

Exam paper Data for Heritage Collections: Exercise [F0CG6a] and Infrastructure for Digital Collections [F0CG7a] 2025 - Jenske Verhamme

date: 01/6/2025

Introduction

This is the exam paper for the course Data for Heritage Collections 2024-2025 (B-KUL-F0CG5A). This course is part of the postgraduate Cultureel Erfgoed: digitale track at KULeuven.

The assignment for this paper and project is to:

1. Request an API key from Europeana or Rijksmuseum
2. Determine a curation theme of your choice (check whether you can find sufficient data for this)
3. Download a selection of data of your choice
4. Clean this dataset using Openrefine
5. Try to enrich the data, e.g. by reconciliation
6. Create a visual presentation of this dataset (still to be determined: can be done via Excel, Tableau or a web page).
7. Create a report about your various steps, code used and the result. Submit the report and your dataset as an exam for the course Data for Heritage Collections.

Important note: All files mentioned in this paper are documented and downloadable on the Github page of this project!

link to project: <https://github.com/GuacamoleKoala/EuropeanaTeapots/>

Research Question(s)

What can we learn about teapots via the Europeana collection?

I decided to create a dataset on the topic of teapots in the collection of Europeana. I wanted to know what teapots are available and what we can get to know about them via the datamodels of Europeana.

An example of a teapot record on Europeana:



Record link: https://www.europeana.eu/en/item/90402/AK_NM_2627

The following metadata can be found on Europeana for this record.

Publisher: Rijksmuseum

Type of item:

urn:rijksmuseum:thesaurus:RM0001.THESAU.354
urn:rijksmuseum:thesaurus:RM0001.THESAU.32
urn:rijksmuseum:thesaurus:RM0001.THESAU.65746
urn:rijksmuseum:thesaurus:RM0001.THESAU.42776
urn:rijksmuseum:thesaurus:RM0001.THESAU.64940
urn:rijksmuseum:thesaurus:RM0001.THESAU.64863
urn:rijksmuseum:thesaurus:RM0001.THESAU.31312 Gilding

Providing institution: Rijksmuseum

Aggregator: Rijksmuseum

Rights statement for the media in this item (unless otherwise specified): <http://creativecommons.org/publicdomain/mark/1.0/> <http://creativecommons.org/publicdomain/mark/1.0/>

Rights: Public Domain Publiek Domein

Creation date: c.1750 - c.1774

Place-Time: third quarter 18th century

Places: China China

urn:rijksmuseum:thesaurus:RM0001.THESAU.403

Identifier: <http://hdl.handle.net/10934/RM0001.COLLECT.2995>
AK-NM-2627

Extent: height 10.7 cm diameter 4.7 cm diameter 11 cm
diameter 6 cm length 19.5 cm

Format: porcelain (material) glaze gold (metal) Vitreous
enamel Gold Porcelain

Language: nl

Is part of: collection: Asian ceramics collectie: Aziatische
keramiek collectie: China (collectie)

Providing country: Netherlands

Collection name: 90402_M_NL_Rijksmuseum

First time published on Europeana: 2014-05-27T13:04:36.401Z

Last time updated from providing institution: 2018-03-
17T13:01:00.155Z

Keywords (provided by the community): People's Republic of
China

Overview of the project

Here I give a quick overview of the different steps I took to work out the assignment. Important to note is that there is a manual available of all steps taken in the project. This manual is divided in four chapters. All these chapters are extensively elaborated in separate notebooks that can further be explored via the provided links. On Github the full project can also be found.

Link to manual:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/Manual>

overview of chapters:

1. API and Python (script)

link:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

2. OpenRefine (clean + enrichment)

link:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

3. Excel and Tableau (analysis and visualisation)

link:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

4. IIIF, HTML, Mirador and Github (iiif viewers)

link:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

Execution of the assignment

Here I give a general overview of how I approached the different steps of this assignment.

1. Request an API key from Europeana or Rijksmuseum

First of all I created an account on Europeana and requested my API key. Normally I have deleted my API key in the script, and it should be replaced with your own API key.

2. Determine a curation theme of your choice (check whether you can find sufficient data for this)

I choose to look for all records of teapots on Europeana. To do this I looked at queries in multiple languages and multiple spelling variations.

3. Download a selection of data of your choice

Here I created a Python script with the help of Gemini and ChatGPT in Google Colab. This script downloads a dataset of records regarding teapots from Europeana using an API key. The dataset consists of multiple metadata fields about the teapot records. For me it was important to get all records of teapots.

link to chapter one:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

Scripts



Below I share two scripts that were created to extract metadata about teapots on Europeana via an API key. The first script is the initial script to extract all teapot records on Europeana. The second script adds extra metadata fields through another strategy via an API key. The steps to achieve these lines of code are described in chapter one of the manual. The idea was to look for teapots in the Europeana collection by looking for all translations and spelling variants.

link to chapter one:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

The initial script looks is given below. It looks for 'europeana_id', 'All', 'country', 'dataProvider', 'dcDescription', 'dcDescriptionLangAware', 'dcLanguage', 'dcLanguageLangAware', 'dcSubjectLangAware', 'dcTitleLangAware', 'dcTypeLangAware', 'dctermsSpatial', 'edmConcept', 'edmConceptLabel', 'edmConceptPrefLabelLangAware', 'edmIsShownAt', 'edmIsShownBy', 'edmPlace', 'edmPlaceAltLabel', 'edmPlaceAltLabelLangAware', 'edmPlaceLabel', 'edmPlaceLabelLangAware', 'edmPlaceLatitude', 'edmPlaceLongitude', 'edmPreview', 'europeanaCollectionName', 'europeanaCompleteness', 'guid', 'id', 'index', 'language', 'link', 'organizations', 'provider', 'rights', 'title', 'type'.

```
In [ ]: # full working code FINAL
# generated by Gemini 2.0 on 15/04/2025
# authorized by Jenske Verhamme

# Import necessary libraries
import requests # Used for making HTTP requests
import csv # Used for writing data to CSV files
import os # Used for creating directories and handling file paths
import time # Used for adding delays between requests
from urllib.parse import quote_plus # Used for encoding special characters
from google.colab import files # Used for downloading files from Google Col
import json # Used for handling JSON data
import pandas as pd # Used for data manipulation and analysis (though not u
```

```

# Replace with your actual Europeana API key
API_KEY = "reeditaccif" # Your API key to access Europeana API

# Define a function to send requests with retries
def send_request(url, retries=5, backoff_factor=2):
    attempt = 0 # Initialize attempt counter
    while attempt < retries: # Loop for retries if request fails
        try:
            response = requests.get(url) # Send GET request
            response.raise_for_status() # Raise an error for bad HTTP status
            return response # If successful, return the response
        except requests.exceptions.RequestException as e: # Catch any exception
            print(f"Attempt {attempt + 1} failed: {e}") # Print error message
            attempt += 1 # Increment the attempt counter
            wait_time = backoff_factor ** attempt # Increase wait time exponentially
            print(f"Retrying in {wait_time} seconds...") # Inform the user
            time.sleep(wait_time) # Wait before retrying
    print("Max retries reached. Exiting...") # If max retries reached, print
    return None # Return None if all retries fail

# Function to extract date-related fields from an item's metadata
def extract_dates_from_search_hit(item):
    dates = {} # Initialize empty dictionary to store date-related fields
    for key, value in item.items(): # Loop over all fields in the item
        if "date" in key.lower() or "year" in key.lower() or "time" in key.lower():
            dates[key] = value # If the field name contains 'date', 'year', or 'time', add it to the dictionary
    return dates # Return the dictionary containing date-related fields

# Main function to fetch metadata from Europeana API based on search queries
def fetch_europeana_metadata_combined(api_key, queries, rows=100, limit=1000):
    directory = f'downloads/{int(time.time())}' # Create a directory based on current timestamp
    os.makedirs(directory, exist_ok=True) # Create the directory if it does not exist

    fetched_records = 0 # Initialize counter for fetched records
    cursor = '*' # Initialize cursor for pagination (used to get the next set of results)
    all_metadata = [] # List to store metadata of all fetched records
    csv_file_path = os.path.join(directory, 'metadata_combined_dates.csv')

    # List of fieldnames to be included in the CSV file
    fieldnames = [
        'europeana_id', 'All', 'country', 'dataProvider', 'dcDescription',
        'dcLanguage', 'dcLanguageLangAware', 'dcSubjectLangAware', 'dcTitleLangAware',
        'dcTypeLangAware', 'dctermsSpatial', 'edmConcept', 'edmConceptLabel',
        'edmConceptPrefLabelLangAware', 'edmIsShownAt', 'edmIsShownBy', 'edmPlaceAltLabel',
        'edmPlaceAltLabelLangAware', 'edmPlaceLabel', 'europeanaCompleteness',
        'europeanaCompleteness', 'guid', 'id', 'index', 'language', 'link',
        'provider', 'rights', 'title', 'type',
    ]
    dynamic_date_fields = set() # Set to store dynamic date-related fields

    # Fields from the search hit that we want to extract
    fields_from_search_hit = [
        'dcDescription', 'dcDescriptionLangAware', 'dcLanguage', 'dcLanguageLangAware',
        'dcSubjectLangAware', 'dcTitleLangAware', 'dcTypeLangAware', 'dctermsSpatial',
        'edmConcept', 'edmConceptLabel', 'edmConceptPrefLabelLangAware',
    ]

```

```

        'edmIsShownAt', 'edmIsShownBy', 'edmPlace', 'edmPlaceAltLabel',
        'edmPlaceAltLabelLangAware', 'edmPlaceLabel', 'edmPlaceLabelLangAware',
        'edmPlaceLatitude', 'edmPlaceLongitude'
    ]

# Estimate the total number of records available for the queries
combined_query = ' OR '.join(queries) # Combine the queries into a single string
search_url = f'https://api.europeana.eu/api/v2/search.json?wskey={api_key}'

try:
    search_response = send_request(search_url) # Send the request to Europeana API
    if search_response is None: # If the request fails, exit
        print("Error: Unable to fetch data.")
        return

    search_data = search_response.json() # Parse the response as JSON
    total_records = search_data.get('totalResults', 0) # Get the total number of records
    print(f"Estimated total records for query '{combined_query}': {total_records}")

except requests.exceptions.RequestException as e:
    print(f"Request error while estimating records: {e}")
    return

# Proceed with fetching the actual records
for i in range(0, len(queries), chunk_size):
    query_chunk = queries[i:i+chunk_size] # Select a chunk of queries
    combined_query = ' OR '.join(query_chunk) # Combine the queries into a single string
    print(f"Processing query chunk: {combined_query}") # Print the current query chunk

    # Fetch records in a loop until we reach the limit
    while fetched_records < limit:
        search_url = f'https://api.europeana.eu/api/v2/search.json?wskey={api_key}&query={combined_query}&size=100'

        try:
            search_response = send_request(search_url) # Send the request to Europeana API
            if search_response is None: # If the request fails, break the loop
                break

            search_data = search_response.json() # Parse the response as JSON
            if not search_data.get('success', True): # Check if the request was successful
                print(f"Error: {search_data.get('error', 'Unknown error')}")
                break

            if 'items' not in search_data: # Check if 'items' is in the response
                print("Error: 'items' key not found in response.") # Print an error message
                break

            # Loop through each item (record) in the search results
            for item in search_data['items']:
                record_id = item.get('id') # Get the unique record ID
                if record_id and fetched_records < limit: # Ensure we have enough records
                    search_dates = extract_dates_from_search_hit(item)
                    dynamic_date_fields.update(search_dates.keys()) # Add the dates to the dynamic fields

                    # Extract year reliably (from the 'year' field)
                    date = item.get("year", "No date available") # Extract the year from the item
                    dynamic_date_fields['year'].append(date)

                    fetched_records += 1 # Increment the number of fetched records
                    if fetched_records == limit:
                        break

        except requests.exceptions.RequestException as e:
            print(f"Request error while fetching records: {e}")
            break

    if fetched_records == limit:
        break

```

```

if date:
    search_dates['year'] = date # Add the 'year' field

# Initialize metadata dictionary with common fields
metadata = {
    'europeana_id': item.get('id', 'N/A'),
    'All': item.get('all', 'N/A'),
    'country': item.get('country', 'N/A'),
    'dataProvider': item.get('dataProvider', 'N/A'),
    'edmPreview': item.get('edmPreview', 'N/A'),
    'europeanaCollectionName': item.get('europeanaCollectionName', 'N/A'),
    'europeanaCompleteness': item.get('europeanaCompleteness', 'N/A'),
    'guid': item.get('guid', 'N/A'),
    'id': item.get('id', 'N/A'),
    'index': item.get('index', 'N/A'),
    'language': item.get('language', 'N/A'),
    'link': item.get('link', 'N/A'),
    'organizations': item.get('organizations', 'N/A'),
    'provider': item.get('provider', 'N/A'),
    **search_dates # Add the date-related fields to the metadata
}

# Extract other fields from the search result
for field in fields_from_search_hit:
    metadata[field] = item.get(field, 'N/A')

#  Extract title reliably
title = item.get("title") # Get the title from the item
if isinstance(title, list) and title: # If title is a list
    metadata['title'] = title[0]
elif isinstance(title, str): # If title is a string
    metadata['title'] = title
else: # If no title is found, set a default message
    metadata['title'] = "No title available"

#  Extract rights reliably
rights = item.get("rights") # Get the rights field
if isinstance(rights, list) and rights: # If rights is a list
    metadata['rights'] = rights[0]
elif isinstance(rights, str): # If rights is a string
    metadata['rights'] = rights
else: # If no rights are found, set a default message
    metadata['rights'] = "No rights available"

# Optional: still fetch full record to get 'type'
record_url = f'https://api.europeana.eu/api/v2/record/{item["id"]}'
record_response = send_request(record_url) # Send request to get the full record
if record_response: # If the full record is successful
    record_data = record_response.json() # Parse the JSON response
    obj = record_data.get('object', {}) # Extract the object part of the response
    metadata['type'] = obj.get('type', 'N/A') # Extract the type field from the object

all_metadata.append(metadata) # Add the metadata to the list
fetched_records += 1 # Increment the record count
print(f"Fetched record {fetched_records}/{limit}")
time.sleep(1) # Pause to avoid rate-limiting

```

```

        else: # If the full record couldn't be fetched
            print(f"Failed to fetch record: {record_id}") #

        if 'nextCursor' in search_data: # Check if there is a next
            cursor = search_data['nextCursor'] # Update the cursor
        else:
            break # If no nextCursor, end the loop

        time.sleep(1) # Sleep between requests to avoid rate-limiting

    except requests.exceptions.RequestException as e: # Catch request errors
        print(f"Request error: {e}") # Print the error message
        break # Exit the loop if there's a request error
    except Exception as e: # Catch unexpected errors
        print(f"An unexpected error occurred: {e}") # Print the error message
        break # Exit the loop if there's an unexpected error

updated_fieldnames = list(fieldnames) + list(dynamic_date_fields) # Concatenate fieldnames

with open(csv_file_path, mode='w', newline='', encoding='utf-8') as file:
    writer = csv.DictWriter(file, fieldnames=updated_fieldnames) # Initialize the writer
    writer.writeheader() # Write the header row
    writer.writerows(all_metadata) # Write the metadata rows

print(f"Process completed. {fetched_records} records fetched and metadata saved")
print(f"CSV file saved at: {csv_file_path}") # Print the location of the CSV file

if 'google.colab' in str(get_ipython()): # Check if running in Google Colab
    files.download(csv_file_path) # Download the CSV file to the user's local machine

# Input data
api_key = 'reeditaccif' # Define the API key for Europeana

# Define translations for 'teapot' in different languages
teapot_translations = {
    'en': ['teapot', 'tea?pot'],
    'nl': ['theepot', 'tee?pot'],
    'de': ['teekanne', '?eekanne'],
    'fr': ['théière', 'th?i?re', 'bouilloire'],
    'es': ['tetera'],
    'it': ['teiera'],
    'sl': ['čajnik'],
    'sv': ['teekanne'],
    'ca': ['tetera'],
    'pt': ['teiera', 'te?iera'],
    'ro': ['teiera'],
    'lt': ['teiera'],
    'uk': ['чайник'],
    'no': ['kettle'],
    'sr': ['čajnik', '?ajnik'],
    'fi': ['teekanne', 'te?kanne'],
    'da': ['teiere', 't?eiere'],
    'is': ['kettle'],
    'lv': ['teiera'],
    'pl': ['czajnik', 'cajnik'],
}

```

```
queries = [term for terms in teapot_translations.values() for term in terms]
unique_queries = list(set(queries)) # Remove duplicates from the queries
desired_limit = 10000 # Set the limit for how many records to fetch

# Start fetching metadata with the defined parameters
fetch_europeana_metadata_combined(api_key, unique_queries, limit=desired_limit)
```

Below is the additional script to extract more metadata. It extracts 'europeana_id', 'dcFormat', 'dctermsExtent', 'dcPublisher', 'dcRights' and 'dcSource'. Later I combined these two datasets in OpenRefine (see chapter two).

```

        response = requests.get(url, timeout=30) # Increased timeout
        response.raise_for_status() # Raise HTTPError for bad responses
        # print(f"Successfully fetched URL (Status: {response.status_code})")
        return response
    except requests.exceptions.RequestException as e:
        print(f"Attempt {attempt + 1} failed: {e}")
        attempt += 1
        wait_time = backoff_factor ** attempt
        print(f"Retrying in {wait_time} seconds...")
        time.sleep(wait_time)
    print(f"Max retries reached for URL: {url}. Skipping.")
    return None

# Function to safely extract values from potentially complex field structures
# Returns a single joined string for CSV compatibility
def get_field_values(data_dict, field_name, separator='; '):
    """
    Extracts all string values for a given field from a dictionary,
    handling nested structures like {'en': [...], 'def': [...]}.
    Strips leading/trailing whitespace from values.
    Returns a single string with values joined by the separator.
    """
    values = []
    field_content = data_dict.get(field_name) # Use .get for safety

    if field_content is None:
        return "" # Return empty string if field is not present

    if isinstance(field_content, dict):
        # Handles structures like {'en': [...], 'def': [...]}, {'def': [...]}
        for lang_key, val_list in field_content.items():
            if isinstance(val_list, list):
                extracted = [str(item).strip() for item in val_list if isinstance(item, str)]
                if extracted:
                    values.extend(extracted)
            elif isinstance(val_list, (str, int, float)): # Handle cases like '[]'
                extracted = str(val_list).strip()
                if extracted:
                    values.append(extracted)

    elif isinstance(field_content, list):
        # Handles simple lists of values like ['value1', 'value2']
        extracted = [str(item).strip() for item in field_content if isinstance(item, str)]
        if extracted:
            values.extend(extracted)

    elif isinstance(field_content, (str, int, float)):
        # Handles single string/numeric values
        extracted = str(field_content).strip()
        if extracted:
            values.append(extracted)

    # Remove duplicates and join
    unique_values = list(set(values))
    return separator.join(unique_values)

```

```

def fetch_europeana_specific_metadata(api_key, queries, target_fields, rows=100):
    """
    Fetches metadata from Europeana API based on search queries,
    extracts ONLY the specified target_fields from the 'rich' profile for each
    and saves the results to a CSV file.
    """
    if not api_key or api_key == 'your_api_key_here':
        print("Error: Please replace 'your_api_key_here' with your actual Europeana API key")
        print("You can request a key at: https://pro.europeana.eu/get-api")
        return

    directory = f'downloads/{int(time.time())}/'
    os.makedirs(directory, exist_ok=True)

    fetched_records = 0
    cursor = '*' # Initial cursor value for pagination
    all_metadata = []

    csv_file_path = os.path.join(directory, 'europeana_specific_metadata.csv')

    # Ensure europeana_id is always the first field if included
    final_fieldnames_order = ['europeana_id'] + [f for f in target_fields if f != 'europeana_id']

    for i in range(0, len(queries), chunk_size):
        query_chunk = queries[i:i+chunk_size]
        combined_query = ' OR '.join(query_chunk)
        print(f"\n--- Processing query chunk: {combined_query} ---")

        chunk_fetched_count = 0 # Counter for records fetched within this chunk

        # Loop through pages using cursor pagination
        while fetched_records < limit:
            print(f"Fetching records {fetched_records + 1} to {fetched_records + chunk_fetched_count}")

            # Request search results with rich profile to get basic item info
            search_url = f'https://api.europeana.eu/api/v2/search.json?wskey={api_key}&cursor={cursor}&size={chunk_size}&profile=rich'

            search_response = send_request(search_url)

            if search_response is None:
                print("Search request failed after retries. Moving to next chunk")
                break # Exit cursor loop for this chunk

            try:
                search_data = search_response.json()
            except json.JSONDecodeError:
                print("Error: Failed to decode JSON response for search.")
                print(f"Response text: {search_response.text}")
                break # Exit cursor loop

            if not search_data.get('success', True):
                print(f"Error: Search API reported failure. Response: {search_data}")
                break # Exit cursor loop

            items = search_data.get('items', [])
            if not items:

```

```

        print("No more items in this search response or chunk.")
        break # Exit cursor loop if no items returned

    print(f"Received {len(items)} items in this search page.")

    # Process each item from the search results
    for item in items:
        if fetched_records >= limit:
            print(f'Reached the overall limit of {limit} records.')
            break # Exit item loop if limit is reached

        record_id = item.get('id')
        if not record_id:
            print("Warning: Item found without 'id'. Skipping.")
            continue

        print(f"Processing record ID: {record_id} ({fetched_records})

        # Initialize dictionary for the current record's flattened metadata
        # Start with the ID, which comes from the search hit
        current_record_metadata = {'europeana_id': record_id}

        # --- Fetch the full record using the record ID and rich profile
        record_url = f'https://api.europeana.eu/record/v2/{record_id}'
        record_response = send_request(record_url)

        if record_response is None:
            print(f"Failed to fetch full record for ID: {record_id}.")
            # Add the partial metadata (just ID)
            # Ensure other target fields are present with empty strings
            for field in target_fields:
                if field != 'europeana_id' and field not in current_record_metadata:
                    current_record_metadata[field] = ''
            all_metadata.append(current_record_metadata)
            fetched_records += 1 # Count the search hit record
            time.sleep(0.5) # Small delay
            continue # Move to next item in search results

        try:
            record_data = record_response.json()
        except json.JSONDecodeError:
            print(f"Error: Failed to decode JSON response for record ID: {record_id}.")
            for field in target_fields:
                if field != 'europeana_id' and field not in current_record_metadata:
                    current_record_metadata[field] = ''
            all_metadata.append(current_record_metadata)
            fetched_records += 1
            time.sleep(0.5)
            continue

        if not record_data.get('success', True):
            print(f"Error: Record API reported failure for ID {record_id}.")
            for field in target_fields:
                if field != 'europeana_id' and field not in current_record_metadata:
                    current_record_metadata[field] = ''
            all_metadata.append(current_record_metadata)

```

```

        fetched_records += 1
        time.sleep(0.5)
        continue

    # --- Extract only the target fields from the record's 'object'
    obj_data = record_data.get('object', {})

    for field_name in target_fields:
        if field_name == 'europeana_id':
            continue # Already handled

        # Use a set to collect unique values for this field
        field_values_set = set()

        # Check top-level object fields
        top_level_value = obj_data.get(field_name)
        if top_level_value is not None:
            extracted_str = get_field_values(obj_data, field_name)
            if extracted_str:
                field_values_set.add(extracted_str)

        # Check proxies
        for proxy in obj_data.get('proxies', []):
            if isinstance(proxy, dict) and field_name in proxy:
                extracted_str = get_field_values(proxy, field_name)
                if extracted_str:
                    field_values_set.add(extracted_str)

        # Check aggregations
        for aggregation in obj_data.get('aggregations', []):
            if isinstance(aggregation, dict) and field_name in aggregation:
                extracted_str = get_field_values(aggregation, field_name)
                if extracted_str:
                    field_values_set.add(extracted_str)

        # Join all unique values found for this field
        current_record_metadata[field_name] = ';' .join(sorted(
            field_values_set))

    # --- Add the collected metadata for this record ---
    # Ensure all target fields are in the dict, even if empty
    for field in target_fields:
        if field not in current_record_metadata:
            current_record_metadata[field] = '' # Add with empty value

    all_metadata.append(current_record_metadata)

    fetched_records += 1
    chunk_fetched_count += 1 # Count within this chunk's cursor

    # Add a small delay between record fetches to be polite to the API
    time.sleep(0.5)

    # Check if the limit was reached during processing items
    if fetched_records >= limit:
        break # Exit cursor loop

```

```

# Check for the next cursor for pagination
next_cursor = search_data.get('nextCursor')
if next_cursor and next_cursor != cursor and chunk_fetched_count > 0:
    cursor = next_cursor
    print(f"Moving to next cursor: {cursor[:10]}...")
    time.sleep(1) # Longer delay between pages
else:
    print("No next cursor or no items fetched in this page. Ending loop")
    break # Exit cursor loop if no next cursor or no items fetched

# --- Writing data to CSV ---
print("\n--- Writing data to CSV ---")

print(f"Total records fetched: {fetched_records}")
print(f"Extracting {len(final_fieldnames_order)} specific fields.")

try:
    with open(csv_file_path, mode='w', newline='', encoding='utf-8') as file:
        writer = csv.DictWriter(file, fieldnames=final_fieldnames_order)
        writer.writeheader()
        # No need to fill missing keys here as we ensured all target fields exist
        # with '' if not found during processing
        writer.writerows(all_metadata)

    print(f"Process completed. Metadata saved to: {csv_file_path}")

    # Download the file in Google Colab if applicable
    if files:
        try:
            files.download(csv_file_path)
            print("CSV file downloaded.")
        except Exception as e:
            print(f"Error during Colab file download: {e}")

    except IOError as e:
        print(f"Error writing CSV file: {e}")
    except Exception as e:
        print(f"An unexpected error occurred during CSV writing: {e}")
        print(traceback.format_exc())

# --- Main execution part ---
if __name__ == "__main__":
    # --- Input Data and Configuration ---
    # Replace with your actual API key or ensure the global API_KEY variable is set
    # API_KEY = 'reeditaccif' # Ensure this matches the global variable

    teapot_translations = {
        'en': ['teapot'],
        'nl': ['theepot', 'tee?pot'],
        'de': ['teekanne', '*eekanne'],
        'fr': ['théière', 'th?i?re'],
        'es': ['tetera'],
        'it': ['teiera'],
        'sl': ['čajnik', '?ajnik'],
    }

```

```

'sv': ['teekanne', 'te??anne'],
'ca': ['tetera'],
'pt': ['teiera', 'te*iera'],
'ro': ['teiera'],
'lt': ['teiera'],
'uk': ['чайник'],
'no': ['kettle'], # Note: Norwegian might use 'kjelle' more commonly
'sr': ['čajnik', '?ajnik'],
'fi': ['teekanne', 'te?kanne'],
'da': ['teiere', 't?eiere'],
'is': ['kettle'],
'lv': ['teiera'],
'pl': ['czajnik', 'cajnik'],
}

queries = [term for terms in teapot_translations.values() for term in terms]
unique_queries = list(set(queries))

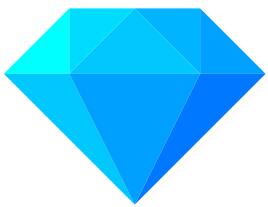
# Configuration for the fetch process
desired_limit = 10000 # Set the maximum number of records to fetch
search_rows = 100 # Number of records per search API call (max 100)
query_chunk_size = 5 # Number of queries to combine in one search request

# --- Run the data fetching process ---
if API_KEY == 'reeditaccif' or API_KEY == 'your_api_key_here':
    print("\n!!! WARNING: Please replace 'reeditaccif' or 'your_api_key_here' with your actual API key.\n")
    # You can uncomment the line below and replace with your key, or edit the variable above
    # API_KEY = 'YOUR_ACTUAL_EUROPEANA_API_KEY'
    # Proceed with the placeholder key for testing, but expect potential errors
    print("Attempting to run with the provided API key.")

# Pass the TARGET_FIELDS list to the function
fetch_europeana_specific_metadata(API_KEY, unique_queries, TARGET_FIELDS)

```

Data Refinement



OpenRefine

4. Clean this dataset using Openrefine

After combining these datasets, I started working on cleaning and refining them. To do this I used OpenRefine. I started by renaming and making copies of columns, tried to delete noise in records, to use regular expressions to refine. In

particular I worked out some methods to refine dates, sizes, concepts, and so on. Here I also used techniques as clustering, cross-reference, multi-valued cells, and so on.

link to chapter two:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

5. Try to enrich the data, e.g. by reconciliation

Next I also used some enrichment and reconciliation via Wikidata to match more cities and countries, or to match more materials in the list of concepts. All of these steps are also extensively annotated in chapter two.

link to chapter two:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

Datasets on Github

I have put several datasets online to see the process of cleaning and refinement in action. I added the initial dataset, the additional dataset, the cleaned dataset in OpenRefine, and the dataset after additional cleaning in Tableau.

link to datasets:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/Datasets>

Visualisations



6. Create a visual presentation of this dataset (still to be determined: can be done via Excel, Tableau or a web page).

First of all, I created several visualisations in Tableau. Here I will place some examples of visualisations, but the full list with more than 40 vizzes can be found on my account on Tableau Public. In particular I will show some examples of

sheets and some examples of dashboards here. The making of these vizzes is well documented in chapter three of the manual. Again, the interactive version can be found on Tableau Public. All information on the construction of these sheets and dashboards (and many more) in Tableau can be found in chapter three of the manual:

link to manual chapter three:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

link to project on Tableau Public:

https://public.tableau.com/views/VisualisationsEuropeanaTeapotsTableau/TimelineOfUS&:sid=&:redirect=auth&:display_count=n&:origin=viz_share_link

The Tableau project as workbook can also be found on Github:

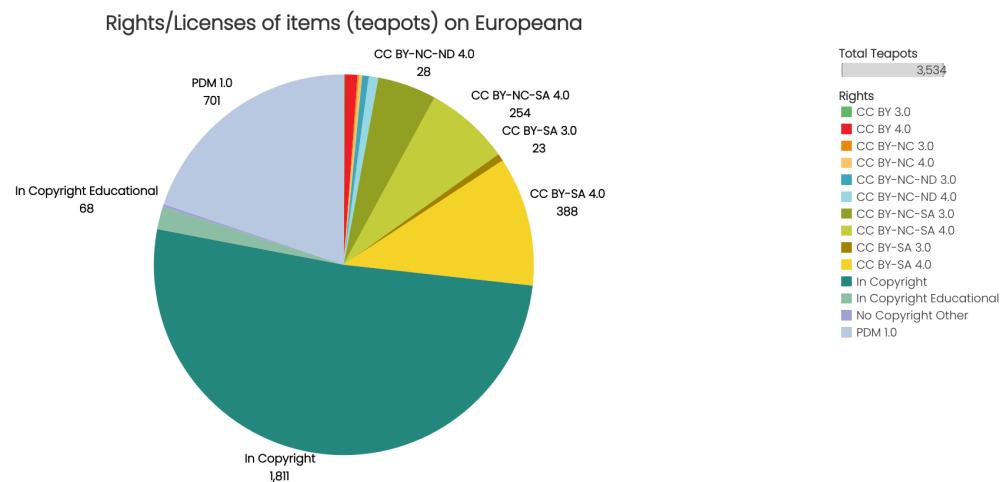
<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Visualisations/Visu>

I also created a pdf of all my visualisations, which can be found here:

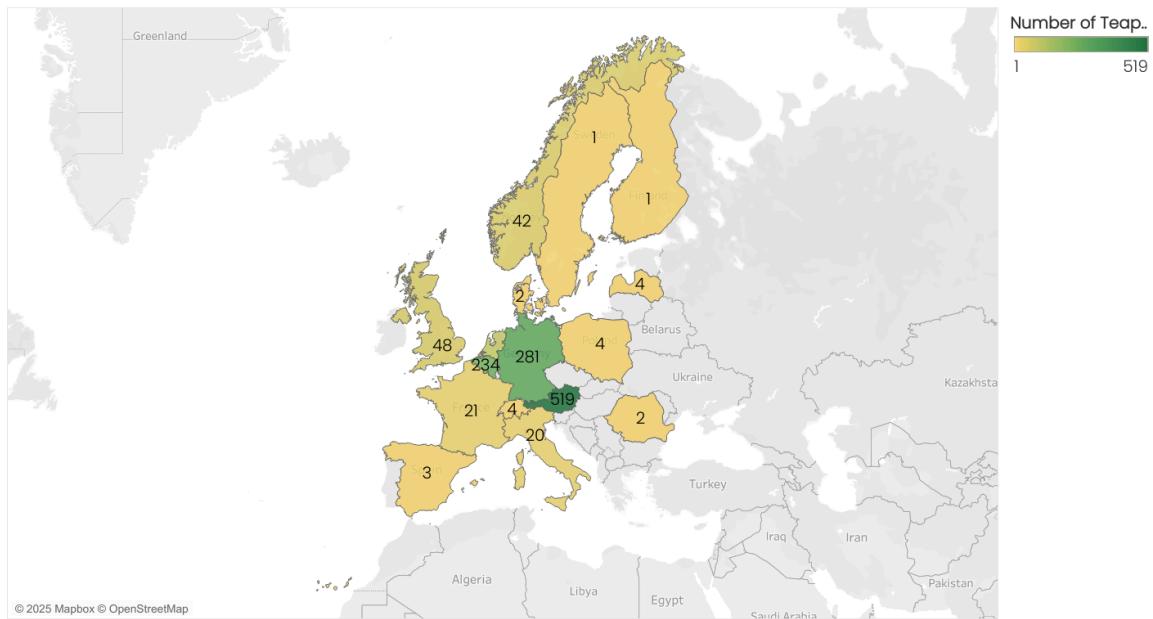
<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Visualisations/Visu>

Vizzes (examples)

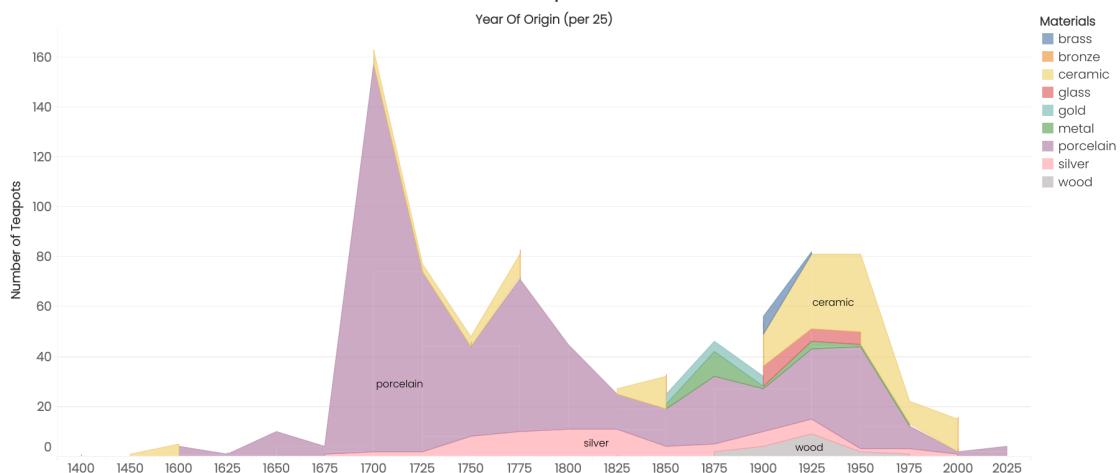
A. Sheets



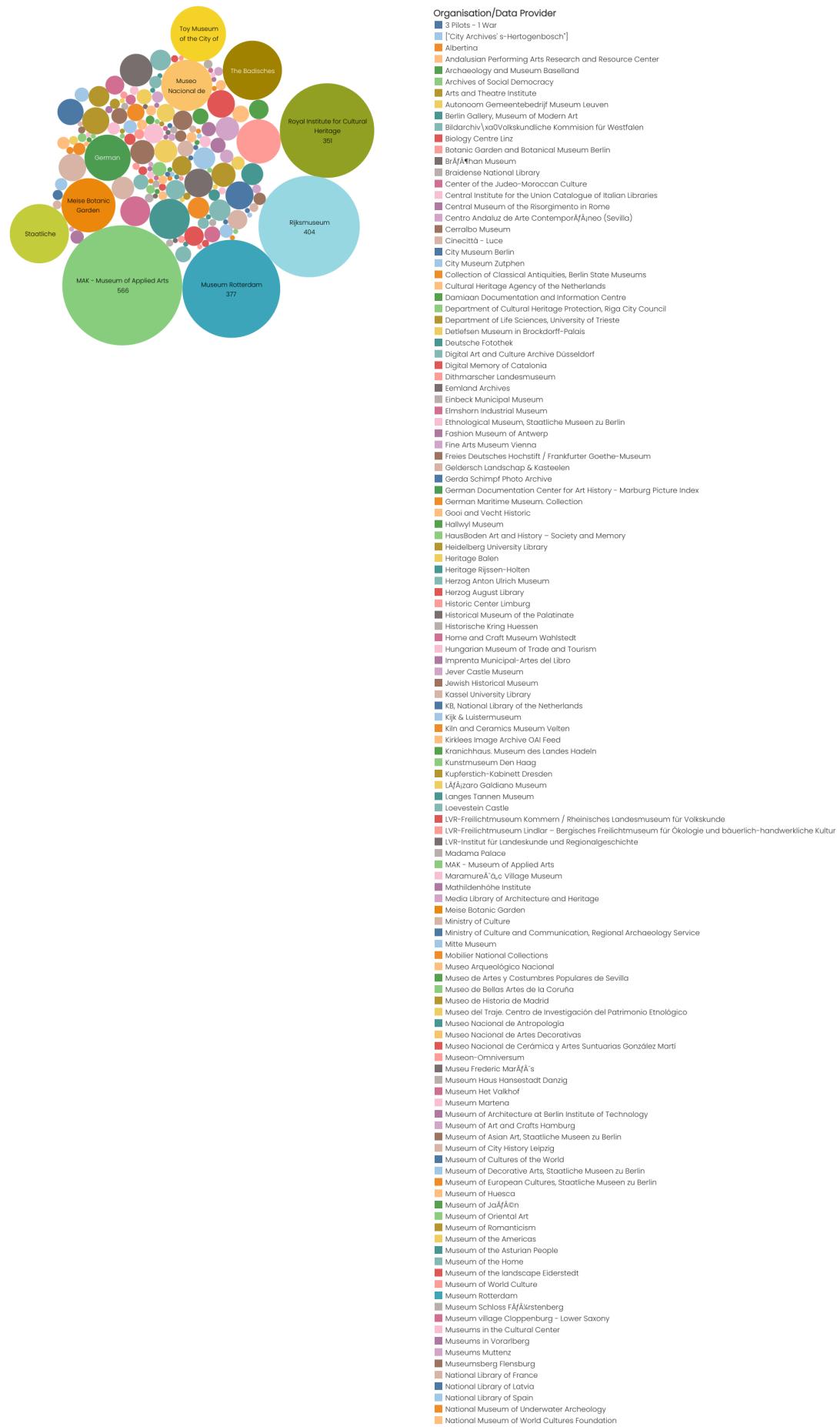
Location of teapots in dataset Europe (Countries)



Materials used for Teapots over Time

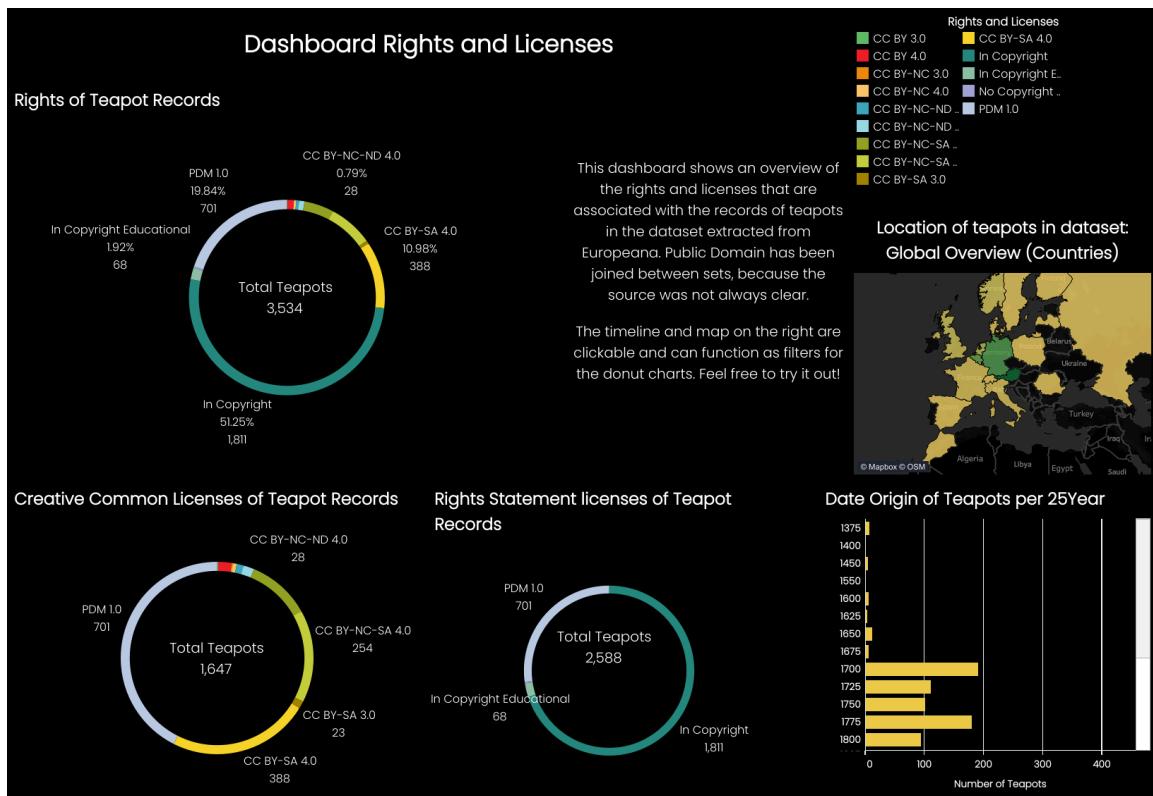


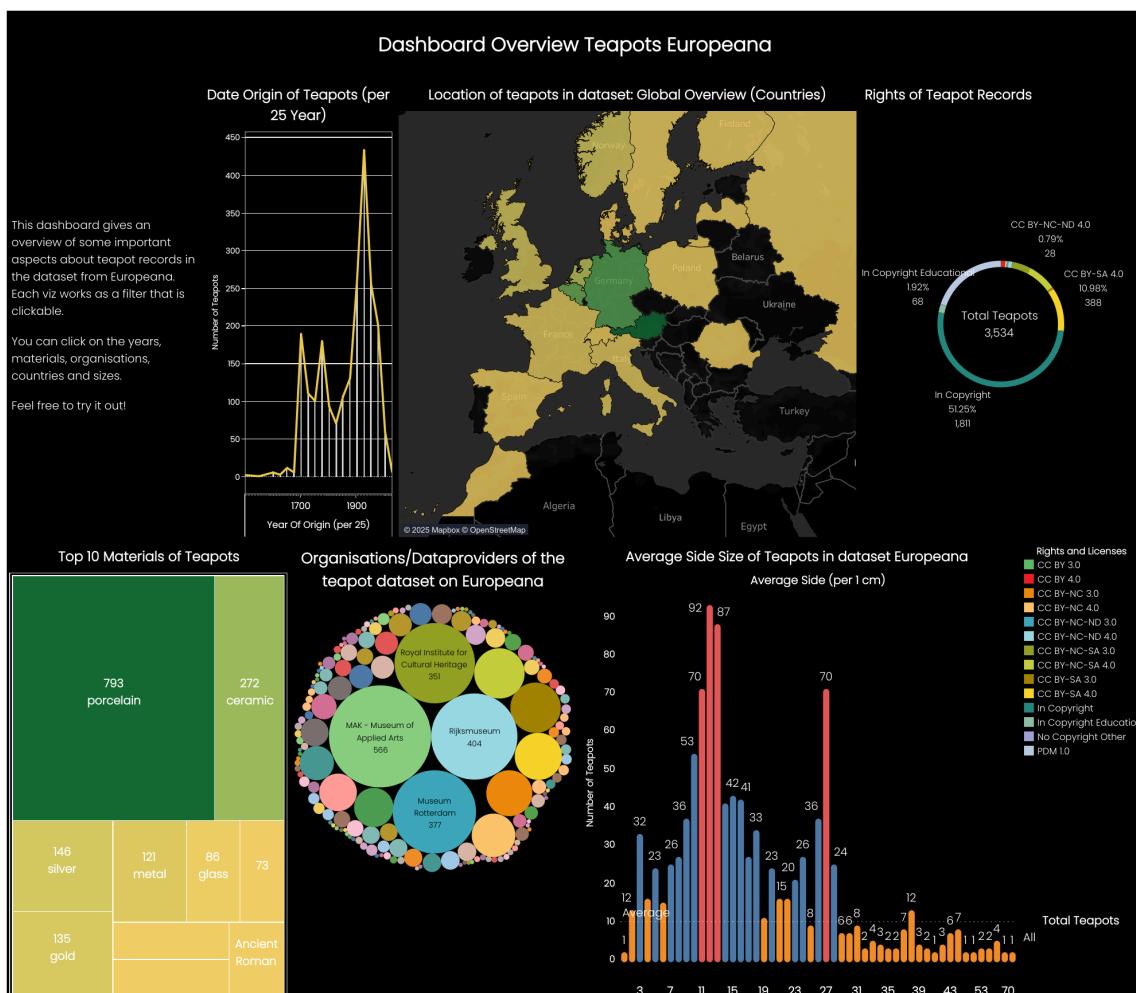
Organisations/Dataproviders of the teapot dataset on Europeana



- Netherlandas Institute for Sound & Vision
- Oderberg Inland Navigation Museum
- Old Frisian House
- Palais Galliera
- Public Center for Mental Health Rekem
- Ranieri di Sorbello Foundation
- Rijksmuseum
- Rijksmuseum van Oudheden
- Royal Institute for Cultural Heritage
- Royal Botanic Garden Edinburgh
- Schola Graphica Art Collection, Budapest
- Sheffield Images
- Soriaa Museum
- Staatliche Kunstsammlungen Dresden
- State Archives of Baden-Württemberg
- State Association of Fine Arts Saxony eV
- Stiftung Domäne Dahlem - Estate and Museum
- Stormarn Village Museum
- Technoseum
- The Badisches Landesmuseum
- The City Museum
- The Government of Catalonia
- The International Center for Information Management Systems and Services
- The National Library of Poland
- The province of North Holland
- The Trustees of the Natural History Museum, London
- The Utrecht Archives
- Toy Museum of the City of Nuremberg
- Turin Gallery for Modern and Contemporary Art
- University of Graz
- University of Vienna
- Valencian Digital Library
- Victoria and Albert Museum
- VILLA NORTH – Space for contemporary art, culture and interdisciplinary research
- Vorarlberg State Library
- Werkbundarchiv –[a-cos](#) museum of things
- World Museum Vienna
- Worpsweder Käseglöcke
- Zuiderzeemuseum

B. Dashboards





III-F-viewer



Last but not least, I wanted to be able to visualize the images of the teapot dataset. My first plan was to make a script that downloads all the thumbnails in a folder.

Thumbnails

To visualize the collection of thumbnails, I started by writing a script that downloads all the thumbnails in a separate folder via the urls in our dataset. I uploaded the folder per 1000 thumbnails on Github, because of file size limits. This way I can scroll easily through the images connected to the teapot records and delete what is not a teapot.

Below you can upload the extracted dataset from Tableau. Next I added the script to download the thumbnails.

```
In [ ]: from google.colab import files  
uploaded = files.upload()
```

```
In [ ]: #script to download all thumbnails of uploaded file into a zip  
# authorized by Jenske Verhamme  
#written with help of Gemini and ChatGPT  
  
import pandas as pd  
import requests  
import os  
from urllib.parse import urlparse  
import hashlib # For creating a hash from the URL to ensure uniqueness  
import time # For adding delays if needed  
import re # For sanitizing filenames  
import zipfile # For creating zip archives  
from google.colab import files # Import for Colab-specific file operations  
  
def download_images_from_dataframe(dataframe_path, image_url_column, output_  
    """  
        Downloads images from URLs specified in a DataFrame column, renames them  
    """
```

```
as 'image_X.ext' upon successful download, and creates a log file mapping original URLs/records to final downloaded filenames.
```

Args:

```
    dataframe_path (str): The path to your dataset file (e.g., 'teapots.csv').  
    image_url_column (str): The name of the column containing image URLs.  
    output_folder (str): The name of the folder where images will be saved.  
                        Defaults to 'downloaded_images'.  
    """
```

```
# --- 1. Create Output Folder ---  
if not os.path.exists(output_folder):  
    os.makedirs(output_folder)  
    print(f"Created output folder: '{output_folder}'")  
else:  
    print(f"Output folder '{output_folder}' already exists.")  
  
# --- 2. Load the Dataset ---  
try:  
    # Added dtype specification as suggested by the DtypeWarning from pandas.  
    df = pd.read_csv(dataframe_path, dtype={'Column19': str, 'Column24': str})  
    print(f"Successfully loaded dataset from: '{dataframe_path}'")  
except FileNotFoundError:  
    print(f"Error: Dataset file not found at '{dataframe_path}'. Please check the path and try again.")  
    return  
except Exception as e:  
    print(f"Error loading dataset: {e}")  
    return  
  
# --- 3. Initialize Log Data and Counters ---  
download_log = []  
download_count = 0 # Counts successfully downloaded files  
skipped_count = 0  
error_count = 0  
image_sequential_counter = 0 # Counter for "image_X" naming  
  
if image_url_column not in df.columns:  
    print(f"Error: Column '{image_url_column}' not found in the dataset.")  
    print(f"Available columns are: {df.columns.tolist()}")  
    return  
  
print("Starting image download process...")  
for index, row in df.iterrows():  
    image_url = row[image_url_column]  
    generated_filename = None # Will store the filename used for download  
    status = "Processing"  
    error_message = None  
    ext = '.jpg' # Default extension, will be updated  
  
if pd.isna(image_url) or not isinstance(image_url, str) or not image_url:  
    status = "Skipped (Invalid URL)"  
    skipped_count += 1  
    error_message = "Empty or invalid URL found."  
    download_log.append({  
        'dataframe_index': index,  
        'original_url': image_url,
```

```

        'generated_filename': None, # No filename generated
        'download_status': status,
        'error_message': error_message
    })
    continue

try:
    # --- Initial Filename Generation (using your existing logic) ---
    parsed_url = urlparse(image_url)
    path_segments = [s for s in parsed_url.path.split('/') if s]

    suggested_name = ""
    if path_segments:
        if path_segments[-1].lower() == 'manifest' and len(path_segments) > 1:
            suggested_name = path_segments[-2]
        else:
            suggested_name = path_segments[-1]

    suggested_name = re.sub(r'^[\w\-\.\.]+', '', suggested_name).strip()
    url_short_hash = hashlib.md5(image_url.encode('utf-8')).hexdigest()

    if '.' in parsed_url.path:
        potential_ext = os.path.splitext(parsed_url.path)[1].lower()
        if potential_ext in ['.jpg', '.jpeg', '.png', '.gif', '.bmp']:
            ext = potential_ext

    # This is the initial filename for download
    if suggested_name:
        initial_download_filename = f"{suggested_name}_{url_short_hash}"
    else:
        initial_download_filename = f"{url_short_hash}{ext}"

    if len(initial_download_filename) > 200:
        initial_download_filename = f"{url_short_hash}{ext}"

    generated_filename = initial_download_filename # Store for logging
    image_path = os.path.join(output_folder, initial_download_filename)

    if os.path.exists(image_path):
        status = "Skipped (Exists)"
        skipped_count += 1
        # Log uses the existing 'generated_filename' (initial_download_filename)
        download_log.append({
            'dataframe_index': index,
            'original_url': image_url,
            'generated_filename': generated_filename,
            'download_status': status,
            'error_message': None
        })
        continue

    print(f"Downloading image for URL index {index}: '{image_url}'")
    response = requests.get(image_url, stream=True, timeout=15)
    response.raise_for_status()

    with open(image_path, 'wb') as f:

```

```

        for chunk in response.iter_content(chunk_size=8192):
            f.write(chunk)

# --- Successful Download: Rename to image_X format ---
image_sequential_counter += 1
final_image_name = f'image_{image_sequential_counter}{ext}' # Change ext to .jpg or .png
final_image_full_path = os.path.join(output_folder, final_image_name)
original_downloaded_path = image_path # Path with initial download

try:
    os.rename(original_downloaded_path, final_image_full_path)
    print(f"Successfully downloaded. Renamed '{initial_download_filename}' to '{final_image_name}'")
    generated_filename = final_image_name # Update for logging
except OSError as e_rename:
    print(f"Error renaming '{initial_download_filename}' to '{final_image_name}'")
    # generated_filename remains initial_download_filename
    if error_message:
        error_message += f"; Rename to {final_image_name} failed"
    else:
        error_message = f"Rename to {final_image_name} failed: {e_rename}"
status = "Downloaded"
download_count += 1
time.sleep(0.1)

except requests.exceptions.RequestException as req_err:
    status = "Error"
    error_count += 1
    print(f"Error downloading '{image_url}': {req_err}")
    error_message = str(req_err)
except Exception as e:
    status = "Error"
    error_count += 1
    print(f"An unexpected error occurred for '{image_url}': {e}")
    error_message = str(e)

download_log.append({
    'dataframe_index': index,
    'original_url': image_url,
    'generated_filename': generated_filename, # This will be image_X
    'download_status': status,
    'error_message': error_message
})

print("\n--- Download Summary ---")
print(f"Total images downloaded and named/renamed: {download_count}")
print(f"Total images skipped (due to existing files or invalid URLs): {skipped_count}")
print(f"Total images failed (due to download errors): {error_count}")
print(f"Images saved in: '{os.path.abspath(output_folder)}'")

# --- 4. Save the Download Log to CSV ---
log_df = pd.DataFrame(download_log)
log_filepath = os.path.join(output_folder, "image_download_log.csv")
try:
    log_df.to_csv(log_filepath, index=False, encoding='utf-8')

```

```

        print(f"\nDownload log saved to: '{log_filepath}'")
    except Exception as e:
        print(f"Error saving download log: {e}")

    return output_folder

def zip_folder(folder_path, zip_name):
    """
    Compresses a specified folder into a zip archive.
    """
    try:
        with zipfile.ZipFile(zip_name, 'w', zipfile.ZIP_DEFLATED) as zipf:
            for root, dirs, files_in_dir in os.walk(folder_path):
                for file_item in files_in_dir:
                    file_path = os.path.join(root, file_item)
                    zipf.write(file_path, os.path.relpath(file_path, folder_
        print(f"\nSuccessfully created zip archive: '{os.path.abspath(zip_na
    except Exception as e:
        print(f"Error creating zip archive for '{folder_path}': {e}")

# --- Script Execution ---
if __name__ == "__main__":
    # --- Configuration ---
    # IMPORTANT: Ensure your file is accessible (upload to Colab or mount Google Drive)
    your_dataframe_file = 'DatasetCleanedEuropeanaTeapotsExcel.csv' # <--- CHANGE THIS TO YOUR DATAFILE
    your_image_url_column_name = 'ImagePreview' # <--- CHANGE THIS TO YOUR IMAGE URL COLUMN NAME
    your_output_folder_name = 'downloaded_sequential_images' # Changed folder name
    your_zip_file_name = f"{your_output_folder_name}.zip"

    # 1. Run the download process
    downloaded_folder = download_images_from_dataframe(
        dataframe_path=your_dataframe_file,
        image_url_column=your_image_url_column_name,
        output_folder=your_output_folder_name
    )

    # 2. After downloading, zip the folder
    if downloaded_folder:
        zip_folder(downloaded_folder, your_zip_file_name)

    # 3. Trigger the download of the created zip file in Colab
    try:
        print(f"\nAttempting to download '{your_zip_file_name}' (Colab specific function)")
        files.download(your_zip_file_name)
    except NameError: # Handles if 'files' from google.colab is not defined
        print(f"\n'files.download()' is a Colab specific function. If not running in Colab, ignore this message")
    except Exception as e:
        print(f"Error during Colab files.download: {e}")

```

IIIF manifests and IIIF collection

Next I wanted to visualize the images in high quality via the IIIF manifests of the records. Therefore I collected all IIIF manifest urls and combined them into one

url, namely the IIIF collection manifest url. This url will be used to upload all records with only one url link (instead of 3500+) into Mirador.

Below is the script to download all IIIF url's.

```
In [ ]: # Script to fetch Europeana ID and IIIF URL with Record API validation
# generated by Gemini 2.0 on 15/04/2025
# authorized by Jenske Verhamme
# Validation step re-added on 02/05/2025

# Import necessary libraries
import requests # Used for making HTTP requests
import csv # Used for writing data to CSV files
import os # Used for creating directories and handling file paths
import time # Used for adding delays between requests
from urllib.parse import quote_plus # Used for encoding special characters
try:
    from google.colab import files # Used for downloading files from Google
    IN_COLAB = True
except ImportError:
    IN_COLAB = False # Not running in Colab
import json # Used for handling JSON data

# <<< --- PASTE YOUR ACTUAL EUROPEANA API KEY HERE --- >>>
API_KEY = "reeditaccif"

# Define a function to send requests with retries (same as before)
def send_request(url, retries=3, backoff_factor=1.5): # Reduced retries slightly
    attempt = 0
    response = None # Initialize response outside the try block
    while attempt < retries:
        try:
            # Add a timeout to the request to prevent hanging indefinitely
            response = requests.get(url, timeout=15) # 15 second timeout
            response.raise_for_status()
            # Check for explicit API error messages even on 200 OK for Records API
            if "/record/" in url: # Only apply this check to record API calls
                try:
                    data = response.json()
                    if not data.get('success', True):
                        print(f"Record API returned error: {data.get('error')}")
                        return None # Treat API error as failure
                except json.JSONDecodeError:
                    print(f"Record API response not valid JSON: {response.text}")
                    return None # Treat invalid JSON as failure
                return response
        except requests.exceptions.Timeout:
            print(f"Attempt {attempt + 1} timed out after 15 seconds.")
            attempt += 1
        except requests.exceptions.RequestException as e:
            # Be less verbose for 404 errors during validation as they are expected
            if isinstance(e, requests.exceptions.HTTPError) and e.response.status_code == 404:
                # print(f"Validation failed: Record not found (404) for {url}")
                return None # Treat 404 as validation failure immediately
        else:
```

```

        print(f"Attempt {attempt + 1} failed: {e}")
        attempt += 1

    if attempt < retries and response is None: # Check if response is still None
        wait_time = backoff_factor ** attempt
        # print(f"Retrying in {wait_time:.2f} seconds...") # Less verbose
        time.sleep(wait_time)

# print("Max retries reached or validation failed.") # Less verbose fail
return response

# Main function to fetch IDs and construct IIIF Manifest URLs with validation
def fetch_europeana_manifests_validated(api_key, queries, rows=100, limit=1000):
    directory = f'downloads_manifests_validated/{int(time.time())}' # New folder
    os.makedirs(directory, exist_ok=True)

    validated_records_count = 0 # Count records that pass validation
    processed_search_results = 0 # Count total search results processed
    cursor = '*'
    all_manifest_data = [] # List to store dicts of {'europeana_id': id, 'iiif_manifest_url': url}
    csv_file_path = os.path.join(directory, 'europeana_manifest_urls_validated.csv')

    fieldnames = ['europeana_id', 'iiif_manifest_url']

    combined_query = ' OR '.join(queries)

    # Estimate total records (optional) - based on search index, not validation
    estimated_total_records = None
    search_url_estimate = f'https://api.europeana.eu/api/v2/search.json?wskey={api_key}&query={combined_query}&rows={rows}&limit={limit}&cursor={cursor}'
    estimate_response = send_request(search_url_estimate, retries=2) # Lower limit
    if estimate_response:
        try:
            estimate_data = estimate_response.json()
            estimated_total_records = estimate_data.get('totalResults')
            print(f'Estimated total search results for "{combined_query}": {estimated_total_records}')
        except json.JSONDecodeError:
            print("Could not estimate total results.")

    print(f"Starting fetch for query: {combined_query}. Aiming for up to {limit} validated records")

    # Fetch records loop - continues until limit of *validated* records is reached
    while validated_records_count < limit:
        # Fetch a batch of search results
        search_url = f'https://api.europeana.eu/api/v2/search.json?wskey={api_key}&query={combined_query}&rows={rows}&limit={limit}&cursor={cursor}'
        print(f"\nFetching search batch using cursor: {cursor}")

        search_response = send_request(search_url, retries=5, backoff_factor=2)
        if search_response is None:
            print("Failed to fetch search batch, stopping.")
            break

        try:
            search_data = search_response.json()
        except json.JSONDecodeError as e:
            print(f"JSON decode error for search batch: {e}")

        for record in search_data['records']:
            europeana_id = record['id']
            iiif_manifest_url = record['iiif_manifest_url']
            all_manifest_data.append({'europeana_id': europeana_id, 'iiif_manifest_url': iiif_manifest_url})
            validated_records_count += 1
            if validated_records_count == limit:
                break
    return all_manifest_data

```

```

        print(f"Response text: {search_response.text[:200]}")
        break # Stop if search results are invalid

    if not search_data.get('success', True):
        print(f"Search API Error: {search_data.get('error', 'Unknown error')}")
        break

    items = search_data.get('items')
    if not items:
        print("No more items found in search results.")
        break

    print(f"Processing {len(items)} items from search batch...")
    items_validated_in_batch = 0

    # Loop through each item found in the search batch
    for item in items:
        if validated_records_count >= limit:
            print("Target limit reached.")
            break # Stop processing this batch if limit hit

        processed_search_results += 1
        record_id = item.get('id')

        if record_id and isinstance(record_id, str):
            cleaned_record_id_for_url = record_id.lstrip('/')

            # --- !!! Add Validation Step !!! ---
            record_url = f'https://api.europeana.eu/api/v2/record/{cleaned_record_id_for_url}'
            record_response = send_request(record_url) # Uses lower retr

            # --- Only proceed if the Record API call was successful ---
            if record_response:
                # Construct IIIF Manifest URL
                iiif_manifest_url = f"https://iiif.europeana.eu/presentat

                manifest_data = {
                    'europeana_id': record_id,
                    'iiif_manifest_url': iiif_manifest_url
                }
                all_manifest_data.append(manifest_data)
                validated_records_count += 1 # Increment count only for
                items_validated_in_batch += 1

                # Print progress periodically
                if validated_records_count % 50 == 0 or validated_records_count >=
                    print(f"Validated record {validated_records_count}/{len(items)}")

            # else: # Optional: Log skipped items
            #     # print(f"Skipped item {record_id} - Failed Record API
            #     #       pass

            # --- Add a small delay within the loop to avoid overwhelming
            # Adjust sleep time as needed. Start small. Remove if not needed.
            time.sleep(0.05) # 50 milliseconds delay between record checks

```

```

    else:
        print(f"Skipping item due to missing/invalid ID in search re

# --- End of loop for items in batch ---
print(f"Validated {items_validated_in_batch} items in this batch.")

if validated_records_count >= limit:
    print("Target limit reached after processing batch.")
    break # Exit outer loop if limit hit

if 'nextCursor' in search_data:
    cursor = search_data['nextCursor']
else:
    print("No nextCursor found in search results. Assuming end.")
    break # Exit outer loop if no more search results

# Optional: slightly longer sleep between batches
# time.sleep(0.5)

# --- End of while validated_records_count < limit loop ---

print(f"\nWriting {len(all_manifest_data)} validated ID/Manifest URL pa
print(f"(Processed {processed_search_results} total search results to fi

try:
    with open(csv_file_path, mode='w', newline='', encoding='utf-8') as
        writer = csv.DictWriter(file, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(all_manifest_data)

    print(f"Process completed. {len(all_manifest_data)} validated records
    print(f"CSV file saved at: {csv_file_path}")

    if IN_COLAB:
        print("Attempting to download file in Colab...")
        files.download(csv_file_path)
    else:
        print(f"File saved locally at: {csv_file_path}")

except IOError as e:
    print(f"Error writing CSV file: {e}")
except Exception as e:
    print(f"An unexpected error occurred during CSV writing: {e}")

return all_manifest_data # Return the list of validated data

# --- Define search terms (same as before) ---
teapot_translations = {
    'en': ['teapot', 'tea?pot'],
    'nl': ['theepot', 'tee?pot'],
    'de': ['teekanne', '?eekanne'],
    'fr': ['théière', 'th?i?re', 'bouilloire'],
    'es': ['tetera'],
    'it': ['teiera'],
    'sl': ['čajnik'],
}

```

```

'sv': ['teekanne'],
'ca': ['tetera'],
'pt': ['teiera', 'te?iera'],
'ro': ['teiera'],
'lt': ['teiera'],
'uk': ['чайник'],
'no': ['kettle'],
'sr': ['čajnik', '?ajnik'],
'fi': ['teekanne', 'te?kanne'],
'da': ['teiere', 't?eiere'],
'is': ['kettle'],
'lv': ['teiera'],
'pl': ['czajnik', 'cajnik'],
}
queries = list(set([term for terms in teapot_translations.values() for term
print(f"Using {len(queries)} unique search terms.")

# --- Set desired limit for VALIDATED records ---
# This should now result in a count closer to your original ~4500
desired_limit = 15000 # Aim for up to 15000 validated records

# --- Run the validated function and store the result ---
if API_KEY == "YourapiKey":
    print("\n--- WARNING: Please replace 'reditaccif' with your actual Euro
else:
    validated_data = fetch_europeana_manifests_validated(API_KEY, queries, l

# --- Print statement of estimated results ---
# The estimated total search results are printed within the `fetch_europeana
# Now we can access the returned `validated_data` list.
print(f"\n--- Estimated and Final Results ---")
# The estimated total number of records based on the initial search query is
# The final number of validated records is now accessed from the returned li
if 'estimated_total_records' in locals() and estimated_total_records is not
    print(f"Estimated total matching records (before validation): {estimated_
else:
    print("Estimated total matching records (before validation): Could not b

# The actual number of validated records is the length of the returned list.
if 'validated_data' in locals():
    print(f"Number of validated records with IIIF Manifest URLs: {len(validat
else:
    print("Number of validated records with IIIF Manifest URLs: Could not be

```

To create the IIIF collection manifest url I have to restructure all IIIF manifest urls of the records into a proper JSON file. Below is the script to do that. Unfortunately, I was not able to let the urls be read from a file, so I had to paste them here manually.

First I saved the list of manifest urls into a txt file and then used that file to create a collection of urls in json format.

link to list of IIIF urls:

https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/IIIFViewer/iiif_man

```
In [ ]: from google.colab import files  
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving iiif_manifest_urls.txt to iiif_manifest_urls.txt

```
In [ ]: #code to change iiif manifest urls to a json that can serve as a iiif collection  
import json  
import requests # Required only for fetching labels automatically  
import time      # Used for a small delay when fetching  
  
# --- Configuration ---  
COLLECTION_ID = "https://your-username.github.io/my-iiif-data/collection.json"  
COLLECTION_LABEL = "My Awesome IIIF Collection" # Replace with your Collection Label  
OUTPUT_FILENAME = "my-collection.json" # Name of the output file  
FETCH_LABELS = True # Set to False if you don't want to fetch labels automatically  
REQUEST_TIMEOUT = 10 # Seconds to wait for each manifest request  
REQUEST_DELAY = 0.1 # Seconds to wait between requests (politeness)  
  
# List of manifest URLs (replace with your actual URLs)  
# Option 1: Define list directly in the script  
manifest_urls = [  
  
]  
# Option 2: Read from a file (uncomment the following lines if using this)  
manifest_urls = []  
try:  
    with open("iiif_manifest_urls.txt", "r") as f:  
        manifest_urls = [line.strip() for line in f if line.strip()]  
except FileNotFoundError:  
    print("Error: iiif_manifest_urls.txt not found.")  
    exit()  
  
# --- Script Logic ---  
collection = {  
    "@context": "http://iiif.io/api/presentation/3/context.json",  
    "id": COLLECTION_ID,  
    "type": "Collection",  
    "label": {"en": [COLLECTION_LABEL]}, # Assuming English label  
    "items": []  
}  
  
print(f"Processing {len(manifest_urls)} manifests...")  
  
for index, url in enumerate(manifest_urls):  
    print(f"[{index+1}/{len(manifest_urls)}] Processing: {url}")  
    item = {  
        "id": url,  
        "type": "Manifest"  
        # Label will be added below if fetched  
    }  
  
    if FETCH_LABELS:
```

```

manifest_label = f"Manifest {index+1}" # Default label
lang_code = "en" # Default language

try:
    # Add headers to potentially look more like a browser
    headers = {'User-Agent': 'Python IIIF Collection Script/1.0'}
    response = requests.get(url, timeout=REQUEST_TIMEOUT, headers=headers)
    response.raise_for_status() # Raise an error for bad status code
    manifest_data = response.json()

    # Try to get label (works for V3 and most V2)
    if 'label' in manifest_data and manifest_data['label']:
        label_obj = manifest_data['label']
        # Handle V3 structure (language map)
        if isinstance(label_obj, dict):
            # Try common language codes first
            if 'en' in label_obj and label_obj['en'][0]:
                manifest_label = label_obj['en'][0]
                lang_code = 'en'
            elif 'none' in label_obj and label_obj['none'][0]:
                manifest_label = label_obj['none'][0]
                lang_code = 'none'
            else:
                # Grab the first available language/value pair
                first_lang = next(iter(label_obj))
                if label_obj[first_lang]:
                    manifest_label = label_obj[first_lang][0]
                    lang_code = first_lang
        # Handle simple V2 string label
        elif isinstance(label_obj, str):
            manifest_label = label_obj
            lang_code = 'en' # Assume English or unspecified

    print(f"  -> Fetched label: '{manifest_label}' ({lang_code})")
    item["label"] = {lang_code: [manifest_label]} # Add label to item

except requests.exceptions.RequestException as e:
    print(f"  -> Error fetching manifest: {e}")
    item["label"] = {"en": [f"Manifest (Error Fetching: {url})"]} #
except json.JSONDecodeError:
    print(f"  -> Error decoding JSON from manifest.")
    item["label"] = {"en": [f"Manifest (Invalid JSON: {url})"]}
except Exception as e:
    print(f"  -> An unexpected error occurred: {e}")
    item["label"] = {"en": [f"Manifest (Processing Error: {url})"]}

time.sleep(REQUEST_DELAY) # Be polite to servers

else:
    # If not fetching, add a generic label
    item["label"] = {"en": [f"Manifest {index+1}"]}

collection["items"].append(item)

# --- Write Output ---
try:

```

```

    with open(OUTPUT_FILENAME, "w", encoding="utf-8") as f:
        json.dump(collection, f, ensure_ascii=False, indent=2) # Use indent
    print(f"\nSuccessfully created collection manifest: {OUTPUT_FILENAME}")
except IOError as e:
    print(f"\nError writing file {OUTPUT_FILENAME}: {e}")

```

Next I uploaded the collection manifest to github to create the url. This url will be used to write the Mirador script.

link to collection manifest url:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/IIIFViewer/IIIFColle>

Mirador Viewer

With the iiif manifest urls and the iiif collection url, I was able to write a HTML script that used my IIIF collection manifest url in a Mirador viewer. Via the script you can automatically load in the collection into the [Mirador](#) interface. Here you can check all metadata and the images of all records in the dataset.

The script below can be saved as a html file and will automatically use the IIIF collection manifest url (on github) to load in the dataset into a mirador viewer via browser. If you open the html-file and navigate the Mirador interface, you will find the collection. Via this viewer we can check all images in high quality and with all metadata for each record. More info about this procedure can also be found in chapter four of the manual.

link to IIIF record manifest URLs:

https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/IIIFViewer/iiif_man

link to IIIF collection manifest URL:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/IIIFViewer/IIIFColle>

link to manual chapter four:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/blob/main/Manual/ManualEur>

```
In [ ]: # HTML Script to use the mirador viewer with the collection manifest url
# Code generated by Gemini 2,5
# Authorized by Jenske Verhamme

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Teapot Collection (Mirador)</title>
    <link rel="stylesheet" href="https://unpkg.com/mirador@3.4.0/dist/mirad
    <style>
        /* Make Mirador fill the entire page */
        html, body, #mirador-viewer {
            margin: 0;
```

```

        padding: 0;
        height: 100%;
        width: 100%;
        overflow: hidden; /* Prevent scrollbars */
    }

```

```
</head>
```

```
<body>
```

```
    <div id="mirador-viewer"></div>
```

```
    <script src="https://unpkg.com/mirador@3.4.0/dist/mirador.min.js"></scri
```

```
    <script type="text/javascript">
```

```
        // --- YOUR IIIF COLLECTION MANIFEST URL ---
```

```
        // IMPORTANT: This URL should point to the manifest served by your l
```

```
        // It assumes you have run the Python script and are serving the 'ou
```

```
        // If you are using the original 'my-small-collection.json' directly
```

```
        // ensure GitHub Pages is enabled for your repository and use the co
```

```
        const MY_IIIF_COLLECTION_MANIFEST_URL = 'https://raw.githubusercontent.com/
```

```
        // Initialize Mirador viewer
```

```
        Mirador.viewer({
```

```
            id: 'mirador-viewer', // ID of the HTML element where Mirador wi
```

```
            // Add your collection to Mirador's catalog. This is how Mirador
```

```
            catalog: [{
```

```
                manifestId: MY_IIIF_COLLECTION_MANIFEST_URL,
```

```
                provider: 'My Teapot Collection' // Custom label for your co
```

```
            }],
```

```
            // Workspace settings for collections
```

```
            workspace: {
```

```
                type: 'mosaic', // 'mosaic' allows multiple windows; 'single'
```

```
            },
```

```
            window: {
```

```
                // For collections, starting with the table of contents is u
```

```
                defaultSidebarPanel: 'tableOfContents',
```

```
                sideBarOpenByDefault: true,
```

```
                panels: {
```

```
                    info: true,
```

```
                    toc: true, // Table of Contents panel is crucial for col
```

```
                    annotations: true,
```

```
                    search: true
```

```
                }
```

```
            },
```

```
            // Enable debugging to see Mirador's internal logs in the browser
```

```
            // This is CRUCIAL for identifying any issues with your manifest
```

```
            debug: true
```

```
        });

```

```
    </script>
```

```
</body>
```

```
</html>
```

GITHUB



**7. Create a report about your various steps, code used and the result.
Submit the report and your dataset as an exam for the course Data for
Heritage Collections.**

During my project I made a Github repository called 'Europeana Teapots'.

link to full Github project:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main>

Here I uploaded several folders with files regarding the project. I have a folder for the datasets, the IIIF viewer, the manual, the visualisations and a readme. I have also included a readme in every folder with further clarifications on what files are where and on how to use them.

Datasets:

link: <https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/Datasets>

IIIFViewer:

link: <https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/IIIFViewer>

Manual:

link: <https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/Manual>

Visualisations:

link:

<https://github.com/GuacamoleKoala/EuropeanaTeapots/tree/main/Visualisations>

README: for each folder I provided a readme with more info on the contents and use of the files that are included on Github.

link: <https://github.com/GuacamoleKoala/EuropeanaTeapots/README.md>

Manual of the project

As mentioned before, I made a manual to follow up on the project and retrace all steps that were made:

Chapter 1: ManualEuropeanaTeapot_Chapter1_API.ipynb

The first chapter handles on how to retrieve metadata through Europeana API with the use of Google Collab or Jupyter Notebook, and Python language. The goal here is to write a script that search for teapots (f.e. through the query 'teapot'), but also to improve the script by making it look for translations (f.e. 'theepot') and spelling variant (f.e. 'tea pot'). In the end I also want to make improvements to the script to look for specific forms of metadata.

Chapter 2: ManualEuropeanaTeapots_Chapter2_DataRefinement.ipynb

This dataset can then be further refined and can be further analyzed. Therefore, in the second chapter I explain how I cleaned and enriched the dataset with software OpenRefine. The goal here is to use some techniques that are commonly used in OpenRefine and to achieve a dataset about teapots that is relatively clean. This is important for the visualisation and analysis in chapter three.

Chapter 3:

ManualEuropeanaTeapots_Chapter3_AnalysisAndVisualisation.ipynb

In chapter three I look at how to calculate some extra parameters with the use of Excel. Next I visualize the data in the software Tableau Public. I create some sheets and some dashboards with the data.

Chapter 4: ManualEuropeanaTeapots_Chapter4_IIIF_GITHUB.ipynb

In chapter four I give an overview of what I did in GITHUB to upload the project and I also give an overview of what files can be found there. This can be usefull when you want to follow the developments of this project. In chapter four I also have a look at the use of IIIF viewers for our dataset and I look for a solution to show this dataset with a viewer. Here I have a look at writing a HTML script to load the collection into a Mirador viewer.

Links and sources

<https://pro.europeana.eu/page/apis>

Python

<https://www.w3schools.com/python/>

Excel

<https://www.w3schools.com/excel/>

OpenRefine

<https://openrefine.org/docs>

Tableau

<https://www.youtube.com/watch?v=zOR0-nygfDE&pp=0gcJCdgAo7VqN5tD>

<https://www.tableau.com/blog/viz-whiz-when-use-lollipop-chart-and-how-build-one-64267>

<https://public.tableau.com/app/profile/lilla.rasztik/viz/Tutorialsofvisualizations/Tutoria>

IIIF/Mirador

<https://projectmirador.org/>

<https://pro.europeana.eu/page/europeana-and-iiif>

Github

<https://docs.github.com/en/get-started/start-your-journey/hello-world>

This notebook was converted with [convert.ploomber.io](#)