.terms"})
      # loop through individual columns
      for col_index, col_val in enumerate(row):
        if col_val == "" and col_index != 2: # col 2 is intern comment
          logging.warn("Got a column without a value. row index: %s, column
index: %s", row_index, col_index)

        # lemma
        if col_index == 0:
          ET.SubElement(skos_concept_tag, "skos:prefLabel",
{"xml:lang":"de"}).text = col_val

        # description
        if col_index == 1:
          ET.SubElement(skos_concept_tag, "skos:definition",
{"xml:lang":"de"}).text = col_val



  # writing xml files in correct folders
  results_path = str(os.getcwd()) + "/results/"
  object_folder_path = results_path
  pathlib.Path(object_folder_path).mkdir(parents=True, exist_ok=True)
  tree = ET.ElementTree(root)

  skos_path = "results/derla_terms.xml"
  tree.write(skos_path, encoding='UTF-8', xml_declaration=True)
  logging.info("Wrote SKOS to the filesystem at path: %s ", skos_path)


def main():
    # at file import create log folder if not exist
    try:
```

```python
      folder_path_name = "." + os.path.sep + "log"
      os.mkdir("./log")
    except FileExistsError:
      pass
    # set logging folder and level

logging.basicConfig(filename='log/derla_skos_to_xml.log',level=logging.INFO,
filemode="w")
    logging.debug("Applying log file -- main.py called. Starting program to
build the SKOS xml")



    # load and backup gsheet datas
    g_data: [[str]] =
g_handler.get_data(sheet_id="1KDjZOUhNX2s7hrcYmCMes6Yo2ENUlPsPplmizP8al0A")
    g_handler.back_up_data(g_data, "./backups/skos_didactics")

    # build the xml data here
    build_skos_xml(g_data)




if __name__ == '__main__':
    main()
```

xml_operations.py

```python
import xml.etree.ElementTree as ET



def clean_xml_string(xml: str) -> str:
    """
    Replaces double whitespace, \n \t and whitespace before and after "<" or
"/>"
    :param xml: xml as string
    :return: cleaned xml.
    """
    # xml = xml.replace("  ", "")
    xml = xml.replace("\n", " ")
    xml = xml.replace("\t", " ")
    xml = xml.replace("  ", " ")
    xml = xml.replace("   ", " ")
    xml = xml.replace(" <", "<")
    xml = xml.replace("< ", "<")
    xml = xml.replace(" />", "/>")
    xml = xml.replace("> ", ">")
    return xml


def parse_xml(xml: str, default_namespaces: {} = None) -> ET.Element:
    """
    Parses given string as xml. Needs namespace declarations. For the default
namespace
    just ommit an empty string as key.
    :param xml: xml as string.
    :param default_namespaces: Dictionary of namespaces. Key is the 'shorctut'
and value the full resource-link.
    The root element of the parsed XML.
    """
    # Register default tei namespace first.
    if default_namespaces:
        for key in default_namespaces:
            ET.register_namespace(key, default_namespaces[key])

    tei_string: str = clean_xml_string(xml)
    root: ET.Element = ET.fromstring(tei_string)
    return root
```

xml_build/archive_of_names/DERLADidacticGlossary.py

```python
from typing import Dict, List, Tuple, Any, Optional
import xml.etree.ElementTree as ET
from cm.builder.GAMSSourceBuilder import GAMSSOURCEBuilder
from harvest.source_types.CSVHarvester import CSVHarvester
from cm.source_templates.TEI.KeyWords import TEI_KEY_WORDS
from cm.SourceDatastream import SourceDatastream
from cm.static_namespaces import TEI_NAMESPACES
from typing import Tuple
import os
import harvest.gSheetHandler



class DERLADidacticsGlossary(GAMSSOURCEBuilder, CSVHarvester):
    def __init__(self, env):
        """
        Class to handle building DERLA's Didactics glossary list
      Args: env (str): The path to the environment file.

        """

        # 01. Initialization and .env
        # pass env file
        GAMSSOURCEBuilder.__init__(
            self, TEI_KEY_WORDS, TEI_NAMESPACES, env)
        CSVHarvester.__init__(self, build_max_rows=1000)

        # 02. Harvest data
        # 02a. get data from gsheets

#https://docs.google.com/spreadsheets/d/1KDjZOUhNX2s7hrcYmCMes6Yo2ENUlPsPplmiz
P8al0A/edit?usp=sharing

self.get("https://docs.google.com/spreadsheets/d/1KDjZOUhNX2s7hrcYmCMes6Yo2ENU
lPsPplmizP8al0A/pub?gid=0&single=true&output=csv")
        # 02b. backup
        self.backup(os.getcwd() + os.sep + "backup" + os.sep + "g_sheet_")

        # 03. Build data
        self.apply_data()

        # 04. Post processing
        self.write_source_xml()

    def apply_data(self):
        """
        Method add data to xml

        This method adds data to the XML file. It performs the following
steps:
        1. Builds the source-datastream and defines pid globals.
        2. Constructs the pid.
        3. Assigns the pid to the publication statement element in the XML.
        4. Assigns the list element to the keywords_list variable.
        5. Assigns titles to the XML.
        6. Handles row elements and label/text elements.
        7. Loops through the data.
        Parameters:
        - None

        Returns:
        - None

        """
        # 01. first build source-datastream + define pid globals
        # (will expose many needed class fields)
        # 01b. construct pid
        self.pid = self.pid_static + "didgloss" # o:derla.pidgloss
        sdstream = self.build_dsource_stream(self.pid)

        # 02. Global xml data assignment (like pid etc.)
        pubstmt_elem = sdstream.root.find(
            ".//publicationStmt", self.xml_namespaces)
        ET.SubElement(pubstmt_elem, "idno", type="PID").text = self.pid
        keywords_list = sdstream.root.find(
            ".//list", self.xml_namespaces)
        self.data_refs["list"] = keywords_list

        # Assign title (needed for the didadics glossary)
        sdstream.root.findall(".//title", self.xml_namespaces)[0].text =
"Didaktisches Glossar"
        sdstream.root.findall(".//title", self.xml_namespaces)[1].text =
"didactical glossary"
        sdstream.root.findall(".//p", self.xml_namespaces)[0].text =
"Vermittlung: Didaktisches Glossar, born digital"

        row_funcs = [self._handle_row_elems]
        col_funcs = [self._handle_label, self._handle_text]

        self.loop_data(row_funcs, col_funcs, skip_rows=2)


    def _handle_row_elems(self, row_ind: int, row: List[str], data_refs:
Dict[str, Any]):
      """
      Adds item per item with and assigns xml:ids.
      Calls list_keywords to add <ptr> references where keywords were used.
        Args:
            row_ind (int): The index of the row.
            row (List[str]): The list of strings representing the row.
            data_refs (Dict[str, Any]): A dictionary containing references to
data elements.

        Returns:
            cur_item (Element): The newly created XML element representing the
current item.

      """
      cur_item = ET.SubElement(data_refs["list"], "item", {"xml:id":
f"gloss{row_ind + 1}"})
      data_refs["cur_item"] = cur_item

      return cur_item


    def _handle_label(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Adds lable of current keyword
Args:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): A dictionary containing data
references.

        Returns:
            None

      """
      ET.SubElement(data_refs["cur_item"], "term").text = col_val

    def _handle_text(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Adds the text of the glossary
Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing references to
data.
```

```
        Returns:
            None

      """
      ET.SubElement(data_refs["cur_item"], "p").text = col_val
```

xml_build/archive_of_names/DERLAKeyWordsHandler.py

```
from typing import Dict, List, Tuple, Any, Optional
import xml.etree.ElementTree as ET
from cm.builder.GAMSSourceBuilder import GAMSSOURCEBuilder
from harvest.source_types.CSVHarvester import CSVHarvester
from cm.source_templates.TEI.KeyWords import TEI_KEY_WORDS
from cm.SourceDatastream import SourceDatastream
from cm.static_namespaces import TEI_NAMESPACES
from typing import Tuple
import os
import harvest.gSheetHandler


class DERLAKeyWordsHandler(GAMSSOURCEBuilder, CSVHarvester):
    def __init__(self, env):
        """
        Initializes an instance of DERLAKeyWordsHandler.

        Parameters:
        - env: The environment variable dictionary.

        Attributes:
        - persons_vor: A list of lists containing data from the "vor" sheet in
Google Sheets.
        - persons_sty: A list of lists containing data from the "sty" sheet in
Google Sheets.
        - pid: The PID (Persistent Identifier) for the keywords.
        - data_refs: A dictionary containing references to already generated
XML elements.

        """
        # TODO need also to ask for sty data! (Move pos + maybe get data from
outside (better for sure!)
        self.persons_vor: List[List[str]] =
harvest.gSheetHandler.get_data(sheet_id="1hFJprVGgTUOBlHoW8RdzmYTGDzek_DebpOut
QRdeZ_k")
        self.persons_sty: List[List[str]] =
harvest.gSheetHandler.get_data(sheet_id="1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z8
3AyjjRM")

        # 01. Initialization and .env
        # pass env file
        GAMSSOURCEBuilder.__init__(
            self, TEI_KEY_WORDS, TEI_NAMESPACES, env)
        CSVHarvester.__init__(self, build_max_rows=100)

        # 02. Harvest data
        # 02a. get data from gsheets
        self.get(self._env["get_param"])
        # 02b. backup
        self.backup(os.getcwd() + os.sep + "backup" + os.sep + "g_sheet_")

        # 03. Build data
        self.apply_data()

        # 04. Post processing
        self.write_source_xml()

    def apply_data(self):
        """
        Method add data to xml
This method adds data to the XML file. It performs the following steps:
            1. Builds the source-datastream and defines pid globals.
            2. Constructs the pid.
            3. Assigns global XML data such as pid.
            4. Handles row elements and column functions.

        """
        # 01. first build source-datastream + define pid globals
        # (will expose many needed class fields)
        # 01b. construct pid
        self.pid = self.pid_static + "keywords"
        sdstream = self.build_dsource_stream(self.pid)
        # 02. Global xml data assignment (like pid etc.)
        pubstmt_elem = sdstream.root.find(
            ".//publicationStmt", self.xml_namespaces)
        ET.SubElement(pubstmt_elem, "idno", type="PID").text = self.pid
        keywords_list = sdstream.root.find(
            ".//list", self.xml_namespaces)
        self.data_refs["list"] = keywords_list

        # title not dynamic

        row_funcs = [self._handle_row_elems]
        col_funcs = [self._handle_id, self._handle_label, self._handle_desc]

        self.loop_data(row_funcs, col_funcs)


    def _handle_row_elems(self, row_ind: int, row: List[str], data_refs:
Dict[str, Any]):
      """
      Adds item per item with and assigns xml:ids.
      Calls list_keywords to add <ptr> references where keywords were used.
       Args:
            row_ind (int): The index of the row.
            row (List[str]): The list of values in the row.
            data_refs (Dict[str, Any]): A dictionary containing references to
XML elements.

        Returns:
            cur_item (Element): The newly created XML element.

      """
      cur_item = ET.SubElement(data_refs["list"], "item", {"xml:id":
f"keyword{row_ind + 1}"})
      data_refs["cur_item"] = cur_item

      # Adding a linkGrp with ptr to individual places where keywords were
used.
      cur_linkgrp = ET.SubElement(cur_item, "linkGrp")
      data_refs["link_grp"] = cur_linkgrp
      # calling method for different states
      self.list_keyword(row[0], data_refs, self.persons_vor, "vor")
      self.list_keyword(row[0], data_refs, self.persons_sty, "sty")

      return cur_item


    def _handle_id(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Dummy method for first col in keywords table
      Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary of data references.

        Returns:
            None

      """
      pass

    def _handle_label(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
        Handles the label by adding it as a term element to the current item.

        Args:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): A dictionary containing data
references.

        Returns:
            None
        """

      ET.SubElement(data_refs["cur_item"], "term").text = col_val

    def _handle_desc(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Creates a desc element with col as value.
      Args:
              col_ind (int): The index of the column.
              col_val (str): The value of the column.
              data_refs (Dict[str, Any]): A dictionary containing references
to data.

          Returns:
              None

      """
      if col_val != "" and col_val != "-" and len(col_val) > 1:
        ET.SubElement(data_refs["cur_item"], "desc").text = col_val

    def list_keyword(self, sheet_kword: str, data_refs: Dict[str, Any],
pers_table: List[List[str]], pid_abbr: str):
        """
        Looks up assigned controlled values in places tables. If found applies
link to generated keyword in xml.
        :param sheet_kword: controlled keyword used in gsheet in keywords
table
        :type sheet_kword: str
        :param data_refs: references to already generated xml elements
        :type data_refs: Dict[str, Any]
        :param pers_table: cur person table as enrichment source
        :type pers_table: List[List[str]]
        :param pid_abbr: abbreviation for person ID
        :type pid_abbr: str

        """

        for place_row_ind, place_row in enumerate(pers_table):
            kword_ind = 0
            if place_row_ind < 2: continue
            try:
                #print(pers_row)
                kword_ind = place_row.index(sheet_kword)
                kword_id, kword_label = self.resolve_keywords_id(sheet_kword)

                ET.SubElement(data_refs["link_grp"], "ptr", rend=place_row[0],
target=f"{self.origin}/{self.pid_static}{pid_abbr}{place_row_ind + 1}")
            except ValueError:
                pass
            # handling vor ja / nein values
            if sheet_kword == "öffentlich_zugänglich":
                if place_row[8] == "ja":
                    ET.SubElement(data_refs["link_grp"], "ptr",
rend=place_row[0],
target=f"{self.origin}/{self.pid_static}{pid_abbr}{place_row_ind + 1}")

            if sheet_kword == "frauen":
                if place_row[15] == "ja":
                    ET.SubElement(data_refs["link_grp"], "ptr",
rend=place_row[0],
target=f"{self.origin}/{self.pid_static}{pid_abbr}{place_row_ind + 1}")

            if sheet_kword == "jugendliche":
                if place_row[16] == "ja":
                    ET.SubElement(data_refs["link_grp"], "ptr",
rend=place_row[0], target=f"{self.origin}/{self.pid_static}sty{place_row_ind +
1}")


    def resolve_keywords_id(self, key_sheet_val) -> Tuple[str, str]:
        """
        Dummy method to potentially resolve new keywords
        XML id is
        NOT UED ATM
        This method takes a keyword value from a Google Sheet and attempts to
resolve its corresponding XML id and label.

            :param key_sheet_val: The value of the keyword from the Google
Sheet.
            :type key_sheet_val: str
            :returns: A tuple containing the XML id and label of the keyword.
            :rtype: Tuple[str, str]

        """
        value_dict = {
            "als Jude verfolgt": "Als Jude oder Jüdin verfolgt (Archiv der
Namen)",
            "Altar": "Altar",
            "Arkade": "Arkade",
            "Bild": "Bild",
            "Brunnen": "Brunnen",
            "brücke": "Brücke",
            "denkmal": "Denkmal",
            "euthanasie": "NS-Euthanasie",
            "frauen": "Frauen explizit erwähnt",
            "freimonument": "freistehendes Monument",
            "gasse": "Gasse",
            "gebäude":  "Gebäude",
            "Gedenkkreuz": "Gedenkkreuz",
            "gedenktafel": "Gedenktafel",
            "gedenkstein": "Gedenkstein",
            "gestapo/ss": "Gestapo/SS/NSDAP (Archiv der Namen)",
            "Gipfelkreuz": "Gipfelkreuz",
            "glasfenster": "Glasfenster",
            "glocke": "Glocke",
            "grab": "Grab",
            "grabanlage": "Grabanlage",
            "homosexuelle": "Homosexuelle Opfer",
            "intervention": "Künstlerische Intervention",
            "jenische": "Jenische",
            "jugendliche": "Jugendliche explizit erwähnt",
            "jüdischeopfer, als Jude verfolgt": "Jüdische Opfer - Als Jude
oder Jüdin verfolgt",
            "jüdischeopfer, gemeindeeinrichtung": "Jüdische Opfer -
Gemeindeeinrichtung",
            "jüdischeopfer, ikg": "Jüdische Opfer - Israelitische
Kultusgemeinde",
            "jüdischeopfer, kz": "Jüdische Opfer - KZ",
            "jüdischeopfer, sammelwohnung": "Jüdische Opfer - Sammelwohnung",
            "jüdischeopfer, todesmarsch": "Jüdische Opfer - Todesmarsch",
            "jüdischeopfer, zwangsarbeit": "Jüdische Opfer - Zwangsarbeit",
            "Kapelle": "Kapelle",
            "kärntner_slowenen": "Kärntner Slowenen",
            "kollektiv": "Kollektiv",
            "kriegerdenkmal": "Kriegerdenkmal",
            "kz_opfer": "KZ Opfer",
            "ns_justiz_opfer": "Opfer der NS Justiz",
            "Orgel": "Orgel",
            "ohnezeichen": "Ort ohne Zeichen",
            "opfer_faschismus": "Opfer des Faschismus",
            "öffentlich_zugänglich": "Öffentlich zugänglich",
            "passage": "Passage",
            "platz": "Platz",
            "roma": "Roma/Romnija und Sinti/Sintize und andere Fahrende wie
Lovara",
            "sammelwohnung": "Sammelwohnung",
            "schriftskulptur": "Schriftskultpur",
            "siedlung": "Siedlung",
            "skulptur": "Skulptur",
            "soldaten, alliierte": "Alliierte Soldaten",
            "soldaten, deserteure": "Soldaten - Deserteure",
            "soldaten, kriegsgefangene": "Soldaten - Kriegsgefangene",
            "soldaten, wehrmacht": "Soldaten - Wehrmacht",
            "statue": "Statue",
            "stolperstein": "Stolperstein",
            "strasse": "Straße",
            "terror, gefängnis": "Orte des Terrors - Gefängnis",
            "terror, gestapossnsdap": "Orte des Terrors - Gestapo/SS/NSDAP",
            "terror, hinrichtungsstätten": "Orte des Terrors -
Hinrichtungsstätten",
            "terror, justiz": "Orte des Terrors - Justiz",
            "terror, kz": "Orte des Terrors - KZ",
            "verkehrsfläche": "Verkehrsfläche",
            "Weg": "Weg",
            "widerstand, individuell": "Individueller Widerstand",
            "widerstand, politisch": "Politischer Widerstand",
            "widerstand, religiös": "Religiöser Widerstand",
            "widerstand, Kärntner PartisanInnenwiderstand": "Kärntner
PartisanInnenwiderstand",
            "zivileopfer": "Zivile Opfer",
            "zwangsarbeiter": "ZwangsarbeiterInnen",
            "Inschrift":"Inschrift"
        }

        lowered_dict =  {k.lower(): v for k, v in value_dict.items()}
        lowerd_keysheet_val = key_sheet_val.lower()

        key_index = 99999999
        keyword_label: str = ""
        try:
            keyword_label = lowered_dict[lowerd_keysheet_val]
            key_index = list(lowered_dict.keys()).index(lowerd_keysheet_val)
        except (ValueError, KeyError, IndexError):
            self.logger.error(f"Given keyword from gsheet not found in value
dict: {key_sheet_val}. Assigning dummy value.")
```

```
            keyword_label = "NOT_FOUND_ERR"

        return (f"keyword{key_index + 1}", keyword_label)
```

xml_build/archive_of_names/DERLALoListHandler.py

```
from typing import Dict, List, Tuple, Any, Optional
import xml.etree.ElementTree as ET
from cm.builder.GAMSSourceBuilder import GAMSSOURCEBuilder
from harvest.source_types.CSVHarvester import CSVHarvester
from cm.source_templates.TEI.KeyWords import TEI_KEY_WORDS
from cm.SourceDatastream import SourceDatastream
from cm.static_namespaces import TEI_NAMESPACES
from typing import Tuple
import os
import harvest.gSheetHandler


class DERLALoListHandler(GAMSSOURCEBuilder, CSVHarvester):
    def __init__(self, env):
        """
        Class to handle building DERLA's Didactics glossary list
      Args:
      env (str): The environment file path.

        """

        # 01. Initialization and .env
        # pass env file
        GAMSSOURCEBuilder.__init__(
            self, TEI_KEY_WORDS, TEI_NAMESPACES, env)
        CSVHarvester.__init__(self, build_max_rows=1000)

        # 02. Harvest data
        # 02a. get data from gsheets

#https://docs.google.com/spreadsheets/d/1KDjZOUhNX2s7hrcYmCMes6Yo2ENUlPsPplmiz
P8al0A/edit?usp=sharing
        self.get(self._env["get_param"])
        # 02b. backup
        self.backup(os.getcwd() + os.sep + "backup" + os.sep + "g_sheet_")
```

```python
        # 03. Build data
        self.apply_data()

        # 04. Post processing
        self.write_source_xml()

    def apply_data(self):
        """
        Method add data to xml
This method adds data to an XML document. It performs the following steps:
      1. Builds a source-datastream and defines pid globals.
      2. Constructs a pid.
      3. Assigns global XML data such as pid.
      4. Adds a region to the tei document based on the pid_var value.
      5. Assigns a title to the XML document.
      6. Assigns a didactics context to the XML document.
      7. Defines row and column functions for processing data.
      8. Loops through the data and applies the row and column functions.

        """
        # 01. first build source-datastream + define pid globals
        # (will expose many needed class fields)
        # 01b. construct pid
        self.pid = self.pid_static + "fix" + self.pid_var # o:derla.fixsty
        sdstream = self.build_dsource_stream(self.pid)

        # 02. Global xml data assignment (like pid etc.)
        pubstmt_elem = sdstream.root.find(
            ".//publicationStmt", self.xml_namespaces)
        ET.SubElement(pubstmt_elem, "idno", type="PID").text = self.pid
        lo_list = sdstream.root.find(
            ".//list", self.xml_namespaces)
        self.data_refs["list"] = lo_list

        # Add region to the tei document
        tei_header = sdstream.root.findall(".//teiHeader",
self.xml_namespaces)[0]
        prof_desc = ET.SubElement(tei_header,"profileDesc")
        set_desc = ET.SubElement(prof_desc, "settingDesc")
        place = ET.SubElement(set_desc, "place")
        region = ET.SubElement(place, "region")
        if self.pid_var == "sty":
          region.text = "Steiermark"
        elif self.pid_var == "vor":
          region.text = "Vorarlberg"
        elif self.pid_var == "tir":
          region.text = "Tirol"
        elif self.pid_var == "car":
          region.text = "Kärnten"
        elif self.pid_var == "bur":
          region.text = "Burgenland"
        elif self.pid_var == "vie":
          region.text = "Wien"
        elif self.pid_var == "sal":
          region.text = "Salzburg"
        else:
          region.text = "NOT_IMPLEMENTED"

        # Assign title (needed for the didadics glossary)
        sdstream.root.findall(".//title", self.xml_namespaces)[0].text =
"Fixierte Vermittlungsangebote " + region.text

        #
        sdstream.root.findall(".//p", self.xml_namespaces)[0].text = "Fixierte
Vermittlungsangebote Glossar, born digital"

        # Assign didactics context
        ab_elem = sdstream.root.findall(".//ab", self.xml_namespaces)[0]
        ET.SubElement(ab_elem, "ref", target="context:derladidactics",
type="context")

        row_funcs = [self._handle_row_elems]
        col_funcs = [
          self._fb_beschreibung,
          self._handleFBOrtGemeinde,
          self._handle_fb_zeitbedarf,
          self._handle_fb_zielalter,
          self._handle_fb_ort,
          self._pass_col,
          self._pass_col,
          self._handle_mm_title,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._pass_col,
          self._handle_linked_places
          ]

        self.loop_data(row_funcs, col_funcs, skip_rows=2)


    def _handle_row_elems(self, row_ind: int, row: List[str], data_refs:
Dict[str, Any]):
      """
      Adds item per item with and assigns xml:ids.
      Calls list_keywords to add <ptr> references where keywords were used.
      Args:
        row_ind (int): The index of the row.
        row (List[str]): The list of strings representing the row.
        data_refs (Dict[str, Any]): A dictionary containing references to
various XML elements.

      Returns:
        Element: The newly created XML element representing the item.
      """
      cur_item = ET.SubElement(data_refs["list"], "item", {"xml:id":
f"fix{self.pid_var}{row_ind + 1}"})
      data_refs["cur_item"] = cur_item

      ET.SubElement(data_refs["cur_item"], "objectType").text = "Learning
Object LO"

      cur_measure = ET.SubElement(data_refs["cur_item"], "measure",
type="lo_metadata")
      data_refs["cur_measure"] = cur_measure

      cur_linkgrp = ET.SubElement(data_refs["cur_item"], "linkGrp",
type="linkedPlaces")
      data_refs["cur_linkgrp"] = cur_linkgrp
      ET.SubElement(data_refs["cur_linkgrp"], "ptr",
target=f"https://gams.uni-graz.at/o:derla.{self.pid_var}{row_ind + 1}",
type="main")

      return cur_item


    def _fb_beschreibung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Adds label of current keyword
       Args:
         col_ind (int): The index of the column.
         col_val (str): The value of the column.
         data_refs (Dict[str, Any]): A dictionary containing references to
data.

      Returns:
        None

      """
      ET.SubElement(data_refs["cur_item"], "desc").text = col_val

    def _handleFBOrtGemeinde(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
```

```
      Handles the processing of the 'Ort, Gemeinde EO/EZ' column value and
adds it to the XML tree.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to data
objects.

      Returns:
        None
      """

      ET.SubElement(data_refs["cur_measure"], "geogFeat", type="region",
rend="Ort, Gemeinde EO/EZ").text = col_val


    def _handle_fb_zeitbedarf(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Adds the text of the glossary
      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to
data.

      Returns:
        None

      """
      ET.SubElement(data_refs["cur_measure"], "time", type="zeitbedarf",
rend="Zeitbedarf").text = col_val

    def _handle_fb_zielalter(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Handles the 'fb_zielalter' column value and adds it to the XML tree.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to the
XML tree and other data.

      Returns:
        None
      """

      ET.SubElement(data_refs["cur_measure"], "time", type="alter",
rend="Alter").text = col_val

    def _handle_fb_ort(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Handles the 'fb_ort' column by adding a geographic feature element to
the current measure element.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """
      ET.SubElement(data_refs["cur_measure"], "geogFeat", type="zielort",
rend="Vermittlungsort").text = col_val

    def _pass_col(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
      """
      Function to skip col if not needed
      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None

      """
      pass
    def _handle_linked_places(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Handles creation ptr elements inside linkgrp for additionally referenced
places.
      Args:
          col_ind (int): The column index.
          col_val (str): The column value.
          data_refs (Dict[str, Any]): The dictionary containing data
references.

        Returns:
          None

      """
      place_nums = col_val.split(";")

      for num in place_nums:
        if num == "" : continue
        ET.SubElement(data_refs["cur_linkgrp"], "ptr",
target=f"https://gams.uni-graz.at/o:derla.{self.pid_var}{num}",
type="additional")

    def _handle_mm_title(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Assigns title of mehr machen to the lo item.
      Parameters:
      col_ind (int): The index of the column.
      col_val (str): The value of the column.
      data_refs (Dict[str, Any]): A dictionary containing references to data
objects.

      Returns:
      None
      """
      ET.SubElement(data_refs["cur_item"], "title",
type="mehrMachenTitle").text = col_val
```

xml_build/archive_of_names/DERLAPersListHandler.py

```python
from typing import Dict, List, Tuple, Any, Optional
import xml.etree.ElementTree as ET
from cm.builder.GAMSSourceBuilder import GAMSSOURCEBuilder
from harvest.source_types.CSVHarvester import CSVHarvester
from cm.source_templates.TEI.PersList import TEI_PERSLIST_TEMPLATE
from cm.SourceDatastream import SourceDatastream
from cm.static_namespaces import TEI_NAMESPACES
import os


class DERLAPersListHandler(GAMSSOURCEBuilder, CSVHarvester):
    def __init__(self, env):
        """
        Class for handling the construction of DERLA person list XML.

        Args:
            env: The environment file.

        """

        # 01. Initialization and .env
        # pass in env file
        GAMSSOURCEBuilder.__init__(
            self, TEI_PERSLIST_TEMPLATE, TEI_NAMESPACES, env)
        CSVHarvester.__init__(self, build_max_rows=10000)

        # 02. Harvest data
        # 02a. get data from gsheets
        self.get(self._env["get_param"])
        # 02b. backup
        self.backup(
            os.getcwd() + os.sep + "backup" + os.sep + "g_sheet_")

        # 03. Build data
        self.apply_data()
        # 04. Post processing

        # 05. Write to file
        # build SOURCE_DATA stream
        # data_stream = self.build_dsource_stream()
        # write stream to xml
        # self.write_source_xml(
        #     data_stream, "my_derla_file")

        # can use here to out all created SourceDatastreams
        self.write_source_xml()

    def apply_data(self):
        """
        Method add data to xml

        This method adds data to the XML file. It performs the following
steps:
        1. Builds the source-datastream and defines pid globals.
        2. Constructs the pid.
        3. Assigns the pid to the publication statement element in the XML.
        4. Assigns the listPerson element in the XML to the data_refs
dictionary.
        5. Assigns a title to the XML based on the value of pid_var.
        6. Defines a list of row functions and column functions.
        7. Loops through the data using the row and column functions.

        """



        # 01. first build source-datastream + define pid globals
        # (will expose many needed class fields)
        # 01b. construct pid
        self.pid = self.pid_static + "pers" + self.pid_var
        sdstream = self.build_dsource_stream(self.pid)

        # 02. Global xml data assignment (like pid etc.)
        pubstmt_elem = sdstream.root.find(
            ".//publicationStmt", self.xml_namespaces)
        ET.SubElement(pubstmt_elem, "idno", type="PID").text = self.pid
        listpers_elem = sdstream.root.find(
            ".//listPerson", self.xml_namespaces)
        self.data_refs["listpers"] = listpers_elem

        # Assign title
        titles = sdstream.root.findall(".//title", self.xml_namespaces)
```

```python
        if self.pid_var == "vor":
            titles[0].text = "Personen Vorarlberg"
        elif self.pid_var == "sty":
            titles[0].text = "Personen Steiermark"
        elif self.pid_var == "bur":
            titles[0].text = "Personen Burgenland"
        elif self.pid_var == "tir":
            titles[0].text = "Personen Tirol"
        elif self.pid_var == "car":
            titles[0].text = "Personen Kärnten"
        elif self.pid_var == "sal":
            titles[0].text = "Personen Salzburg"
        elif self.pid_var == "vie":
            titles[0].text = "Personen Wien"
        else:
            raise ValueError(f"Not supported pid var {self.pid_var}")


        row_funcs = [
            self._handle_row_elems
        ]
        col_funcs = [
            self._handleSurName,
            self._handleForeName,
            self._handle_place_references,
            self._handle_maiden_name,
            self._handle_alt_name,
            self._handle_free_text,
            self._skip_col,  # comment intern
            self._skip_col,  # GND
            self._skip_col,  # VIAF
            self._handle_wikidata,  # Wiki-Data
            self._handle_persec_cause,   # prosec01
            self._handle_persec_cause,
            self._handle_female,
            self._handle_adolescents,
            self._handle_birth_date,
            self._handle_death_date
        ]

        self.loop_data(row_funcs, col_funcs, skip_rows=2)
```

```python
    def _handle_row_elems(self, row_ind: int, row: List[str], data_refs:
Dict[str, Any]):
"""
        Handles the row elements in the DERLAPersListHandler class.

        Args:
            row_ind (int): The index of the row.
            row (List[str]): The list of strings representing the row.
            data_refs (Dict[str, Any]): The dictionary containing data
references.

        Returns:
            Tuple[Element, Element]: A tuple containing the current person
element and the person name element.
        """

        cur_pers_elem = ET.SubElement(data_refs["listpers"], "person", {
                                      "xml:id": f"pers{self.pid_var}{row_ind +
1}"})
        data_refs["cur_pers"] = cur_pers_elem
        persname_elem = ET.SubElement(cur_pers_elem, "persName")
        data_refs["cur_persname"] = persname_elem
        return (cur_pers_elem, persname_elem)

    def _handleSurName(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]) -> ET.Element:
"""
        Handles the surname element in the XML structure.

        Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing references to
data elements.

        Returns:
            ET.Element: The created surname element.
        """

        surname = ET.SubElement(data_refs["cur_persname"], "surname")
        surname.text = col_val
        return surname
    def _handleForeName(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]) -> ET.Element:
"""
            Creates and returns a new XML element for the forename.

            Args:
                col_ind (int): The column index.
                col_val (str): The column value.
                data_refs (Dict[str, Any]): A dictionary containing references
to data objects.

            Returns:
                ET.Element: The newly created XML element for the forename.
            """

        forename = ET.SubElement(data_refs["cur_persname"], "forename")
        forename.text = col_val
        return forename

    def _handle_place_references(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handle place references for a given column value.

        Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing data
references.

        Returns:
            None
        """

        linkgrp = ET.SubElement(data_refs["cur_pers"], "linkGrp")
        col_val = col_val.replace(" ", "")
        refs = col_val.split(";")
        for ref in refs:
            if ref == "":
                continue
            ET.SubElement(linkgrp, "ptr", target=f"{self.pid_base}{ref}")
```

```python
    def _handle_maiden_name(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]) -> Optional[ET.Element]:
"""
        Handles the maiden name column value and returns an XML element
representing the maiden name.

        Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing references to
data elements.

        Returns:
            Optional[ET.Element]: An XML element representing the maiden name,
or None if the column value is '-'.
        """

        if col_val == "-":
            return None
        maiden_name = ET.SubElement(
            data_refs["cur_persname"], "note", type="maiden_name")
        maiden_name.text = col_val
        return maiden_name

    def _handle_alt_name(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the alternative name column value.

        Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing references to
data objects.

        Returns:
            None: If the column value is '-'.
            Element: The alternative name element if the column value is not
'-'.
        """
        if col_val == "-":
            return None
        alt_name = ET.SubElement(
            data_refs["cur_persname"], "note", type="alt_name")
        alt_name.text = col_val

    def _handle_free_text(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handle free text data by creating a list event element with type
'prosecutions' and setting the description.

        Args:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): A dictionary containing references to
data elements.

        Returns:
            None
        """

        list_event = ET.SubElement(
            data_refs["cur_pers"], "listEvent", type="prosecutions")
        data_refs["list_event"] = list_event
        list_event_head = ET.SubElement(list_event, "head")
        ET.SubElement(list_event_head, "desc").text = col_val

    def _skip_col(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
        Skips a column based on the column index and value.

        Args:
            col_ind (int): The index of the column to skip.
            col_val (str): The value of the column to skip.
            data_refs (Dict[str, Any]): A dictionary containing references to
data.

        Returns:
            None
        """
        pass

    def _handle_wikidata(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
        """
        Generate idnos with wikidata:
        - like: <idno
type="wikidata">http://www.wikidata.org/entity/Q2283480</idno>

        Args:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): The data references.

        Returns:
            None
        """

        if col_val != "" and col_val != "-":
            if len(col_val) > 1:
                ET.SubElement(data_refs["cur_pers"], "idno",
type="wikidata").text = f"http://www.wikidata.org/entity/{col_val}"

    def _handle_persec_cause(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the persecution cause for a person.

        Args:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): The data references.

        Returns:
            None
        """

        if col_val == "-" or col_val == "":
            return

        kword_id, kword_label = self.resolve_pers_keywords(col_val)
        event = ET.SubElement(data_refs["list_event"], "event")
        label = ET.SubElement(event, "label")
        ET.SubElement(label, "term",
ref=f"{self.origin}/o:derla.keywords#{kword_id}").text = kword_label
        self.logger.debug(f"Added persecution cause: {kword_label} to person.
According to given val {col_val}.")


    def _handle_female(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
        """
            Adds sex statement only when explicitly called as women.

            Parameters:
            col_ind (int): The column index.
            col_val (str): The column value.
            data_refs (Dict[str, Any]): The data references.

            Returns:
            None
            """

        if col_val == "ja":
            sex_elem = ET.SubElement(data_refs["cur_pers"], "sex")
            kword_id, kword_label = self.resolve_pers_keywords("frauen")
            ET.SubElement(sex_elem, "term",
ref=f"{self.origin}/o:derla.keywords#{kword_id}").text = kword_label
            self.logger.debug(f"Added female stmt: {kword_label} to person.
According to given val {col_val}.")
        elif col_val != "nein":
            self.logger.error(f"Female stmt was neither set to 'ja' nor 'nein'
at pid: {str(self.pid)}. Didn't generate sex stmt")

    def _handle_adolescents(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the 'adolescents' column value for a person.

        Args:
            col_ind (int): The index of the column.
            col_val (str): The value of the column.
            data_refs (Dict[str, Any]): A dictionary containing data
references.

        Returns:
            None
        """

        if col_val == "ja":
            age_elem = ET.SubElement(data_refs["cur_pers"], "age")
            kword_id, kword_label = self.resolve_pers_keywords("jugendliche")
            ET.SubElement(age_elem, "term",
ref=f"{self.origin}/o:derla.keywords#{kword_id}").text = kword_label
            self.logger.debug(f"Added age stmt: {kword_label} to person.
According to given val {col_val}.")
        elif col_val != "nein":
            self.logger.error(f"Age stmt was neither set to 'ja' nor 'nein' at
pid: {str(self.pid)}. Didn't generate sex stmt")

    def _handle_birth_date(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
            Handles the birth date column value and constructs the
corresponding XML structure.

            Args:
                col_ind (int): The index of the birth date column.
                col_val (str): The value of the birth date column.
                data_refs (Dict[str, Any]): A dictionary containing references
to the XML elements.

            Returns:
                None
            """

        if col_val == "-":
            self.logger.debug("Skipping birth date value because set to '-'")
            return
        birth_elem = ET.SubElement(data_refs["cur_pers"], "birth")
        date_elem = ET.SubElement(birth_elem, "date")
        date_elem.text = col_val
        self.logger.info("Succesfully constructed birth date xml structure.")
    def _handle_death_date(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the death date column value and constructs the corresponding
XML structure.

        Args:
            col_ind (int): The index of the death date column.
            col_val (str): The value of the death date column.
            data_refs (Dict[str, Any]): A dictionary containing references to
the current person's XML element and other data.

        Returns:
            None
        """

        if col_val == "-":
            self.logger.debug("Skipping deat date col because set to '-'")
            return
        death_elem = ET.SubElement(data_refs["cur_pers"], "death")
        date_elem = ET.SubElement(death_elem, "date")
        date_elem.text = col_val
        self.logger.info("Succesfully constructed death date xml structure.")


    def resolve_pers_keywords(self, key_sheet_val: str) -> Tuple[str, str]:
        """
            Resolve keywords (instead of SKOS).

            This method takes a keyword value from a Google Sheet and returns
a tuple containing the ID and label of the corresponding keyword in DERLA.

            :param key_sheet_val: The keyword value from the Google Sheet.
            :type key_sheet_val: str
            :returns: A tuple containing the ID and label of the corresponding
keyword in DERLA.
            :rtype: Tuple[str, str]
            """

        """
        value_dict = {
            "als Jude verfolgt": "Als Jude oder Jüdin verfolgt (Archiv der
Namen)",
            "Altar": "Altar",
            "Arkade": "Arkade",
            "Bild": "Bild",
            "Brunnen": "Brunnen",
            "brücke": "Brücke",
            "denkmal": "Denkmal",
            "euthanasie": "NS-Euthanasie",
            "frauen": "Frauen explizit erwähnt",
            "freimonument": "freistehendes Monument",
            "gasse": "Gasse",
            "gebäude":  "Gebäude",
            "Gedenkkreuz": "Gedenkkreuz",
            "gedenktafel": "Gedenktafel",
            "gedenkstein": "Gedenkstein",
            "gestapo/ss": "Gestapo/SS/NSDAP (Archiv der Namen)",
            "Gipfelkreuz": "Gipfelkreuz",
            "glasfenster": "Glasfenster",
            "glocke": "Glocke",
            "grab": "Grab",
            "grabanlage": "Grabanlage",
            "homosexuelle": "Homosexuelle Opfer",
            "intervention": "Künstlerische Intervention",
            "jenische": "Jenische",
            "jugendliche": "Jugendliche explizit erwähnt",
            "jüdischeopfer, als Jude verfolgt": "Jüdische Opfer - Als Jude
oder Jüdin verfolgt",
            "jüdischeopfer, gemeindeeinrichtung": "Jüdische Opfer -
Gemeindeeinrichtung",
            "jüdischeopfer, ikg": "Jüdische Opfer - Israelitische
Kultusgemeinde",
            "jüdischeopfer, kz": "Jüdische Opfer - KZ",
            "jüdischeopfer, sammelwohnung": "Jüdische Opfer - Sammelwohnung",
            "jüdischeopfer, todesmarsch": "Jüdische Opfer - Todesmarsch",
            "jüdischeopfer, zwangsarbeit": "Jüdische Opfer - Zwangsarbeit",
            "Kapelle": "Kapelle",
            "kärntner_slowenen": "Kärntner SlowenInnen",
            "kollektiv": "Kollektiv",
            "kriegerdenkmal": "Kriegerdenkmal",
            "kz_opfer": "KZ Opfer",
            "ns_justiz_opfer": "Opfer der NS Justiz",
            "Orgel": "Orgel",
            "ohnezeichen": "Ort ohne Zeichen",
            "opfer_faschismus": "Opfer des Faschismus",
            "öffentlich_zugänglich": "Öffentlich zugänglich",
            "passage": "Passage",
            "platz": "Platz",
            "roma": "Roma/Romnija und Sinti/Sintize und andere Fahrende wie
Lovara",
            "sammelwohnung": "Sammelwohnung",
            "schriftskulptur": "Schriftskultpur",
            "siedlung": "Siedlung",
            "skulptur": "Skulptur",
            "soldaten, alliierte": "Alliierte Soldaten",
            "soldaten, deserteure": "Soldaten - Deserteure",
            "soldaten, kriegsgefangene": "Soldaten - Kriegsgefangene",
            "soldaten, wehrmacht": "Soldaten - Wehrmacht",
            "statue": "Statue",
            "stolperstein": "Stolperstein",
            "strasse": "Straße",
            "terror, gefängnis": "Orte des Terrors - Gefängnis",
            "terror, gestapossnsdap": "Orte des Terrors - Gestapo/SS/NSDAP",
            "terror, hinrichtungsstätten": "Orte des Terrors -
Hinrichtungsstätten",
            "terror, justiz": "Orte des Terrors - Justiz",
            "terror, kz": "Orte des Terrors - KZ",
            "verkehrsfläche": "Verkehrsfläche",
            "Weg": "Weg",
            "widerstand, individuell": "Individueller Widerstand",
            "widerstand, politisch": "Politischer Widerstand",
            "widerstand, religiös": "Religiöser Widerstand",
            "widerstand, Kärntner PartisanInnenwiderstand": "Kärntner
PartisanInnenwiderstand",
            "zivileopfer": "Zivile Opfer",
            "zwangsarbeiter": "ZwangsarbeiterInnen",
            "Inschrift":"Inschrift"
        }

        lowered_dict =  {k.lower(): v for k, v in value_dict.items()}
        lowerd_keysheet_val = key_sheet_val.lower()

        key_index = 99999999
        keyword_label: str = ""
        try:
            keyword_label = lowered_dict[lowerd_keysheet_val]
            key_index = list(lowered_dict.keys()).index(lowerd_keysheet_val)
        except (ValueError, KeyError, IndexError):
            self.logger.error(f"Given keyword from gsheet not found in value
dict: {key_sheet_val}. Assigning dummy value.")
            keyword_label = "NOT_FOUND_ERR"

        return (f"keyword{key_index + 1}", keyword_label)
```

xml_build/archive_of_names/DERLATeiLoHandler.py

```
from cm.SourceDatastream import SourceDatastream
from cm.builder.GAMSSourceBuilder import GAMSSOURCEBuilder
from harvest.source_types.CSVHarvester import CSVHarvester
from cm.source_templates.TEI.TeiLo import TEI_LO_TEMPLATE
from cm.static_namespaces import TEI_NAMESPACES
import os
import xml.etree.ElementTree as ET
from typing import Dict, List, Tuple, Any, Optional
from cm.xml_operations import clean_xml_string
from pathlib import Path
from shutil import copy
import re

class DERLATeiLoHandler(GAMSSOURCEBuilder, CSVHarvester):
    def __init__(self, env) -> None:
        """
      Class to handle building of Derla's fixed learning objects.

      Args:
      env: The environment variable.
    """



        # Load required dependent gsheet data

        # 01. Initialization and .env
        GAMSSOURCEBuilder.__init__(self, TEI_LO_TEMPLATE, TEI_NAMESPACES, env)
        CSVHarvester.__init__(self, build_max_rows=99999)
        # 02. Harvest data
        # 02a. get data from gsheets
        self.get(self._env["get_param"])

        # 02b. backup
        self.backup(os.getcwd() + os.sep + "backup" + os.sep +
"g_sheet_sty_fix")

        # 03. Build data
        self.apply_data()

        # 04. Post processing (no param to method will write out all intern
xmls)
        self.write_source_xml()

        # 05. copy_images_to_ingest
        self.copy_images_to_ingest()

    def apply_data(self):
        """
        Method to add required data to the xml
        """



        # row and col functions
        row_funcs = [self._handle_row_elems]

        col_funcs = [
          self._handleFBBeschreibung,
          self._handleFBOrtGemeinde,
          self._handleFBZeitbedarf,
          self._handleFBAlter,
          self._handleFBOrt,
          self._handleMEText,
          self._handleMELiteratur,
          self._handleMMTitle,
          self._handleMMDidComm,
          self._handleMMAuftrag,
          self._handleMMHinweis,
          self._handleMMZiele,
          self._handleMMLiteratur,
          self._handleQTText,
          self._handleQTBeschr,
          self._handleQTZitat,
          self._handleImg01Dummy,
          self._handleImg01Beschreibung,
          self._handleImg01Orientierung,
          self._handleImg01Title,
          self._handleImg02Dummy,
          self._handleImg02Beschreibung,
          self._handleImg02Orientierung,
          self._handleImg02Title,
          self._handleVimeoLink,
          self._handleVimeoBeschreibung,
          self._handleVimeoTitel,
          self._handleAudio01Link,
          self._handleAudio01Beschreibung,
          self._handleAudio01Titel,
          self._handleAudio02Link,
          self._handleAudio02Beschreibung,
          self._handleAudio02Titel,
          self._handle_linked_places
          ]

        # 2 rows have to skipped
        self.loop_data(row_funcs, col_funcs, skip_rows=0)

    def _handle_row_elems(self, row_ind: int, row: List[str], data_refs:
Dict[str, Any]):
"""
      Handle the elements in a row.

      Args:
        row_ind (int): The index of the row.
        row (List[str]): The row data.
        data_refs (Dict[str, Any]): A dictionary to store references to XML
elements.

      Returns:
        None
      """



      # this has all to be done in row func
```

```
        self.pid = self.pid_static + "fix" + self.pid_var + str(row_ind + 1)

        sdstream = self.build_dsource_stream(self.pid)


        # 02. Global xml data assignment (like pid etc.)
        pubstmt_elem = sdstream.root.find(
            ".//publicationStmt", self.xml_namespaces)
        ET.SubElement(pubstmt_elem, "idno", type="PID").text = self.pid

        # assign connected place of remembrance
        linked_place_uri =
f"https://gams.uni-graz.at/{self.pid_static}{self.pid_var}{row_ind + 1}"
        sdstream.root.findall(".//idno[@type='URI']",
self.xml_namespaces)[0].text = linked_place_uri

        # collect references
        data_refs["sdstream"] = sdstream
        data_refs["tei_elem"] = sdstream.root
        data_refs["body"] = sdstream.root.findall(".//body",
self.xml_namespaces)[0]

        # refs for LO specific elements
        data_refs["factbox"] = sdstream.root.findall(".//div[@type='factbox']",
self.xml_namespaces)[0]
        data_refs["mehrErfahrenDiv"] =
sdstream.root.findall(".//div[@type='mehrErfahren']", self.xml_namespaces)[0]
        data_refs["mehrMachenDiv"] =
sdstream.root.findall(".//div[@type='mehrMachen']", self.xml_namespaces)[0]
        data_refs["quellenTextDiv"] =
sdstream.root.findall(".//div[@type='quellenText']", self.xml_namespaces)[0]

        data_refs["link_grp_elem"] = None


    def _handleFBBeschreibung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the 'FBBeschreibung' column value and adds it to the factbox
element.

        Args:
```

```
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """

      ET.SubElement(data_refs["factbox"], "p", rend="Beschreibung").text =
col_val

    def _handleFBOrtGemeinde(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the processing of the 'Ort, Gemeinde EO/EZ' column value.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to data
objects.

      Returns:
        None
      """

      ET.SubElement(data_refs["factbox"], "p", rend="Ort, Gemeinde
EO/EZ").text = col_val

    def _handleFBZeitbedarf(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the 'FBZeitbedarf' column value and adds it to the factbox
element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.
```

```
      Returns:
        None
      """

      ET.SubElement(data_refs["factbox"], "p", rend="Zeitbedarf").text =
col_val

    def _handleFBAlter(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handles the FBAlter column value and adds it to the factbox element in
the XML.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """

      ET.SubElement(data_refs["factbox"], "p", rend="Alter").text = col_val

    def _handleFBOrt(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handles the 'FBOrt' column value and adds it to the factbox element in
the XML.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """
      ET.SubElement(data_refs["factbox"], "p", rend="Vermittlungsort").text =
col_val

    def _handleMEText(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handles the METext element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary of data references.

      Returns:
        None
      """

      self.add_xmlp(data_refs["mehrErfahrenDiv"], col_val,
data_refs["sdstream"])

    def _handleMELiteratur(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the 'MELiteratur' column value and adds it to the XML
structure.

        Args:
          col_ind (int): The index of the column.
          col_val (str): The value of the column.
          data_refs (Dict[str, Any]): A dictionary containing references to
the XML elements.

        Returns:
          None
        """
      ET.SubElement(data_refs["mehrErfahrenDiv"], "caption").text =
"Literatur"
      self.split_to_xml_list(col_val,
container_elem=data_refs["mehrErfahrenDiv"])

    def _handleMMTitle(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
```

```
"""
        Sets the 'head' element text in the 'mehrMachenDiv' element and the
text of the first 'title' element in the XML document.

        Args:
          col_ind (int): The column index.
          col_val (str): The column value.
          data_refs (Dict[str, Any]): A dictionary containing references to
relevant data.

        Returns:
          None
        """


      ET.SubElement(data_refs["mehrMachenDiv"], "head").text = col_val
      data_refs["sdstream"].root.findall(".//title",
self.xml_namespaces)[0].text = col_val
      pass

    def _handleMMDidComm(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the MehrMachen Didaktischer Kommentar.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """

      ET.SubElement(data_refs["mehrMachenDiv"], "caption").text =
"Didaktischer Kommentar"
      self.add_xmlp(data_refs["mehrMachenDiv"], col_val,
data_refs["sdstream"])

    def _handleMMAuftrag(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
```

```
      Handles the MMAuftrag data by adding a caption element to the
mehrMachenDiv element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None
      """

      ET.SubElement(data_refs["mehrMachenDiv"], "caption").text =
"Arbeitsaufträge"
      self.add_xmlp(data_refs["mehrMachenDiv"], col_val,
data_refs["sdstream"])

    def _handleMMHinweis(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the MMHinweis element by adding a caption to the mehrMachenDiv
element and
      calling the add_xmlp method to add col_val to the sdstream.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None
      """

      ET.SubElement(data_refs["mehrMachenDiv"], "caption").text =
"Vermittlungshinweise"
      self.add_xmlp(data_refs["mehrMachenDiv"], col_val,
data_refs["sdstream"])

    def _handleMMZiele(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
```

```
      Handles the processing of MMZiele column values and adds them to the XML
structure.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to
various XML elements.

      Returns:
        None
      """

      ET.SubElement(data_refs["mehrMachenDiv"], "caption").text =
"Vermittlungsziele"
      self.split_to_xml_list(col_val,
container_elem=data_refs["mehrMachenDiv"])

    def _handleMMLiteratur(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the 'MMLiteratur' column value and adds it to the XML structure.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to the
XML elements.

      Returns:
        None
      """

      ET.SubElement(data_refs["mehrMachenDiv"], "caption").text = "Literatur"
      self.split_to_xml_list(col_val,
container_elem=data_refs["mehrMachenDiv"])

    def _handleQTText(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handle the processing of a QTText element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary of data references.

      Returns:
        None
      """

      self.add_xmlp(data_refs["quellenTextDiv"], col_val,
data_refs["sdstream"])

    def _handleQTBeschr(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handles the 'QTBeschr' column by adding the title/description to the
'quellenTextDiv' element.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to XML
elements.

      Returns:
        None
      """

      ET.SubElement(data_refs["quellenTextDiv"], "caption").text = "Titel /
Beschreibung"
      ET.SubElement(data_refs["quellenTextDiv"], "p").text = col_val

    def _handleQTZitat(self, col_ind: int, col_val: str, data_refs: Dict[str,
Any]):
"""
      Handles the processing of a QTZitat element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.
```

```
      Returns:
        None
      """

      ET.SubElement(data_refs["quellenTextDiv"], "caption").text =
"Quellenzitat"
      ET.SubElement(data_refs["quellenTextDiv"], "p").text = col_val

    def _handleImg01Dummy(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      This method handles the Img01Dummy data.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """
      pass

    def _handleImg01Beschreibung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the description of the image.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The dictionary containing data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["facs_elem"] = ET.SubElement(data_refs["tei_elem"],
"facsimile")
```

```
      data_refs["graphic01_elem"] = ET.SubElement(data_refs["facs_elem"],
"graphic",{"xml:id":"IMAGE.1", "url": "file:///1.JPG",
"mimeType":"image/jpeg"})
      ET.SubElement(data_refs["graphic01_elem"], "desc").text = col_val

    def _handleImg01Orientierung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the 'Img01Orientierung' column value.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      pass

    def _handleImg01Title(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the title for img01.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      ET.SubElement(data_refs["graphic01_elem"], "desc", type="title").text =
col_val
```

```
    def _handleImg02Dummy(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      This method handles the Img02Dummy data.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """

      pass

    def _handleImg02Beschreibung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the description for the second image.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): Dictionary containing references to data
elements.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      if not data_refs["facs_elem"]:
        self.logger.error("The facsimilie element should've generated in the
first image. Aborting operations")
        raise ValueError("The facsimilie element should've generated in the
first image. Aborting operations")

      data_refs["graphic02_elem"] = ET.SubElement(data_refs["facs_elem"],
"graphic", {"xml:id":"IMAGE.2", "url": "file:///2.JPG",
"mimeType":"image/jpeg"})
```

```
      ET.SubElement(data_refs["graphic02_elem"], "desc").text = col_val

    def _handleImg02Orientierung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the 'Img02Orientierung' column value.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      pass

    def _handleImg02Title(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handle the title of the Img02 element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      ET.SubElement(data_refs["graphic02_elem"], "desc", type="title").text =
col_val

    def _handleVimeoLink(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
```

```
"""
      Handles the Vimeo link column value.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): The dictionary containing data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      link_grp = self.get_create_linkgrp(data_refs)
      data_refs["vimeo_ptr_elem"] = ET.SubElement(link_grp, "ptr",
ref=col_val)

    def _handleVimeoBeschreibung(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the Vimeo Beschreibung column value.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["vimeo_ptr_elem"].set("rendition",col_val)

    def _handleVimeoTitel(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the Vimeo title by setting the 'rend' attribute of the
'vimeo_ptr_elem' element.
```

```
      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["vimeo_ptr_elem"].set("rend",col_val)

    def _handleAudio01Link(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
        Handles the audio01 link by creating a ptr element in the XML tree.

        Args:
          col_ind (int): The column index.
          col_val (str): The column value.
          data_refs (Dict[str, Any]): The dictionary containing data
references.

        Returns:
          None
        """

      if col_val == "" or col_val == "-":
        return
      link_grp = self.get_create_linkgrp(data_refs)
      data_refs["audio01_ptr_elem"] = ET.SubElement(link_grp, "ptr",
ref=col_val)

      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio01 link to xml {pid}")

    def _handleAudio01Beschreibung(self, col_ind: int, col_val: str,
data_refs: Dict[str, Any]):
"""
      Handles the audio01 Beschreibung column value.
      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to data
elements.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["audio01_ptr_elem"].set("rendition",col_val)

      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio01 Beschreibung to xml {pid}")

    def _handleAudio01Titel(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the 'Audio01Titel' column value and updates the corresponding
XML element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to
relevant data.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["audio01_ptr_elem"].set("rend",col_val)

      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio01 Titel to xml {pid}")

    def _handleAudio02Link(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
```

```
"""
        Handles the audio02 link in the XML file.

        Args:
          col_ind (int): The column index.
          col_val (str): The column value.
          data_refs (Dict[str, Any]): The dictionary containing data
references.

        Returns:
          None
        """

      if col_val == "" or col_val == "-":
        return
      link_grp = self.get_create_linkgrp(data_refs)
      data_refs["audio02_ptr_elem"] = ET.SubElement(link_grp, "ptr",
ref=col_val)

      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio02 link to xml {pid}")

    def _handleAudio02Beschreibung(self, col_ind: int, col_val: str,
data_refs: Dict[str, Any]):
"""
      Handles the 'Audio02Beschreibung' column value and updates the XML
accordingly.

      Args:
        col_ind (int): The index of the column.
        col_val (str): The value of the column.
        data_refs (Dict[str, Any]): A dictionary containing references to
relevant data.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["audio02_ptr_elem"].set("rendition",col_val)
      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio02 Beschreibung to xml {pid}")

    def _handleAudio02Titel(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
"""
      Handles the Audio02 Titel column value and updates the corresponding XML
element.

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing references to
relevant data.

      Returns:
        None
      """

      if col_val == "" or col_val == "-":
        return
      data_refs["audio02_ptr_elem"].set("rend",col_val)

      pid = data_refs["sdstream"].pid
      self.logger.debug(f"Added Audio02 Titel to xml {pid}")

    def _handle_linked_places(self, col_ind: int, col_val: str, data_refs:
Dict[str, Any]):
      """
      Adds link group with further linked places

      Args:
        col_ind (int): The column index.
        col_val (str): The column value.
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        None

      """
      if col_val == "" or col_val == "-":
        return
      setting_desc = data_refs["sdstream"].root.findall(".//settingDesc",
self.xml_namespaces)[0]
      list_place = ET.SubElement(setting_desc, "listPlace")
      place_nums = col_val.split(";")
      for num in place_nums:
        if place_nums == "": continue
        place_elem = ET.SubElement(list_place, "place", type="additional")
        ET.SubElement(place_elem, "idno", type="URI").text =
f"https://gams.uni-graz.at/{self.pid_static}{self.pid_var}{num}"

    def add_xmlp(self, container_elem: ET.Element, xml_str: str,
source_datastream: SourceDatastream):
      """
      Method adds xml string to intern built xml. String must start with.
          Also removes unnecessary whitespace + line breaks etc. from string.

          :param container_elem: The container element to which the xml string
should be added.
          :type container_elem: ET.Element

          :param xml_str: The xml string to be parsed and added.
          :type xml_str: str

          :param source_datastream: The source datastream object.
          :type source_datastream: SourceDatastream

      """
      # clean string a bit
      xml_str = clean_xml_string(xml_str)

      if "<p>" in xml_str:
        # will parse xml in column and insert at content div
        try:
          root = ET.fromstring(xml_str)
          # giving the count will lead to appending the element
          child_count: int = len(list(container_elem.iter()))
          container_elem.insert(child_count, root)
          self.logger.info(f"Succescfully added xml for pid:
{source_datastream.pid}")
        except:
          self.logger.error(f"Failed to parse xml string for pid:
{source_datastream.pid} from string: {xml_str}");
          # raise ValueError
      else:
        self.logger.error(f"No <p> inside xml string for pid:
{source_datastream.pid} from string: {xml_str}");
        # raise ValueError


    def split_to_xml_list(self, split_able: str, container_elem: ET.Element):
      """
      Splits a string into an array according to the delimiter ";".
          Adds a tei:list element and tei:item elements as a list
representation.

          :param split_able: The string to be split.
          :param container_elem: The element to which the list should be
applied.

      """
      list: List[str] = split_able.split(";")
      list_elem = ET.SubElement(container_elem, "list")
      for item in list:
        if item =="": continue
        ET.SubElement(list_elem, "item").text = item

    def get_create_linkgrp(self, data_refs: Dict[str, Any]):
      """
      Tries to get linkGrp element. If not available adds it to TEI.
      Args:
        data_refs (Dict[str, Any]): A dictionary containing data references.

      Returns:
        Element: The linkGrp element.
      """
      if data_refs["link_grp_elem"] == None:
        data_refs["link_grp_elem"] = ET.SubElement(data_refs["body"],
"linkGrp")
        return data_refs["link_grp_elem"]
      else:
         return data_refs["link_grp_elem"]
    def copy_images_to_ingest(self):
      """
      Handles the process of copying images from Google Drive to ingest files.

        This method copies images from a specified Google Drive folder to the
ingest folder.
        It loops through the built files in the ingest folder and searches for
corresponding images in the Google Drive folders.
        The images are then copied to the respective locations in the ingest
folder.

        Args:
          None

        Returns:
          None
      """

      working_dir_path = str(os.getcwd())
      home_path = str(Path.home())
      # path to google drive images!
      data_path = f"G:\\Meine
Ablage\\01_Berufliches\\01_Projekte\\02_digitale_erinnerungslandschaft\\03_dat
a\\"
      # specify folder for output.
      results_path = working_dir_path + "\\ingest"
      self.logger.debug("Copying main image file from folder '%s' to '%s'",
                    data_path, results_path)

      state_abbr = self.pid_var
      if len(state_abbr) != 3:
          self.logger.error(
              "The state abbreviation in the pid should exactly have 3
characters. But got: '%s'", state_abbr)

      state_folder = ""
      if self.pid_var == "sty":
        state_folder = "steiermark" + os.path.sep + "steiermark_bilder"
      elif self.pid_var == "vor":
        state_folder = "vorarlberg" + os.path.sep + "vorarlberg_bilder"
      elif self.pid_var == "tir":
        state_folder = "tirol" + os.path.sep + "tirol_bilder"
      elif self.pid_var == "car":
        state_folder = "kaernten" + os.path.sep + "kaernten_bilder"
      elif self.pid_var == "bur":
        state_folder = "burgenland" + os.path.sep + "burgenland_bilder"
      elif self.pid_var == "sal":
        state_folder = "salzburg" + os.path.sep + "salzburg_bilder"
      elif self.pid_var == "vie":
        state_folder = "wien" + os.path.sep + "wien_bilder"
      else:
        raise ValueError(f"Not supported pid var {self.pid_var}")


      # loop through folder and copy to related
      source_path = data_path + state_folder
      results_path_files = os.listdir(results_path)

      # loop through built files and search for material in google drive
folders
      for fix_folder_name in results_path_files:
        # place_number = fix_folder_name[len(fix_folder_name)-1]
        place_number = re.findall('[0-9]+', fix_folder_name)[0]
        place_folder = state_folder + os.path.sep + str(place_number) +
os.path.sep + "fix"
        try:
          # first image
          source_img_path = data_path + place_folder + "\\1.JPG"
          target_img_path = results_path + os.path.sep + fix_folder_name +
"\\1.JPG"
          self.logger.debug(f"Trying to copy img from {source_img_path} to
{target_img_path}")
          copy(source_img_path, target_img_path)

          # second image
          source_img_path = data_path + place_folder + "\\2.JPG"
          target_img_path = results_path + os.path.sep + fix_folder_name +
"\\2.JPG"
          self.logger.debug(f"Trying to copy IMG.2 from {source_img_path} to
{target_img_path}")
          copy(source_img_path, target_img_path)
        except:
          pass
```

xml_build/archive_of_names/__main__.py

```python
# Author: Sebastian
#
from DERLAPersListHandler import DERLAPersListHandler
from DERLAKeyWordsHandler import DERLAKeyWordsHandler
from DERLATeiLoHandler import DERLATeiLoHandler
from DERLALoListHandler import DERLALoListHandler
from DERLADidacticGlossary import DERLADidacticsGlossary


def build_fixed_lo_xmls(env):
    """
    Method handles the building of educational related xml files.
    Individual TEI XML objects per educational material.

    Args:
        env (dict): Environment variables and parameters for the XML building
process.

    Returns:
        None
    """

    # sty fixed
    # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z83
AyjjRM/pub?gid=1790748251&single=true&output=csv"
    # env["pid_var"] = "sty"
    # derla_vor_lo_builder = DERLATeiLoHandler(env)

    # vor fixed (created by hand - skipped)

    # tir fixed
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc&gid=2
63259825"
    env["pid_var"] = "tir"
    derla_vor_lo_builder = DERLATeiLoHandler(env)
    # car fixed
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0&gid=7
40894826"
    env["pid_var"] = "car"
    derla_vor_lo_builder = DERLATeiLoHandler(env)


def build_lo_list(env):
    """
    Method handles construction of los as lists
    (for the fixierte vermittlungsangebote).

    Args:
        env (dict): The environment variables.

    Returns:
        None
    """

    # for styria
    # env["get_param"] =
"https://docs.google.com/spreadsheets/d/1NcZMgepoxPRi3X9By8qHvnvLdmoRwFKPl_z83
AyjjRM/pub?gid=1790748251&single=true&output=csv"
    # env["pid_var"] = "sty"
    # DERLALoListHandler(env)

    # vor list
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&id=1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLDFjGZnc&gid=2
63259825"
    env["pid_var"] = "tir"
    DERLALoListHandler(env)

    # tir list
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&id=13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51xLzFL0&gid=7
40894826"
    env["pid_var"] = "car"
    DERLALoListHandler(env)

def build_perslists(env):
    """
    Method to handle construction of perslist xmls needed in DERLA.

    Args:
        env (dict): The environment dictionary containing necessary
parameters.

    Returns:
        None
    """

    # vor persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zq
VqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639611&single=t
rue&output=csv"
    env["pid_var"] = "vor"
    DERLAPersListHandler(env)

    # sty persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/e/2PACX-1vTRFAMnUQ7YC9HZwwSrw55tXhopXc
XrGaVl2CMGd8JMRMYiZn7lmDG-pCU2SmDSuIR0-6vzgbTD29lS/pub?gid=1356804042&single=t
rue&output=csv"
    env["pid_var"] = "sty"
    DERLAPersListHandler(env)

    # tir persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1bI2DCh9XA6nkaKJgMSAD8w_orrnmJEjOGgDLD
FjGZnc/export?format=csv&gid=1986675455"
    env["pid_var"] = "tir"
    DERLAPersListHandler(env)

    # car persons
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/13W9Isf32jbjYaxBAtjDdZE3onVGcxAaZ6ns51
xLzFL0/export?format=csv&gid=1026060477"
    env["pid_var"] = "car"
    DERLAPersListHandler(env)


def build_keywords_tei(env):
    """
    Handles building of keywords TEI.

    Args:
        env (dict): The environment setup.

    Returns:
        None
    """
    # needed env setup
    env["get_param"] =
"https://docs.google.com/spreadsheets/d/1j3GeeoIwC72tlSToz8yrEgKo845gQ_yVMDymc
VhvNus/pub?gid=0&single=true&output=csv"
    env["pid_var"] = ""
    DERLAKeyWordsHandler(env)

def build_didactical_glossary(env):
    """
    Handles building of the didactical glossary for DERLA

    Args:
        env: The environment object used for building the glossary.

    Returns:
        None
    """

    DERLADidacticsGlossary(env)


if __name__ == "__main__":

    # Define env variables here as dictionary
    # with basic setting
    env = {
        "log_level": "DEBUG",
        "origin": "https://glossa.uni-graz.at",
        "pid_static": "o:derla.",
        "pid_var": "vor",
        "get_param":
"https://docs.google.com/spreadsheets/d/e/2PACX-1vRfam42JhbxFMs8u84im8twvZS5zq
VqpFHI6iyL_pnzfmO5WKDTefozICa8qngto4CB2PpdWYOxPBe0/pub?gid=2079639611&single=t
rue&output=csv"
    }

    # # instantiate your custom class here
    # derla_builder = DERLAPersListHandler(env)
    # build_perslists(env)

    # keywords tei
    # build_keywords_tei(env)

    # perslists
    # build_perslists(env)

    # build educational xml files
    # build_fixed_lo_xmls(env)

    #####
    # LO lists
    # build_lo_list(env)

    ## didactical glossary
    build_didactical_glossary(env)
```