Documentazione tecnica

Progetto PRIN C.Re.Te.
Catalogue of Reinassance Terracotta Sculpture in North Italy

Versione 1.0

Redatto da: Dott. Manuele Veggi

Collaboratore esterno, Università di Bologna

Università di Trento, Dipartimento di Lettere e Filosofia

Università Bologna, Dipartimento delle Arti

28 Novembre 2025

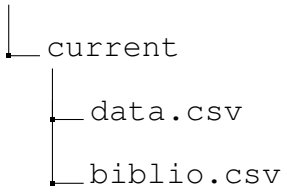
The current document contains the technical documentation of C.Re.Te. (www.c-re-te.it), a website hosting a catalogue of terracotta sculptures produced for Lombardia, Emilia-Romagna, and Veneto regions in Italy during the 15th and 16th century.

Contents

1. File Organisation Overview and Main Pages

The catalog of C.Re.Te. project entirely relies on front-end technologies (HTML, CSS, JavaScript). The corresponding files are uploaded on the server of the University of Trento (Italy) and are organised as follows:

```
(index, project, biblio, search, query, atlas, privacy).html
schede
    entry.html
css
    styles.css
    dataTables.css
    mycss.css
    mycss-entries.css
js
    scripts.js
    generaljs.js
    bib-generator.js
    map-leaflet.js
    entry-populator.js
    query-search-manager.js
    query-display-manager.js
    query-result-sorter.js
assets
    img
        loghi
        img-index
        schede
            list of all entries images
    data
```



As visible in this list, eight HTML pages make up for the content of the catalog:

- *index.html* contains the home page;
- *project.html* presents the scope of the project, most recent events and publication, and the consortium;
- *biblio.html* lists the references of the catalog in an interactive table, built with the dataTables library (v. 1.11.5);[1] its appearance is customised in *css/dataTable.css*;
- *atlas.html* contains an interactive atlas of the production of terracotta sculpture in the Renaissance in North Italy;
- *search.html* consists of an advanced query form; it then redirects to *query.html*.
- *query.html* lists the results of a query with additional filters to refine them. Its content is dynamically updated at every query;
- *schede/entry.html* is a simple html page with blank divs and spans, labelled by an id. As explained later in the documentation, they are dynamically populated with the data of each entry, whose unique identifier is provided in the query string. Its appearance is handled by *css/mycss-entries.css*;
- *privacy.html* contains the privacy policy of the website, revised and approved by the responsible office at the University of Trento.

The website is based on Bootstrap version 5.2.3: its scripts and stylesheets are stored locally—with minor personalisations—on the files *js/generaljs.js*, *js/scripts.js* and *css/styles.css*. Additional customisation to the core theme are instead made in *css/mycss.css*.

[1] See https://datatables.net/.

Most significant changes to Bootstrap template have been made in the use of colours and fonts of the user interface, which has been designed to be as consistent as possible with the visual identity of the University of Trento, Italy. As visible already in the home page, the primary colour of the website is #B10B25 (RGB: 177, 11, 37). The font "Newsreader" by Production Type has been used for the headings, while "Noto Sans" for paragraph and smaller texts. Both fonts are available on Google Fonts.[2]

2. C.Re.Te. Data Model

Being based on pure front-end technology, C.Re.Te. catalogue does not rely on a relational database, rather data are stored in single distinct *.csv files: *data.csv* and *biblio.csv*.

The metadata structure of the first one allows researchers to systematically analyse not only individual sculptures but also their broader artistic and cultural context. The schema includes both information traditionally found in art-historical heritage datasets,[3] and specific fields tailored to the study of terracotta sculpture. Among the former, object title, subject, date (with confidence range), geographical provenance and current location, collecting history (permanence in museums, churches, private collections), documentation of restorations and current conservative conditions, bibliographic data.[4]

Particular additions have been made to the ICCD schema to record all alternative attributions that art historians have proposed for sculptural objects and their motivations/sources. The following table illustrate each metadata field. Note that authorship and provenance of an artwork are represented through multiple repeated fields related to locations, each labelled with a progressive index (e.g. *l0_city, l1_city, ..., l7_city*). In this documentation, this index is rendered through a capital bold I.

[2]See https://fonts.google.com/specimen/Newsreader and https://fonts.google.com/noto/specimen/Noto+Sans.

[3]See as reference the "Scheda OA" (*artwork entry*) by Istituto Centrale per il Catalogo e la Documentazione (ICCD) - Italian Ministry of Culture: http://www.iccd.beniculturali.it/it/ricercanormative/29/oa-opere-oggetti-d-arte-3_00.

[4]See: M. Veggi *et al.*, "An Open Access Catalogue for Renaissance Terracotta Sculpture in Northern Italy", in S. Campana *et al.*, *Digital Heritage*, The Eurographics Digital Library, 2025. doi: 10.2312/dh.20253237

Table 1: Metadata fields of the main dataset

Metadata field	**Description**
id	Unique key identifying the artwork.
autore-**I**-name	Author of the artwork / most authoritative attribution. See *AUTN* field in Scheda OA-ICCD.
autore-**I**-url	Authority file associated to the author (e.g. ULAN), if available.
autore-**I**-rif	Attribution qualifier and specification about the connection with the author (e.g. intervention of the workshop, school of, unsure attribution etc.). See *AUTS* field in Scheda OA-ICCD.
autore-**I**-ambito	Association of this author to a specific regional context or a local artistic school.
autore-**I**-motiv	Motivation and source of the authorship attribution (with primary key of cited publications, described in *biblio.csv*). See *AUTM* field in Scheda OA-ICCD.
oggetto-def	Specific object type. See *OGTD* field in Scheda OA-ICCD.
oggetto-id	Configuration of the object in relation to a context or its status (e.g. fragment, part of a cycle etc.). See *OGTV* field in Scheda OA-ICCD.
soggetto	Iconographic subject of the object. See *SGTI* field in Scheda OA-ICCD.
denominazione	Title or traditional name of the artwork. See *SGTT* field in Scheda OA-ICCD.
tecnica	Technique used to create the artwork. See *MTC* field in Scheda OA-ICCD.
lavorazione	Surface treatment.
descrizione	Free text description of the artwork. See *DES* field in Scheda OA-ICCD.

misure	Free text on the measurement of the artwork
loc-0-geonames, -lat, -long	Gegraphic location of the artwork, indicated by latitude, longitude (only for current location) and associated to a geonames URI
loc-**I**-contenitore	Location of the artwork (repository, e.g. a church, museum, collection etc.). See *LDCN* field in Scheda OA-ICCD.
loc-**I**-comune	Location of the artwork (city). See *PVCC* field in Scheda OA-ICCD.
loc-**I**-prov	Location of the artwork (province, shortened). See *PVCP* field in Scheda OA-ICCD.
loc-**I**-specifica	Further details on the specific location of the artwork. See *LDCS* field in Scheda OA-ICCD.
loc-**I**-isOriginal	Boolean value, indicating whether this location is the original.
data-da	Chronology of the artwork (*post quem*)). See *DTSI* field in Scheda OA-ICCD.
data-a	Chronology of the artwork (*ante quem*). See *DTSF* field in Scheda OA-ICCD.
not-storico-critiche	Free text containing the main art-historical analysis of the artwork. See *NSC* field in Scheda OA-ICCD.
stato-conservazione	Free text on the current conservation state of the artwork.
restauro	Free text on past restoration campaigns on the artwork.
relazioni	Free text field used to create links between different entries and describe the type of relationship. Related works of art have the same content in this field.
bibliografia	Lists of relevant publications on the artwork. It consists of a structured list of unique keys (followed by the specification of the pages) described in *biblio.csv*.
img-path	File path of the main photograph of the artwork

more-imgs-path	List of file paths of other photographs of the artwork (shown only in the carousel of *entry.html*
autore-scheda	Author of the entry (currently, Dr. Andolina and Dr. Scansani)

As mentioned in the table, data of this main *.csv file (*data.csv*) reference to a second dataframe, *biblio.csv*. This document uses a primary key to identify each reference, which is then described by traditional bibliographical metadata (publication type, editor, author, publication, title, venue, publisher, publication year, URL for online resources).

3. Query and Search Algorithm

The JavaScript behaviour of the catalogue is structured in two separate phases:

- *Query generation* with the form elements included in *search.html* and processed in *query-search-manager.js*;
- *Results visualisation* handled by the script *query-display-manager.js*, called in *query.html*, and other additional JavaScript scripts.

The following subsection describes the main components of the search functionalities in the C.Re.Te. catalog.

3.1. Main Search Pipeline

While the home page makes available a free text input field for simple search, in the web page *search.html* users can set the parameters for an advanced query. Users can submit their query either clicking on the button or, thanks to ad hoc inline scripts in both pages, by pressing the Enter key. As visible in the HTML snippet below, this calls the function *sendNewQueryVal()*, which is defined in *query-search-manager.js*.

```
1 <button type="button" class="btn btn-primary fs-5" onclick="
    sendNewQueryVal()">Ricerca <i class="bi bi-search"></i></button>
```

It defines the ancillary functions *getValue(inputElement)* and *getRadioValue*. They enable the script to read the value provided by the user in the select menus and the input fields (the former) and radio buttons (the latter). In the body of *sendNewQueryVal()*, these values are stored in the object *queryParams*.

```
var queryParams = {
   txt: getValue(formEls[0]), // Free text search
   aut: getValue(formEls[1]), // Author
   dfr: getValue(formEls[2]), // Date From
   dto: getValue(formEls[3]), // Date To
   loc: getValue(formEls[4]), // Location
   sub: getValue(formEls[5]), // Subject
   obj: getValue(formEls[6]), // Object
   tec: getRadioValue('tech-radio-options'), // Technique
   sfc: getValue(formEls[7]), // Surface
};
```

These parameters are later merged and formatted into *queryString*, a single query string, via *encodeURIComponent()*. Lastly, the function redirects the user to the page *query.html*, followed by the query string (if any).

The script *query-search-manager.js* defines the main function for the search algorithm (*runQuery()*), together with *parseQueryURLString()* and its reverse function *getURL-StringfromParams(queryParams)*. These last two ancillary functions are called to retrieve the aforementioned *queryParams* object from the current URL or take these key-value pairs to generate a new query string (used for query refinement through filters, see later).

These functions also include ad hoc lines of codes to handle multiple query parameters for the same key. As an example, *parseQueryURLString()* will translate multiple values for the same key, as in $l?alt\_a = AntonioBegarelli\&alt\_a = PaoloBisogni$, into the list *["Antonio Begarelli","Paolo Bisogni"]*.

Each time the page is loaded, the query is run via *query-display-manager.js*. It firstly calls the function *loadCSV()*, based on the PapaParse library (v. 5.3.0).[5] This function is responsible

[5] See https://www.papaparse.com/.

for the main behaviours of *query.html*:

1. Opening the CSV file and maps them with simple keys of a new array *data* (see the code line *data = results.data.map(row => (...))* in *loadCSV()*; for mapping with original *.csv file, see Annex 7);

2. Starting from the query string, running the query only on the elements of the array *data* for which a unique identifier is available (coloumn "ID" of the source CSV file). The results of the query are saved in the variable *filterData* and sorted (see later);

3. Visualising the results;

4. Generating the map visualisation and the filters;

5. Display the set query parameters as interactive buttons (which can trigger the function *removeQueryParams(paramKey)*, see later);

6. Eventually updating the *data* array *filterData*.

3.1.1. C.Re.Te. Batch Search

The architecture of *runQuery()* function, defined in *query-search-manager.js*, is summed up in the pseudo-code algorithm 1. It relies on *data*, loaded by *loadCSV()*. As the total amount of entries was estimated at almost 2000, a simple *brute-force* approach was not possible. Conversely, a **batch search** makes up an optimised solution. As shown, in pseudo-code algorithm 1 the batch-size value *s* is currently set to 25.

Algorithm 1: runQuery function (high level)

```
Result: Array of entries matching the query (d)
q ← parse query string into parameters;
BATCH_SIZE ← s;
refine location fields in q;
d ← [ ];
function advancedSearch(q)
    for i ← 0 to |data| BATCH_SIZE do
        batch ← data[i : i + BATCH_SIZE];
        processBatch(batch, q);
    return d;
function processBatch(batch, q)
    foreach item ∈ batch do
        apply filters to item;
        if matches then
            push item.url to d;
return advancedSearch(q);
```

Before running the full search function, *runQuery()* completes few preliminary steps, namely i) retrieval of the query parameters objects via *parseQueryURLString()*; ii) definition of the batch size; iii) as in the dataset, the location string also includes reference to Italian province (as usual shortened and in round brackets, e.g. "Medole (MN)"), this element is removed to streamline the search process; iv) lastly, initialisation of the array *dataFiltered* (*d* in pseudo-code).

Later, two ancillary functions are defined: *advancedSearch(queryParams)* is the first one and is called directly in the return statement of *runQuery()*. It is responsible to iterate over the dataset, slicing it in mini-batches. Each of them (*batch* in pseudo-code) is then used as input of the second ancillary function, *processBatch(batch, queryParams)*. This second function is instead responsible of the actual search, iteratively applying filters set by query parameters on

the current batch.

Before applying the filters, *processBatch(batch, queryParams)* also defines three ancillary functions:

- *filterLocStringCreator(item,index)*: as seen above and in Annex 7, the provenance of an artwork is rendered through multiple repeated parameters, labelled similarly using a progressive number (here named *index* and later referred to as I, e.g. *l0_city, l1_city, ..., l4_city*). For each *item* of an array, it combines pieces of information (city, container) of the same *index* from different columns to retrieve a single location field (e.g. "Bologna, Santa Maria della Vita"). It is called while searching through previous and current locations (*alt_l* and *l_0* in query parameters object).;

- two matching functions, comparing a *valueArray* (V, the list of available values, retrieved by iterating over *batch*) with a *queryArray* (Q), which contains the values that must be matched.

  - *matchesValues(valueArray,queryArray,isPartialCheck)*, whose boolean *isPartialCheck*, by default set to *true*, enables to toggle between two distinct behaviours. Given the two arrays V and Q, the *partial match* checks if $\exists q \in Q \land \exists v \in V : q \subseteq v$, i.e. if q is a substring of v. On the contrary, the *exact match* checks if there is no dfference between these two strings, i.e. if $\exists q \in Q \land \exists v \in V : q = v$.
  - *allValuesMatchExact(valueArray,queryArray)*, checking if there is an exact match through multiple fields (multiple match). This function is called exclusively for repeated fields (namely alternative authors and previous locations). In this case, repeated fields are manually merged into a new array which is later passed to the function as *queryArray* argument.

  In order for these two functions to accept either a single value or multiple values, if *queryArray* is not already an array, it wraps it in one. Additionally, both functions are not case sensitive and their output is a boolean value.

For each query, these functions are applied for the different query parameters. Table provides a more detailed overview on the topic, associating for each query parameter (in

bracket the key of *queryParams*), the corresponding properties (column names, see Annex 7) of *dataFiltered* making up the *valueArray* and the matching functions.[6] In the second column, the bold *I* identifies the increasing indices for repeated fields, as described above. The last fourth column declares instead if these queries are called from the main search page (*search.html*), the filters in the results page (*query.html*) or in the hyperlinks of the single entry (see later).

A partial exception to this filter application algorithm is the free text search. In this case, a search is run for each word of the input string. Only results meeting all these queries are selected.

Table 2: Application of filters on the batch

Query Parameter in *Q*	**Property in** *V*	**Function**	**From**
Free text search (*txt*)	*all*	Multiple *PM*	*Search*
Authors and other attributions (*aut*)	*author, author-amb, author-**I**, author-**I**-amb*	*PM*	*Search*
Current author (*a_0*)	*author*	*PM*	*Filter*
Other attributions (*alt_a*)	*author-**I***	*MM*	*Filter*
Object (*obj*)	*obj-def*	*EM*	*Search, Filter*
Subject, Denomination (*sub*)	*subj, deno*	*PM*	*Search, Filter, Entry*
Technique (*tec*)	*tech*	*EM*	*Search, Filter*
Surface (*sfc*)	*lav*	*EM*	*Search, Filter*
Chronology (from) (*dfr*)	*date-from*	*DM*	*Search, Filter*
Chronology (to) (*dto*)	*date-to*	*DM*	*Search, Filter*
Location (any) (*loc*)	*l**I**-cont, l**I**-city, l**I**-prov*	*PM*	*Search, Filter*
Current location (*l_0*)	*l0-cont, l0-city**	*EM*	*Filter*

[6]In the table, these functions are referenced as *PM* and *EM* (*matchesValues*, partial match and exact match respectively), *MM* (*allValuesMatchExact*, multiple match). Dates, instead, are compared converting the input date as an integer (*DM*). Lastly, an asterisk signals that the function *filterLocStringCreator(item,index)* is called.

Parameter	**Keys**	**Function**	**From**
Current location (city) (*city*)	*l0-city*	*EM*	*Filter (map), Entry*
Previous locations (*alt_l*)	*l**I**-cont, l**I**-city**	*MM*	*Filter*
Connected artworks (*rel*)	*rel*	*EM*	*Entry*

As shown in pseudo-code algorithm 1, all these statements apply the filters and check if there is a match; if not, the boolean value *isMatch*—initialised as $true$—is set as $false$ and it is used to filter out the not corresponding entries. Conversely, the ID of the ones satisfying the query parameters are pushed to the dataset *dataFiltered*.

3.2. Results Visualisation and Filter Generation

The page *query.html* contains the results of the query. Its behaviour is almost entirely managed by the script *query-display-manager.js*.

On *window.onload*, the page is initialised and, as soon as the document is ready, the function *loadCSV()*. This algorithm is responsible for:

- loading the source *.csv file as a JavaScript array through the PapaParse library;
- from the query parameters in the query string (see above), running the *runQuery()* function and storing the results IDs in a set;
- using this new set to filter the original dataset;
- sort results chronologically by default (see Sect. 3.2.3);
- render the results;
- render the filters (the accordion, the Leaflet map, and the filter chips of the query parameters).

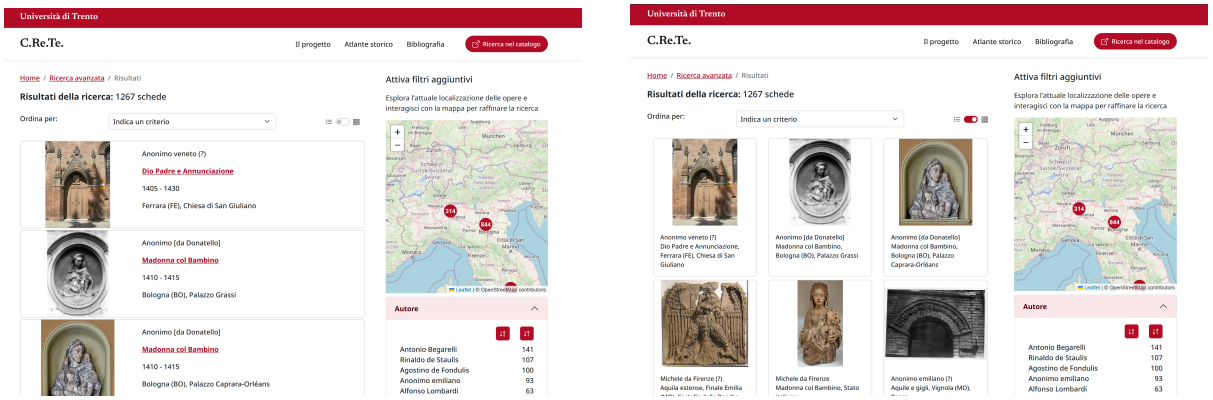


Figure 1: Comparison between the two visualisation modes of query results: list (a) and grid (b)

3.2.1. Visualising Query Results

Query results can be visualised in two distinct modes, either in a list or as a grid. The toggle between these two display modes is handled by a boolean *isGrid*. By default (initialised on *window.onload*), this value is set to false and can be changed by a Bootstrap switch toggle *#viz_mode*. The function *renderResults* takes as argument: this boolean value, an integer referring to the current page and the dataset of the entries matching with the result of the query.

This function is structured in two—similar—branches, on the basis of the visualisation mode (list vs. grid). It firstly sets the maximum number of results per page and uses this value to compute the indices of the first and last results to visualise in the current page. For each of the entries included in this interval it creates a card (either as list or grid item, as visualised in Fig. 1) to populate the rows defined in *query.html*.

In the card, an ancillary function *createLocLabel(item)* is called: this algorithm is responsible for converting location metadata into a single string formatted as *<city> (<province, shortened>), <container>*. Exceptions for private collection and auctions are hard coded as *if* statements.

Lastly, *renderResults* calls a second ancillary function, named *renderPagination*. Its scope is to build the pagination system as per Bootstrap documentation.[7] The functions add list items of class *.page-item* for the first and last page, the current page and its two adjacents ones, and

[7] See https://getbootstrap.com/docs/5.2/components/pagination/.

disabled button for the ellipsis. Lastly, it also provides an input field to directly access a specific page.

Movements among these pages is ultimately handled by new calls to *renderResults*. Conversely, in the input field, user can confirm the desired page with an ad hoc button, responsible for calling *goToPage(data, totPage)*: it checks whether the given page number is acceptable (not out of boundaries) to later call the usual function *renderResults*.

3.2.2. Rendering Filters

Filters of the *query.html* page are essentially of two kinds: those visualised in the accordion and the interactive Leaflet map.

Filters in Accordions The function *renderFilter(data)* is responsible for the visualisation of the former ones, which are formatted as:

a) sortable frequency tables

b) input fields to chronologically refine the search

c) non-sortable frequency tables (for the fields *oggetto*, *tecnica*, *lavorazione*)

Apart from (b), *renderFilter(data)* calls a system of ancillary functions to create the body of the accordion of (a) and (c) filters, as visualised in Fig. 2.

The first called function is *renderFreqTableInAccordion*, which takes as input five different arguments:

1. the *data* array, containing the results of the query;
2. an array containing the property. For most of the properties in the dataset, it consists of a single-item array (e.g. *['subj']*), with the exception of the previous authorship attributions (an array of multiple *author*-**I** fields: *['author-1', 'author-2',...]*, and previous locations of the artworks. In this last case, each item of this array is in turn an array of three items, in the form *['l**I**-city', 'l**I**-prov', 'l**I**-cont']*;
3. a string of the label describing the property of the filter, which will be visualised in the header of the accordion;

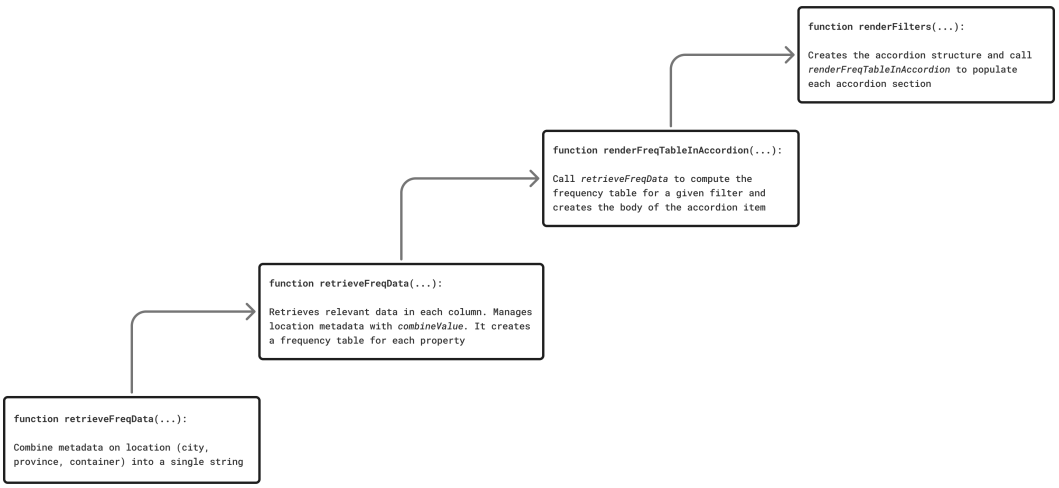


Figure 2: Visual representation of the ancillary functions involved in the creation of the filters.

4. a boolean value *isFirst*, used to determine whether to add the *.collapsed* class to the accordion button of each accordion item;

5. a second boolean value, *hasSortingButton*, to add the functionality to re-sort the elements in specific filters (a);

6. a final boolean value *isByName*, determining whether the rows of the frequency table should be ordered alphabetically or by frequence (descending)

The content visualised in accordion body, instead, is the output of *retrieveFreqData*. It is responsible for three key processes:

1. creation of a large list (*propertyArray*) of all the values that should be counted. Its behaviours on the type of property under consideration:

    - for past authorship attributions, they iterate on all the items of the input array (*author-1, author-2* etc.) and append them on *propertyArray*;
    - for past locations, the algortithm still iterates on the elements of the input array, now calling a new function: *combineValues*. It merges the different location metadata into a single location string, handling exception for private collations and for town districts (always separated by colon in the dataset);

- for current location, the same call to *combineValues* function is made, now without iterating in a *for* loop;
- for all the other properties, *combineValues* is still called. In this way, the algorithm can handle simple properties (e.g. *['subj']*): the nested *if statement* structure of the *combineValues* function just returns an array of the raw values of that property.

2. refinement of author fields in the case of multiple authors for the same artwork (separated by semicolon) or of artworks derived of a model (here, the field is in the form *<author of the artwork> [da <author of the model>]*, e.g. *Anonimo [da Antonio Rossellino]*)

3. counts of the elements in the *propertyArray* and sort them either by name or by values. In the former case, the functions described in Sect. 3.2.3 are called.

This chained system of ancillary functions is used anew in *resortFilter*, called *onclick* the two re-sort buttons in the first row of each accordion item. It updates the content of the accordion body for a single property relying on *retrieveFreqData*.

Leaflet Map The second class of filter is an interactive map, displaying the current location of the artworks matching the provided query parameters. They are visualised above the filters accordion and is handled by the JavaScript script *map-leaflet.js*, based on the Leaflet library (v. 1.9.4).[8] The function *leaflet_data()* first aggregates the artworks by city, considering only records with valid latitude and longitude, and builds a structure that stores for each location its province, coordinates, the list of artworks associated with it, and their total count. This converts the dataset from an artwork-centred to a location-centred representation.

A new Leaflet map is then created—removing any previous instance—and initialized with an OpenStreetMap base layer. To handle potentially dense geographic distributions, the function employs the MarkerCluster plugin (v. 1.5.3),[9] which groups nearby markers and replaces them with a single cluster icon indicating the total number of artworks it represents. The visual appearance of clusters, including colors and polygon outlines, is customized via CSS directly

[8] See https://leafletjs.com/.
[9] See https://github.com/Leaflet/Leaflet.markercluster.

injected into the document head (always in *map-leaflet.js*, classes *.cluster-circle* and *.cluster-icon*). Lastly, it also imports the Leaflet Full Screen Control plug-in (v. 1.6.0).[10]

For each city, the function generates a marker using a custom SVG. Each marker is linked to a popup containing the city name, its province when available, and the number of artworks found there, together with a button that triggers a search filtered by that location (*refineQuery*, see Sect. 3.2.2). The markers are added to the cluster group, which is then displayed on the map.

Filter Chips for Query Parameters In *query-display-manger.js*, the function *display-Params()* is responsible for generating the visual filter chips - visualised as Bootstrap outline buttons - that summarize the active search parameters selected by the user.

It firstly extracts all query parameters from the URL through *parseQueryURLString()*, then compares them with a predefined mapping that associates each parameter key with a human-readable label (see Tab. 2) and establishes a coherent display order.

The function clears the container that holds the filter chips and iterates through the mapping list, ensuring that the chips appear in a structured and hierarchical sequence rather than in arbitrary URL order. For each parameter actually present in the query string, the function composes a single textual representation—capable of handling multiple values—and generates a small button styled as a removable chip.

Each chip includes the parameter's label and its current value, and embeds a call to *remove-QueryParams()* (see Sect. 3.2.2) so that users can delete individual filters dynamically. The resulting chips are appended to the interface above the list / grid visualisation of the results.

Refine the query Filters enable users to obtain a synoptic visualisation of the data, highlighting the most frequent values and allowing them to refine the query by adding additional parameters. For filters displayed in the accordion and in the Leaflet map, this behaviour is managed by the function *refineQuery(...)* in *query-search-manager.js*.

The function takes as input the filter selected by the user and a filter type, and then updates the URL accordingly so that the search results page can be reloaded with the new constraints.

[10] See https://brunob.github.io/leaflet.fullscreen/.

Internally, an auxiliary function constructs the new query string by appending the appropriate parameter to the existing URL, taking care of whether a query part is already present. When the filter originates from a row in the accordion, the function maps the human-readable label (such as “Autore”, “Ubicazione attuale” or “Tecnica”) to the corresponding URL parameter key (for instance *a_0*, *l_0*, *tec*) and appends the selected value as a new query token; the browser is then redirected to the updated URL, which triggers a new search.

When the filter is selected from the Leaflet map, the city name is encoded as a city parameter and appended in the same way, so that the map can act as an entry point for location-based refinement. Date filters are handled differently: since the “from” and “to” years (*dfr* and *dto*) must overwrite any existing temporal constraints, the function first parses the current query string into an object, updates or replaces the date-related fields with the newly chosen values, and then reconstructs the full query URL from this updated parameter set.

Conversely, clicking on filter chips (Sect. 2) triggers the function *removeQueryParams(...)*, always defined in *query-search-manager.js* script functionalities: it removes the query parameters associated to the chip and run the updated query.

3.2.3. Sort Functionalities

A final script, *query-result-sorter.js*, is responsible for the sort functionalities available in *query.html*. In this scripts, ancillary functions are called to handle sorting functions in the main results visualisation, in the sortable filters in the accordions (labelled as type (a) in Sect. 3.2.2). Additionally, between the filter chips and the results list, a select form allows user to sort by author, subject, object, date, and current location (city)

Sort by Date Results date fields are firstly normalised and parsed as integer by *parseYear()*; non-numeric, empty or null entries are instead returned as null. This function supports the comparator *compareByYearThenTo()*, which applies a two-stage ordering: it first compares the “from” year (*date-from*) of two records, placing entries with missing values at the end, and sorting the remaining ones in ascending order. If the “from” years are equal, it then compares the “to” years (*date-to*) using the same rules. When both ranges are identical or missing, the function returns zero.

Sort by Author Name The naming conventions for artists in Early Modern Era is peculiar and a traditional alphabetical sorting by name or surname may be not always meaningful. To overcome this limit, a custom order has been set as list and stored in the constant *CUSTOM_AUTHOR_ORDER*.

On this basis, the JavaScript scripts implements a custom ordering mechanism. At its core lies a second constant, *AUTHOR_RANK*, a map that associates each author (stored as a normalised string) with its position in *CUSTOM_AUTHOR_ORDER*. Author names are first normalised by converting them to lowercase, trimming whitespace, removing quotation marks and collapsing multiple spaces, so that variations in formatting do not affect the sorting logic. A small special case is handled explicitly for "Luca della Robbia il giovane", whose variant with quotation marks is mapped to the same rank. The auxiliary function *isAnonimoName(string)* identifies anonymous attributions by checking whether the normalised string starts with "anonimo"; these entries are always pushed to the end of the ordering.

The function *authorRankValue* takes an author field that may contain multiple names separated by semicolons, extracts the first name, and returns a pair rank, key: anonymous or empty authors are assigned an infinite rank so that they appear last, while known authors either receive the rank defined in *AUTHOR_RANK* or, if not present in the custom list, are sorted alphabetically using their normalised name as key. The comparator *compareByAuthor()* applies this logic to two records by first comparing their ranks (i.e. the custom order) and, in case of ties, falling back to alphabetical order.

Main Sorting Function The function *sortData(criterion)* provides a unified entry point to sort the current dataset according to different criteria and then re-render the results. When sorting by city (*l0-city*), it temporarily separates records belonging to "Collezione privata" (private collection) and "Ubicazione ignota" (uknown location), as well as entries with missing city information, sorts the remaining records by city name, and then appends the private, unknown and missing-location entries at the end in a controlled order.

When the criterion refers to dates (*date-from* or *date-to*), it calls the date comparator (*compareByYearThenTo*); when sorting by author, it uses *compareByAuthor* described above. For all other fields, a simple alphabetical sort on the lowercased string value is used as a fallback. In

all cases, a copy of the data is sorted, the global dataset is updated, and *renderResults()* (Sect. 3.2.1) is invoked to display the newly ordered lists.

4. Single Entry Creation

All the entries of the C.Re.Te. catalogue are rendered on the same HTML page, *schede/entry.html*. This document contains the structure of the entry, organised in divs and spans labelled by an id. These are dynamically selected using JQuery library (version 3.6.0) and populated through the script *entry-populator.js*.

This system relies on the query parameters of the URL (e.g. *entry.html?no=1*), which contains the unique identifier of the entry. The script calls an asyncronous function, *init()*, which loads the datasets of the artworks and of the bibliography via PapaParse and retrieves the value of the "no" parameter in the URL. Later, it calls the ancillary function *filterById()*, which collects the associated metadata from the main dataset and stores it in the array *filteredData*. It lastly calls the core function of the script, *uploadData(filteredData)*.

It is responsible for updating the divs and the spans in the source HTML file, directly uploading data from the source *.csv:

- author and previous attributions, handled by an ad hoc function (see below). The rendering of the motivation is instead handled by the *bib-generator.js* script (see Sect. 4.1)
- chronology
- object definition and subject
- technique and surface working
- object description and measures
- previous and past locations. The content of these fields are generated by *createLocLabel()*, which—as in other JS scripts of this catalogue— is responsible for merging different metadata (city, province, container, and—for previous locations—) into a single
- art-historical and stylistic analysis of the artwork

- conservation state
- bibliographical references (see Sect. 4.1)
- connected artworks

The generation of the author strings is delegated to a new ancillary function, *getAuthorString()*, which takes as input the array *authorData*. This object collects the main author's metadata (name, ULAN URL, bibliographic reference, field, and motivation) and a list of up to four alternative attributions with the same structure.

The function then converts the main author's data into a HTML snippet, which will be injected into the page. The core logic of *getAuthorString()* is to normalize and format different kinds of authorship strings into a consistent layout while preserving special cases. In particular, it hard-codes exceptions for anonymous author with identified name in brackets (e.g. "Anonimo dell'Italia settentrionale [Antonio Rossellino]") and for multiple authors separated by semicolons (e.g. "Ludovico Castellani; Michele da Firenze"). Lastly, for each author it adds a superscript hyperlink to a ULAN authority file (if available).

Some fields of the injected HTML content also contains hyperlinks to run new queries: authors and past attributions, subject, the current collocation (city), and the related artworks. At the bottom of the page, the script generates a suggested citation of the entry, stating its author. Lastly, the source HTML file provides an hyperlink to a Google Form to signal corrections and integrations to the information contained catalogue (https://forms.gle/uv4NyaHAuTCp7xpN7).

4.1. Bibliography Generation

In the entry, the script *bib-generator.js* is responsible for the generation of the bibliography. Its functions are called in two different locations:

a) as a in-text citation in the motivation supporting an authorship attribution;

b) at the end of the entry, for the creation of the reference list.

As mentioned above, the *init()* function of *entry-populator.js* reads the *biblio.csv* file and stores it into a single array (*bibData*). This object is then processed by the function *retrieveBibData(bibId)*: given a citation key in the motivation (*autore-**I**-motiv*) and general bibliography

field (*bibliografia*) of *data.csv*, it searches the bibliography dataset for the matching record (by "ID") and returns the corresponding object.

For the motivation supporting authorship attributions (a), the function *refineMotivationField(cell)* is evoked. It refines *autore-**I**-motiv*, checking if it contains the term “Bibliografia”. If so, for each bibliographic key, it retrieves the bibliographic record via *retrieveBibData*, creating an author-year in-text citation to be injected in the HTML. Conversely, if no bibliographic reference is present, the function simply returns the original text unchanged.

The reference list (b) is instead handled by *gitBibString(cell)*, which in turn calls the function *retrieveBib(cell)*. It is responsible for decode the content of the input cell, structured as list (whose items are separated by semicolon) of bibliographic keys with page specifications, separated by a comma. For each item, it gets the metadata via *retrieveBibData* and it later generates a full bibliographic string through *getFullRef()*. This new function reads the type of the publication and builds the correctly formatted citation using a switch statement, evoking a series of ad hoc “minor” formatter (*getArticle()*, *getMonograph()*, *getEssayInBook()*, *getEntry()*, *getThesis()*).

As a result, the output of *retriveBib()* is a list of citation strings: *getBibString()* later wraps each of these references in a *p* element and concatenates them into a single HTML fragment. *uploadData()* (see above) is lastly responsible for injecting it in the source page.

5. Visualisations

Besides the interactive map available in the filter section (see Sect. 3.2.2) of *query.htmk*, the C.Re.Te. catalogue also provides an additional interactive visualisation of the artworks, available in the section "Historical atlas" ("Atlante storico", i.e. *atlas.html*).

This page imports a map created on Fluorish (https://flourish.studio/), wrapped into a div of class *.fluorish-embed* (suggested import). The map contains an animation showing at the bottom a progressing timeline, with a line chart representing a timeline of the produced artworks. Over time, different points on the map are shown, which rapresents the original location / originally intended location (see Fig. 3).

The map has been built upon the "3D map with animated points" template available on Flu-

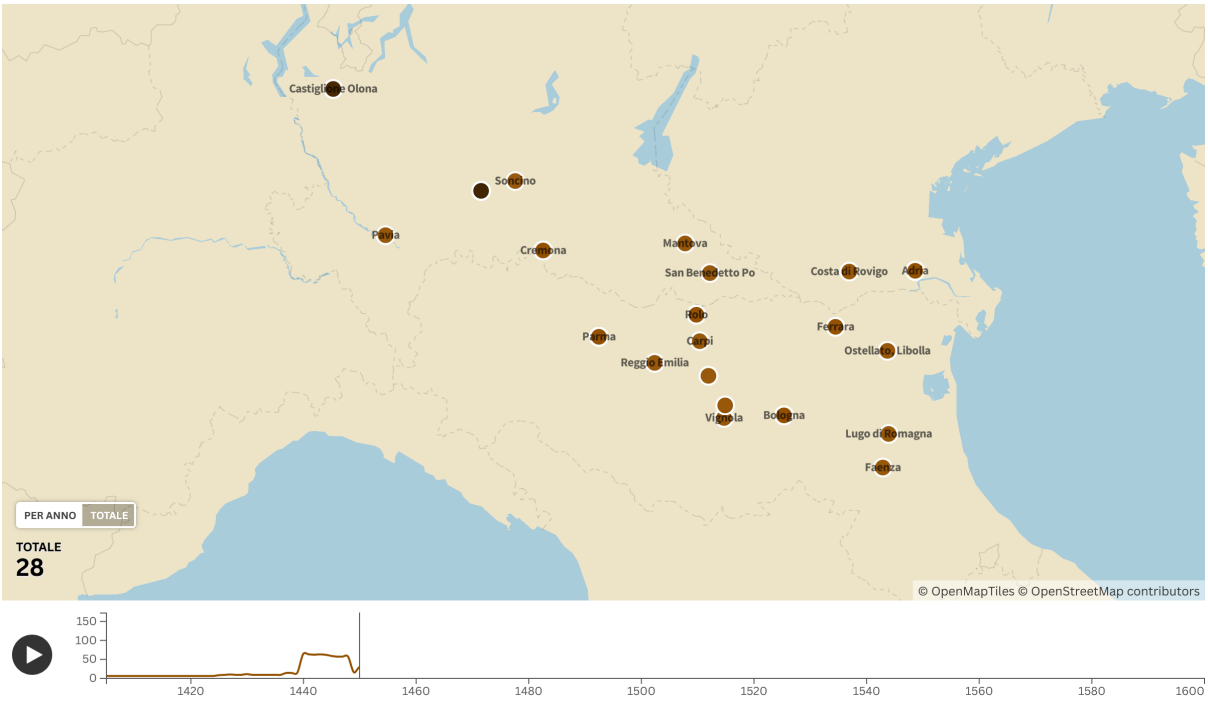


Figure 3: Interactive and animated map in the "Historical atlas" section - C.Re.Te. Catalogue

orish.[11] To populate this visualisation, a new dataset has been derived from *data.csv*, including all those artwork for which the original location is known. This dataset is pasted in the "Points" sheet of the map data tab on Fluorish and contains:

- URL of the entry (*entry.hmtl?no=<ID of the artwork>*): each points of the map represent an artwork. While hovering, a small modal appears, containing the main metadata and a direct hyperlink to the corresponding entry;
- Name of the artwork (author and subject);
- Chronology of the production (as interval between two dates);
- Object type;
- Original location (city), with corresponding latitude and longitude

6. How to Update the C.Re.Te. Catalogue

C.Re.Te catalogue can be updated when metadata of new artworks are available. This section provides a checklist to complete this procedure:

[11] See \urhttps://flourish.studio/blog/animated-point-map/.

1. **Clean *data.csv* and *biblio.csv***. Before uploading, controls if newline characters are by mistake present in the source *.csv: this would result in the scripts not being able to correctly parse the dataset;

2. **Upload the source *.csv files** in the folder *assets/data/current*. If you need to upload in a separate directory, change the path in

    - *query-display-manager.js*, function *loadCSV()*;
    - *entry-populator.js*, functions *loadMainCSV()* and *loadBibCSV()*.

3. **Update the full bibliography table** available on the page *biblio.html*. You can either choose to manually add a new table row or to bulk convert data in *biblio.csv* in HTML table. In the GitHub repository for the project documentation, a tailored Python file is available (access GitHub repository at https://github.com/c-re-te/docs);

4. **Update the open-dataset** uploading the new datasets on the documentation repository;

5. **Update the Flourish map**. When a new version of the dataset of the artworks with original locations is available, upload it on Fluorish to visualise the new data in the interactive map. For access in edit mode at the Fluorish project, contact Manuele Veggi (manuele.veggi@gmail.com).

Acknowledgment

The project received funding through the call PRIN 2022 (Catalogue of Reinassance Terracotta Sculpture in North Italy).

7. Annexes

Annex 1

Comparison between original catalogue column names (in the source *.csv) files and mapped field names used in the *data* array of the function *loadCSV()*. As above, index of repeated fields is indicated by **I**.

Original column	**Mapped field**
id	url
autore-0-name	author
autore-0-rif	author-rif
autore-0-ambito	author-amb
autore-**I**-name	author-**I**
autore-**I**-rif	author-**I**-rif
autore-**I**-ambito	author-**I**-amb
data-da	date-from
data-a	date-to
soggetto	subj
denominazione	deno
oggetto-def	obj-def
tecnica	tech
lavorazione	lav
loc-0-lat	lat
loc-0-long	long
loc-0-contenitore	l0-cont
loc-0-comune	l0-city
loc-0-prov	l0-prov
loc-**I**-contenitore	l**I**-cont
loc-**I**-comune	l**I**-city
loc-**I**-prov	l**I**-prov

loc-**I**-crono	l**I**-cron
img-path	path
descrizione	desc
not-storico-critiche	critic
stato-conservazione	conserv
restauro	rest
relazioni	rel