This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 870811

Social cohesion, Participation, and Inclusion through Cultural Engagement

D5.2 Prototype interfaces for interpretation and reflection

Deliverable information	
WP	WP5
Document dissemination level	PU Public
Deliverable type	R Document, report
Lead beneficiary	PG
Contributors	GVAM, CNR, UNITO
Date	28/04/2022
Document status	Final
Document version	1.0

INTENTIONALLY BLANK PAGE

Project information

Project start date: 1st of May 2020
Project Duration: 36 months
Project website: https://spice-h2020.eu

Project contacts

Project Coordinator
Silvio Peroni
ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
Department of Classical Philology and Italian Studies – FICLIT
E-mail: silvio.peroni@unibo.it

Project Scientific coordinator
Aldo Gangemi
Institute for Cognitive Sciences and Technologies of the Italian National Research Council
E-mail: aldo.gangemi@unibo.it

Project Manager
Adriana Dascultu
ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
Executive Support Services
E-mail: adriana.dascultu@unibo.it

SPICE consortium

No.	Short name	Institution name	Country
1	UNIBO	ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA	Italy
2	AALTO	AALTO KORKEAKOULUSAATIO SR	Finland
3	DMH	DESIGNMUSEON SAATIO - STIFTELSEN FOR DESIGNMUSEET SR	Finland
4	AAU	AALBORG UNIVERSITET	Denmark
5	OU	THE OPEN UNIVERSITY	United Kingdom
6	IMMA	IRISH MUSEUM OF MODERN ART COMPANY	Ireland
7	GVAM	GVAM GUIAS INTERACTIVAS SL	Spain
8	PG	PADAONE GAMES SL	Spain
9	UCM	UNIVERSIDAD COMPLUTENSE DE MADRID	Spain
10	UNITO	UNIVERSITA DEGLI STUDI DI TORINO	Italy
11	FTM	FONDAZIONE TORINO MUSEI	Italy
12	CELI	MAIZE SRL	Italy
13	UH	UNIVERSITY OF HAIFA	Israel
14	CNR	CONSIGLIO NAZIONALE DELLE RICERCHE	Italy

Executive summary

SPICE is an EU H-2020 project dedicated to research on novel methods for citizen curation of cultural heritage through an ecosystem of tools co-designed by an interdisciplinary team of researchers, technologists, and museum curators and engagement experts, and user communities. The aim of this document is to describe the initial implementation of the citizen curation interfaces that support the citizen curation activities defined in the use cases of the SPICE project, organized as framework of reusable interface components named *inSPICE*. In addition to the components, inSPICE also includes several templates implementing the citizen curation scripts identified in the use cases.
This document describes the software architecture of the inSPICE framework along with the supported citizen curation activity templates and the interfaces components included in the framework.

Document History

Version	Release date	Summary of changes	Author(s) - Institution
V0.1	25/01/2022	First draft released	PG
V0.2	1/04/2022	Revisions and integration	All partners
V0.3	15/04/2022	Internal review	GVAM, UNITO
V0.4	28/04/2022	Incorporated comments from internal review	PG
V1.0	29/04/2022	Final version submitted to REA	UNIBO

Table of Contents

1. Introduction

The SPICE project builds on the growing trend for museums, rather than providing an authoritative view, to present multiple voices related to their collection and exhibitions. In SPICE, an approach we term citizen curation is proposed as a way of supporting visitors to share their own interpretations of museum objects and reflect on the variety of interpretations contributed by others. This document describes the initial implementation of the citizen curation interfaces that support the citizen curation activities defined in the use cases of the SPICE project (see also D2.3, D4.5 and D7.5), organized as framework of reusable interface components named inSPICE. inSPICE provides a collection of “templates for citizen curation activities”, where a user, typically museum personnel, can instantiate and configure a given citizen curation activity.

For the development of inSPICE we have followed a bottom-up approach, from the use case prototypes to the templates and components in the framework. From the individual prototypes, we have identified those interface components that can be integrated into a common framework and implemented them. The process starts with the implementation of those components not already in the framework needed to build the use case interface, and then combine them into higher-order components (or "pages") that provide most of the use case functionality.

The developed components will be detailed below based on a preliminary classification into use categories. In this way, we intend to provide a catalogue of support components for the view layer of the different entities and flows present in different citizen curation activities.

Before this, however, it is useful to give a general idea of the workflow followed for the creation and design of the components at hand, which adopts a bottom-up strategy whereby we start from the particular use cases, and progressively adapt the tools designed to solve them so that they become more general-purpose components, and not just tailored to suit their original use cases. The workflow followed is thus activity (use case) driven, and can be summarized in the following steps:

1. For the considered activity, define its view layer requirements in terms of required components and the interaction between them, as well as the types of data with which the activity in question works.
2. Once the functional and component requirements are clear, identify which components from our catalogue could be used, after a potential functionality extension step, to represent those specified in step 1.
   Here we also consider the problem of "translation" of data types between activities: indeed, each activity may define different data types on potentially very similar entities (e.g. data types related to artworks, fundamentally identical between use cases save for subtle differences in naming and a few fields specific to each activity), so that the components that served to manage the view layer of an entity in a use case can often be extended to be able to support the new requirements induced by incorporating the new activity. Additionally, particular data types can be converted to a common data type that generalizes the others, by means of mapping functions defined to solve the problem of incompatibility between use cases.
3. After this identification step, implement from scratch those components that cannot be obtained as a generalization of other components in our catalogue, and

incorporate the necessary extensions in those that can be generated from the previous ones.

The rest of the document runs as follows. Next Section describes the software architecture of the inSPICE framework. Section 3 covers the citizen curation templates included in inSPICE, which implement citizen curation activities from the project use cases. Section 4 and 5 provide a description of the reusable interface components provided by the framework, first describing general purpose components (Section 4) and then those components specifically designed to fulfil the needs of a particular use case in SPICE. Section 6 describes the initial considerations about accessibility of the interfaces which have been initially considered for the GAM Game use case. Last Section discusses current limitations and the next steps.

2. General Architecture

The inSPICE framework is structured in two fundamental blocks used to mediate the communication between the users of the offered templates and the different services and persistence layers belonging to the other work packages within SPICE.

The **first block** comprises a single-page web application developed in React + Typescript, which contains all the necessary building blocks to build and define new templates in a modular way, as well as the higher-order components that combine these atomic pieces to produce specific pages within the application that represent flows associated with particular use cases.

The **second block** is given by a server developed in NodeJS (also using Typescript instead of JavaScript), intended to expose a series of mechanisms and endpoints that enable communication between web applications and the various services offered within SPICE. In particular, at the time of writing, the inSPICE server allows communication with the WP6 reasoning services for the recommendation of works that are similar / opposite to a given one in terms of expressed emotions, and with the WP4 LDH as the main persistence mechanism for a large majority of the data needed to support the operation of the different templates. Additionally, the inSPICE server counts with its own private persistence layer, built on a MongoDB database, aimed at storing any sort of sensitive information that due to privacy and security considerations should not be saved directly on the LDH (information related to user records and credentials, for instance).

An general overview of this structure for one of the templates included in the framework (Find Artwork/ Treasure Hunt activities) can be found in Figure 1.

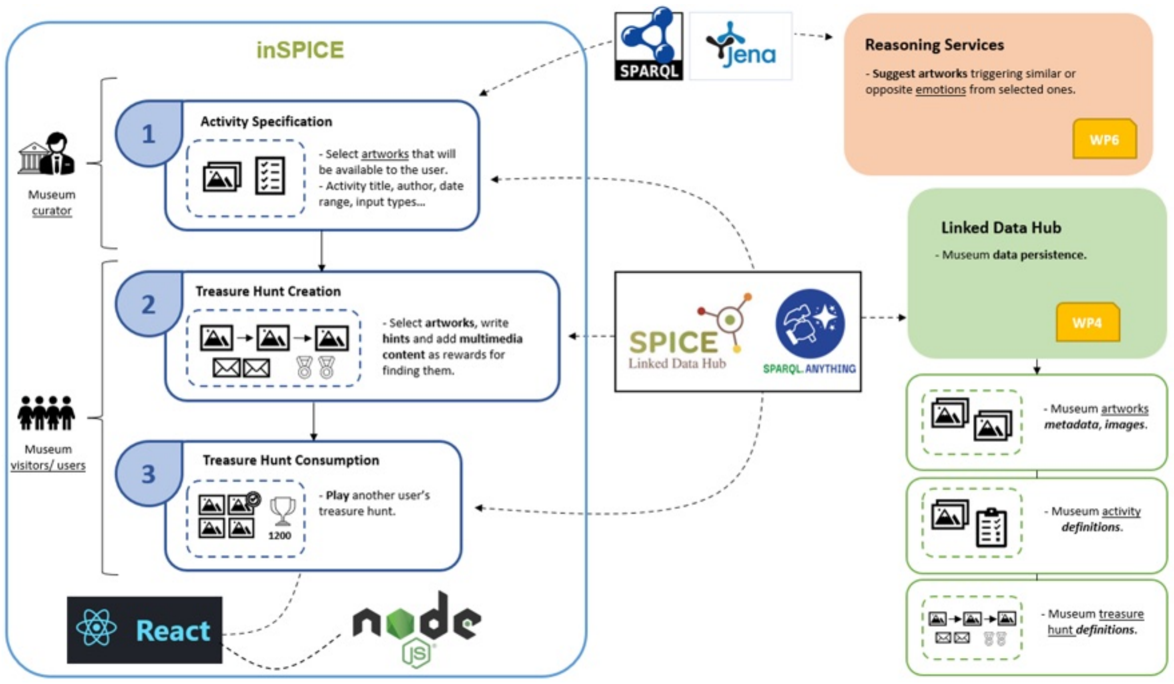

inSPICE
SPARQL
Jena
Reasoning Services
- Suggest artworks triggering similar or opposite emotions from selected ones.
WP6
Museum curator
1
Activity Specification
- Select artworks that will be available to the user.
- Activity title, author, date range, input types...
Linked Data Hub
- Museum data persistence.
WP4
2
Treasure Hunt Creation
- Select artworks, write hints and add multimedia content as rewards for finding them.
SPICE Linked Data Hub
SPARQL.ANYTHING
- Museum artworks metadata, images.
Museum visitors/ users
3
Treasure Hunt Consumption
1200
- Play another user's treasure hunt.
- Museum activity definitions.
- Museum treasure hunt definitions.
React
node JS

Figure 1

2.1. Web Infrastructure - Templates

We will now present the infrastructure of the first building block of the framework in more detail, describing the technologies used and the general structure/logic of this sub-project.

2.1.1. React and Typescript

React is a “JavaScript library for building user interfaces”, while TypeScript is a “typed superset of JavaScript that compiles to plain JavaScript.” By using them together, we essentially build our UIs using a typed version of JavaScript. The reason to use them together would be to get the benefits of a statically typed language (TypeScript) for your UI. This means more safety and fewer bugs shipping to the front end. TypeScript compiles the React code to type-check the code. It doesn’t emit any JavaScript output (in most scenarios). The output is still similar to a non-TypeScript React project. Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen. Using TypeScript with React allows to develop strongly-typed components that have identifiable props and state objects, which will ensure any usages of your components are type-checked by the compiler.

2.1.2. Component-driven design - Storybook

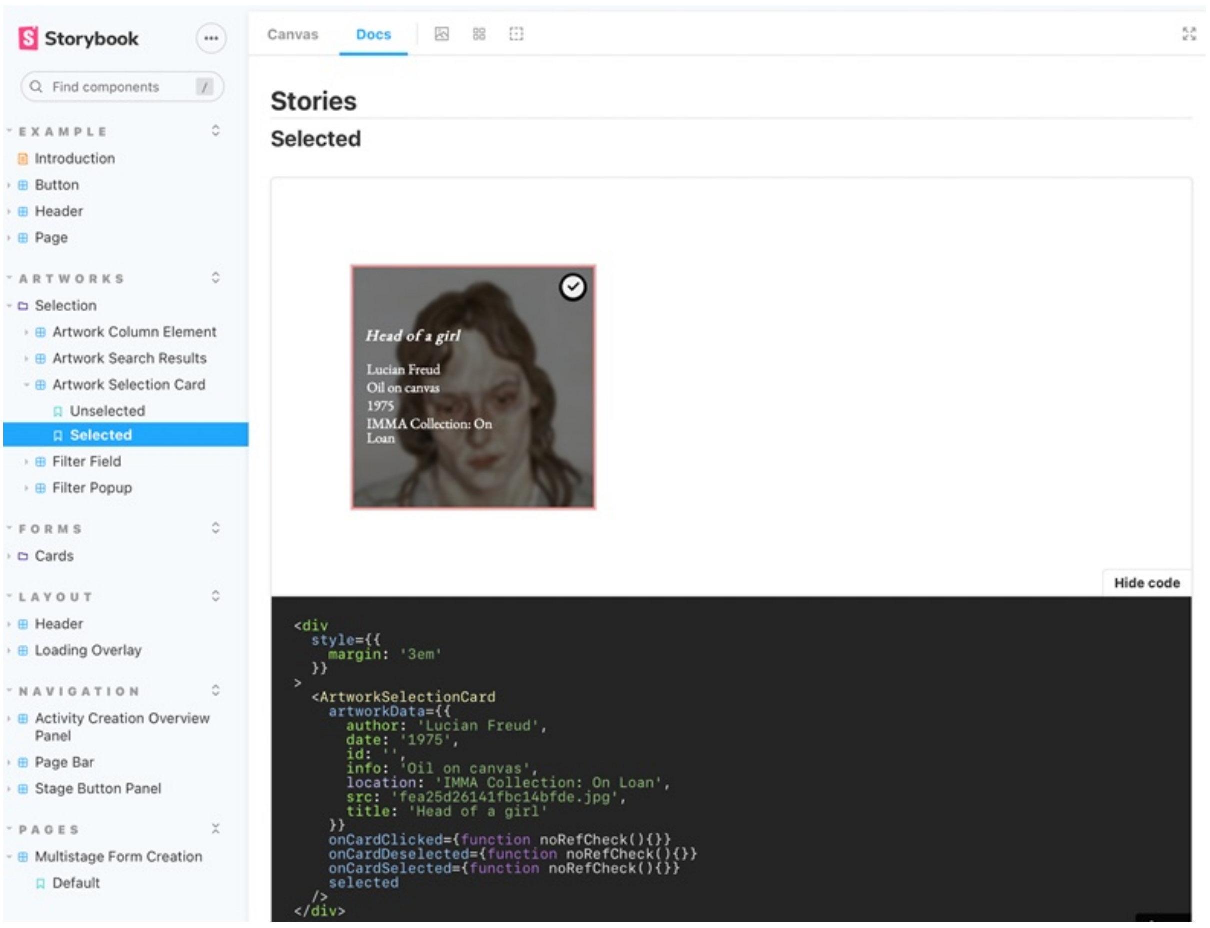
When developing the different flows and components of inSPICE, we chose to follow a component-driven work process, whereby the creation of a new template starts with the development of atomic components and ends with high-level pages. Thus, the process of incorporating a new template into the project code base can be viewed as a succession of the following major steps:

1. Create mockups (see also D5.1) that provide an overview of the workflow to be developed and the interfaces required within the template at hand.
2. Within these mockups, identify those elements for which it is possible to find a component that already exists within the current library and which, possibly after a generalization phase, could be used in the context described in the corresponding mockup. As an example, if a mockup makes use of an interface element that displays a thumbnail of an artwork along with its most important information, and there is already a general component within our library dedicated to rendering such thumbnails, except perhaps for some fields of interest, this can be generalized by adding the possibility of incorporating additional fields, and used to build the UI of the given mockup.
3. For all those elements that do not seem to have a sufficiently similar component within the framework library, create new dedicated components and incorporate them into the project repository.
4. Once all the components needed to build the interface described in the mockup are implemented, combine them into higher-order components (or "pages"), which should normally be independent of external data. By this we mean that the pages referred to in this step are limited to rendering content and exposing a series of callbacks in their props to notify their parent components of the various end-user interactions with the interface, but do not make any actual API calls themselves.

In order to isolate and simplify the generalization of existing components in point 2 and the development of new components in point 3 as much as possible, the project makes use of *Storybook*[1], an open source tool for building UI components and pages in isolation. In practice, this means that it is possible to test and visualize the different components in the library without the need to integrate them into any larger part of the application. This is convenient not only for development (the programmers/designers in charge of the creation of atomic components do not have to interact directly with the rest of the framework to know that their work is functional), but also for the distribution of the project, since it essentially acts as an interactive gallery of components that can be easily distributed to third parties to dynamically display the developed elements, and even provide code samples to exemplify their use. With this, Storybook also serves as UI documentation, additionally enabling the inclusion of illustrative notes and explanations based on the comments embedded in the code.

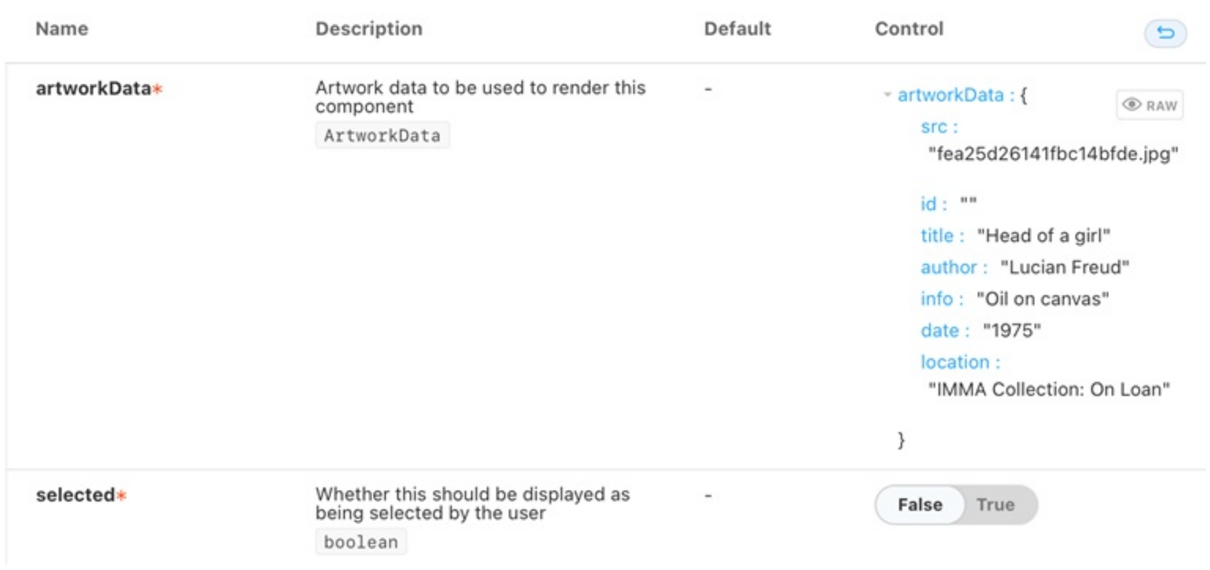
At the functional level, Storybook relies on the use of "stories", defined as functions that return the rendered state of a component given a set of compatible props as a parameter. In practice, this means that each component that should be included in the Storybook catalogue needs to at least expose a story to the framework that specifies how to render that component when it appears in the documentation. In general, most stories simply return an instance of the component at hand with a set of default parameters, but in higher-order components it may be necessary to provide additional mocking of services (e.g. stories concerning complete pages within the application), or of global contexts (theme, styles, language, etc.). An example of a story with auto-generated reference code can be found in Figure 2, and the associated menu to dynamically interact with the different props that are passed to the component in Figure 3.

[1] https://storybook.js.org



```
<div
  style={{
    margin: '3em'
  }}
>
  <ArtworkSelectionCard
    artworkData={{
      author: 'Lucian Freud',
      date: '1975',
      id: '',
      info: 'Oil on canvas',
      location: 'IMMA Collection: On Loan',
      src: 'fea25d26141fbc14bfde.jpg',
      title: 'Head of a girl'
    }}
    onCardClicked={function noRefCheck(){}}
    onCardDeselected={function noRefCheck(){}}
    onCardSelected={function noRefCheck(){}}
    selected
  />
</div>
```

Figure 2



Name	Description	Default	Control

artworkData*	Artwork data to be used to render this component
	`ArtworkData`
		-	artworkData : {
			src : "fea25d26141fbc14bfde.jpg"
			id : ""
			title : "Head of a girl"
			author : "Lucian Freud"
			info : "Oil on canvas"
			date : "1975"
			location : "IMMA Collection: On Loan"
			}

selected*	Whether this should be displayed as being selected by the user `boolean`	-	False True

Figure 3

2.1.3. Handling of Services

As mentioned earlier, the application "pages" or "screens" associated with specific flows should be as isolated as possible from any type of logic that requires calling external services, so that it is possible to fully develop a complete interface with mocked data without the need to understand how this data is obtained or how the user's actions translate into communication processes with the server.

As a result, all "screen" type components are usually divided into two nested parts: a parent element that manages the use logic of services and calls to external APIs, and a child element that receives the relevant data from the parent and uses it to render the interface, possibly calling a series of callback methods to notify the parent component of the user's actions. Upon receiving these actions, the parent component in charge of communicating with the services may trigger a series of API calls to submit information or update the data rendered in the child.

These components dedicated to encapsulate rendering elements and manage communication with services make use of their own hook, useAsyncRequest, to execute requests to external APIs from services. This hook is reused throughout the framework to centralize the asynchronous function triggering logic, and has a number of interesting properties that justify its use over the direct use of a service function within a *useEffect* hook:

- Every asynchronous request handled by the hook is wrapped in a union data type that allows specifying in which particular state the request is in. This is also convenient in that when the request ends up in a success or failure state, the result can be easily retrieved and assumed to be of the type specified when declaring the request function.

```
export type AsyncProgress<ResultType = any, ErrorType = any> =
  | { kind: 'idle' }
  | { kind: 'running' }
  | { kind: 'success', result: ResultType }
  | { kind: 'failure', reason: ErrorType }
  ;
```

  More specifically, the hook keeps an object adhering to the above type as its internal state, and updates it as the request progresses. This means that any change in that state triggers a rerendering in a React component that makes use of the hook, allowing conditional renderings as in the following example:

```
const [fetchActivityArtworksStatus] =
useAsyncRequest(fetchActivityArtworks, []);
if (fetchActivityArtworksStatus.kind === 'running') {
  return <LoadingOverlay message='Fetching activity artworks' />;
}

if (fetchActivityArtworksStatus.kind === 'failure') {
  return <ErrorPanel message={`There was an error when fetching the
activity artworks:
${fetchActivityArtworksStatus.reason}
`} />;
}

// cache the value of our activityArtworks after they are fetched.
const artworks = fetchActivityArtworksStatus.result.data;
```

```
if (artworks !== undefined) {
  return (
    <GamGameUserFlow
      activityDefinition={activityDefinition}
      artworks={artworks}
      artworkCount={artworks.length}
    />
  );
}

return <ErrorPanel message='Found no activity artworks' />;
```

This allows for a very exhaustive management of the different scenarios in which a call may derive, and to modify what is shown to the user in each situation in a conditional and declarative way.

- The hook supports a list of dependencies as a second parameter and re-executes the associated request each time these dependencies are modified. This is especially relevant if we are working with some type of page in which the content received from the services must change continuously throughout the life cycle of the component, for example with elements that make active use of pagination systems, or when we want to execute an asynchronous request conditioned to some variable belonging to the internal state of our page (for example, submitting template definitions to be stored in persistence, only when certain conditions over the state of the page are met). The most typical example with a search and pagination component would be:

```
// API Requests to fetch artworks and unique filtering fields
const [fetchDataStatus] = useAsyncRequest(fetchStepData, [page,
  itemsPerPage, appliedFilter]);
```

- Lastly, it is possible to make the request passed to the hook only be executed when a method acting as a trigger is explicitly called, which can be accessed from the array in which the hook is declared, as in the following example:

```
// Perform login request only after actively calling triggerRequest
const [performLoginRequestStatus, triggerRequest] =
useAsyncRequest(performLogin, [], false);
```

Here, the request to perform a login attempt is only executed when the triggerRequest method is explicitly called from some point in the component definition.

The vast majority of asynchronous requests used from this hook stem from the need to use functions belonging to template-structured services that connect to the corresponding endpoints within the inSPICE server layer. The services required by the application are initialized when it is launched for the first time, and from that moment on they can be accessed globally from any point of the application.

1.1.1 Global Contexts. Styling and Multi-language support

Another important aspect to manage within the framework is the support for global configurations that allow defining both the language and the general style of a given part of the application. Indeed, each of the templates may wish to use a different visual style for the general-purpose components of the application, and additionally each museum may require its own style when deploying its own instance of inSPICE. On the other hand, the interface of a given template (and of the common parts of the framework) should be, in most cases, available in at least two different languages (the language of the country hosting the use case on which the template is based, in addition to English as the standard language), and again, the language options available could be specific to each of the independent instances of inSPICE.

In order to address the theming requirement, in the framework we make use of global contexts provided by styled-components (the library used to manage the project's styles). In summary, within the project there exists a component whose fundamental role is to act as a provider of a general theme context for all its child components (here context is interpreted in React terminology as a shared state to the component tree under the component that acts as provider), so that all components that use styled-components and are within the scope of this provider will have direct access to the theme variables.

The theme in question, that is, the state that is shared with all the child components of the provider, has the following signature:

```
interface IThemeContext {
  theme: keyof AvailableThemes;
  switchTheme: (theme: keyof AvailableThemes) => void;
}

export interface AvailableThemes {
  light: DefaultTheme;
  dark: DefaultTheme;
};
```

where AvailableThemes is an interface that declares the themes supported in this given instance of the application (in this example, light and dark). DefaultTheme in turn is an interface that declares a series of properties to be defined for each of the specific themes to be added to the application (such as background color or fonts to be used for each type of text). The shared state object is basically composed of a key that specifies the identifier of the theme to be used within the available themes object, and a theme change function by means of which any child component is able to update the global configuration. The theme provider itself can be seen in the following piece of code:

```
const Theme: React.FC = ({ children }) => {
  const { theme } = useContext(ThemeContext);
  return (
    <ThemeProvider theme={themes[theme]}>
      <GlobalStyle />
      {children}
    </ThemeProvider>
  );
};
```

where ThemeContext is a concrete React context created through a React.createContext call. On the other hand, the GlobalStyle element acts as a general style to be used throughout the application and serves as a base with global settings, which can later be overwritten by specific themes or styles. A usage example of these global themes from a particular component could be:

```
export const InputText = styled.input`
  font-size: 0.9em;
  font-weight: 200;
  font-family: ${props => props.theme.contentFont};
  line-height: 135%;
  width: 85%;
  margin-top: 0.2em;
  color: ${props => props.theme.textColor};
  border: none;
  border-bottom: 2px solid transparent;
  outline: none;
  padding: 2px 0;
  background-color: transparent;
  transition: all 0s;
`;
```

Here the access to the theming is done from props.theme, where styled-components injects the theme set in the Theme component shown above. The theme changes in the current version of the framework are performed through the Header component, where the *switchTheme* function is used to alternate between light and dark theme to exemplify the use of this functionality.

The approach used to manage the problem of languages and content internationalization is very similar to the one used in the case of global theming and styles, in that it essentially uses a component that exposes a global context with an object designed to manage the key-value mappings and translations for the language selected at the time, which its children can access whenever they need to. In this case, the provider is given by a component supplied by the react-i18next[2] library, in charge of implementing the i18n internationalization mechanism within the application. The usage model for this context is, however, different from the approach used in the case of the styling. At the configuration level, it is first necessary to include key-value json files for each of the namespaces to be supported (in this particular case, one for the general elements of the application, and another for each template or activity to be implemented), and one for each language to be supported within that namespace. A fragment of such a file for the "gamGame" namespace, for English and Italian languages could be:

```
{ // gamGame/en.json
  ...
  "youreAboutToChooseArtwork": "You're about to choose {{artwork}} as your
   next artwork. Continue with this choice?",
  ...
}

{ // gamGame/it.json
```

[2] https://react.i18next.com

```
  ...
  "youreAboutToChooseArtwork": "Hai scelto {{artwork}}. Continuare?",
  ...
}
```

Secondly, at the usage level, it is necessary to utilize the useTranslation hook from any component that wishes to access the internationalization services before being able to make use of the translation or language change functions. This hook must be declared with the name of the namespace on which to instantiate the relevant translation functions. On a practical level, a sample code for a component within the GAM Game use case might look like this:

```
const { t } = useTranslation('gamGame');

// returns interpolated translation in selected language
const localisedText = t('youreAboutToChooseArtwork',
      { artwork: artwork.title });
```

As can be observed in the above code, it is sufficient to call the function t obtained from the globally exposed internationalization hook to retrieve a string localized to the currently selected language. Note that this infrastructure also supports interpolation mechanisms, by means of which it is possible to include variables within localized strings, and even apply rules concerning gender, pluralization, etc.

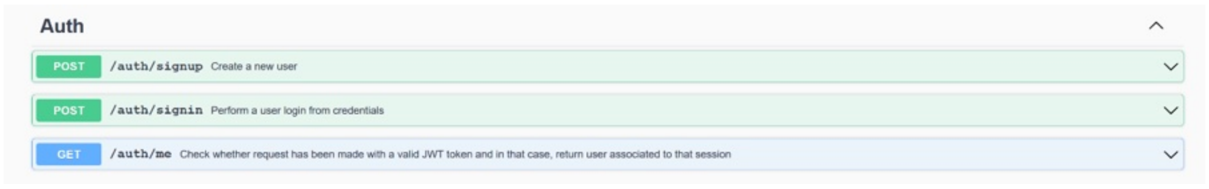
2.2. Interaction between Client and Server projects

Although we will not go into detail about the implementation or architecture of the inSPICE server, since it is a fairly standard backend from an architectural point of view, it is worth clarifying some considerations about its use, especially with regard to authentication and performance.

One of the major problems that arise when trying to access an API for which some kind of key or user authentication protocol is required from a JS-based frontend is that, if one wishes to be able to make the request from the web application itself, the keys to be used must be either supplied by the users themselves from the browser at some point in the flow (which could only be sensible if all users were administrators, but museum users need not have API keys to access the LDH), or set as compiler variables that are "saved" in the final code that is served to the client in the browser (which raises a security concern by essentially rendering these credentials public).

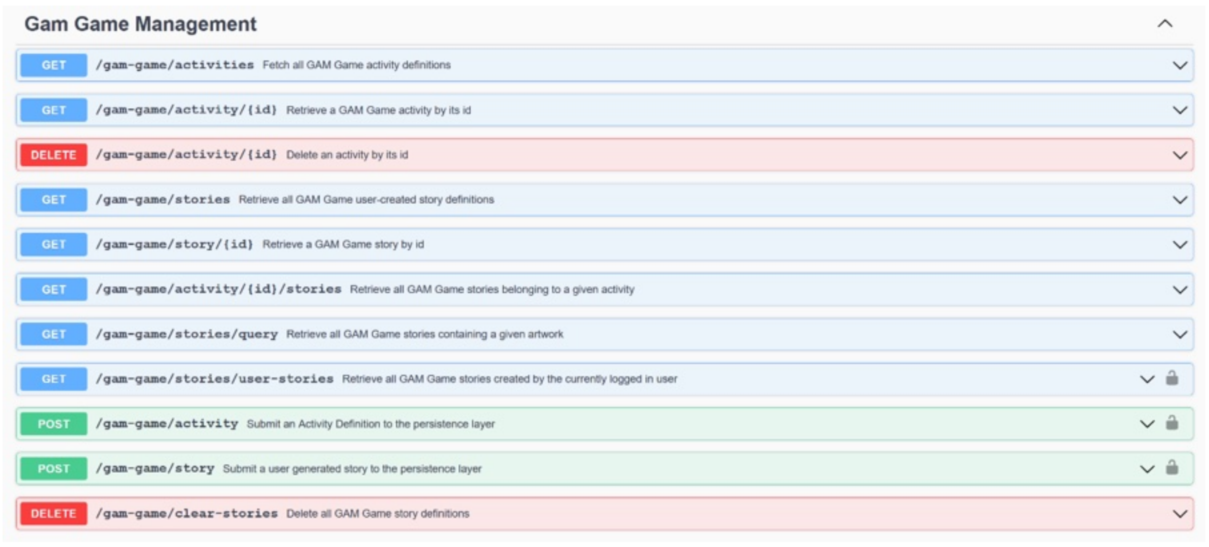
The simplest way to manage this problem is to delegate all the access logic to protected services, such as LDH, to a dedicated server with its own credential management system and issuing of access tokens to users who are properly authenticated within the application. Thus, all the API keys for access to protected services are kept secure within the backend of our application, and the frontend is restricted to making requests directly to this server every time it needs to access any of its functionalities, without compromising confidential keys.

In this context, some endpoints are publicly available to any user regardless of whether or not they are logged into the application (for instance, obtaining a list of artworks accessible from an activity is considered a publicly accessible query), while others are restricted to authenticated users (viewing content generated by the current user, uploading one' s own content, etc.). Any flow limited to authenticated users is locked behind a GuardedRoute, an auxiliary component that wraps any component that requires authentication and redirects the user to the login page if it is detected that the user is not yet authenticated at rendering time. Additionally, the application exposes a global context intended to make the current user's authentication data (backend token, current user profile, etc) accessible, so that any component may retrieve this information when needed (e.g. to query user profile data in prompts). Some examples of inSPICE server endpoints, for authentication and GAM Game related services, may be found in Figure 4 and Figure 5, respectively.


Auth

POST `/auth/signup` Create a new user
POST `/auth/signin` Perform a user login from credentials
GET `/auth/me` Check whether request has been made with a valid JWT token and in that case, return user associated to that session

Figure 4


Gam Game Management

GET `/gam-game/activities` Fetch all GAM Game activity definitions
GET `/gam-game/activity/{id}` Retrieve a GAM Game activity by its id
DELETE `/gam-game/activity/{id}` Delete an activity by its id
GET `/gam-game/stories` Retrieve all GAM Game user-created story definitions
GET `/gam-game/story/{id}` Retrieve a GAM Game story by id
GET `/gam-game/activity/{id}/stories` Retrieve all GAM Game stories belonging to a given activity
GET `/gam-game/stories/query` Retrieve all GAM Game stories containing a given artwork
GET `/gam-game/stories/user-stories` Retrieve all GAM Game stories created by the currently logged in user
POST `/gam-game/activity` Submit an Activity Definition to the persistence layer
POST `/gam-game/story` Submit a user generated story to the persistence layer
DELETE `/gam-game/clear-stories` Delete all GAM Game story definitions

Figure 5

Authentication follows a user/password scheme, where the application makes the login request to the server with the credentials provided by the user and, if valid, receives a JWT token that can be used in subsequent requests to access protected services (and which is stored in the global context mentioned above). The user data, including the hash of the password set at registration, is kept in a MongoDB database managed by the same server, and is never included in the LDH, in order to keep this information as isolated as possible from the general data persistence. Entries in such a database must be kept consistent with entries in anonymized user profile collections in the LDH, using a common identifier in order to be able to access the information provided by that model when needed. An overview of the authentication process can be seen within the diagram in Figure 6.

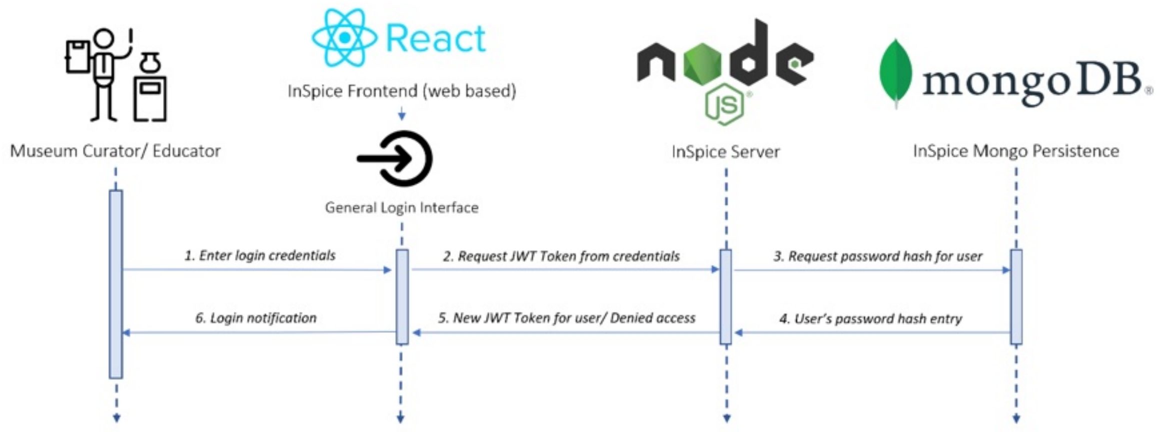

React
InSpice Frontend (web based)
Museum Curator/ Educator
General Login Interface
InSpice Server
InSpice Mongo Persistence
mongoDB
1. Enter login credentials
2. Request JWT Token from credentials
3. Request password hash for user
4. User's password hash entry
5. New JWT Token for user/ Denied access
6. Login notification

Figure 6

3. Use Cases included in inSPICE

Currently, the inSPICE framework includes working versions of the following modules and activity templates:

- GAM Game (Activity Template): configuration and consumption flows of GAM Game type activities, in which users can go through the museum exploring the information of the works included in the activity and engage with them through "stories", or sequences of linked interactions of users with museum artworks (through texts, emojis, tags). A user can create new stories as a form of interpretation of the works available in the activity and explore the stories of other visitors within instances of this activity. This activity template is associated with the GAM museum use case.
- Find Artwork (Activity Template): configuration and consumption flows of Find Artwork type activities, in which museum visitors can create their own treasure hunts from works chosen by the curators and play treasure hunts designed by other users of the application. In this sense, the activity can be divided into two parts: an interpretation phase in which visitors can configure their treasure hunts through a process in which they choose works from the collection, writing clues to help future players find those works, and "prizes" in audio or text format to be shown when players successfully find the chosen work; and a reflection phase in which visitors can decide to play other users' treasure hunts, being exposed in the process to their views and comments on the selected artworks. This activity template was developed as a first proof of concept for the Madrid Natural Science Museum use case.
- Multistage Form (Activity Template): configuration and consumption flows of Multistage Form type activities, in which museum visitors go through a series of stages with different prompts and inputs through forms in which they are guided through a process of interpretation and reflection on different texts and audiovisual materials. This activity template aims to generalize the use cases of Hecht and the configuration of scavenger hunts for the National Museum of Natural Sciences of Madrid, although this last case is still under development at the time of writing this document.
- IMMA Viewpoints (Activity not adhering to a template): non-instantiable activity that allows users to explore the IMMA collection and share their views on selected works with other users, as well as view comments from other users.
- Museum Catalogue Explorer (module): module aimed at facilitating the exploration of catalogues of artifacts or works associated with a given museum, as a mechanism to facilitate the task of the museum curator in the design of activity instances. There currently exists a demo of this functionality for the case of the National Museum of Natural Sciences.
- Activity dashboard (module): module designed to facilitate the exploration and creation of activity instances associated to the different templates described above. Through this flow, any identified user can view, create, duplicate, edit or delete activity instances created through one of the inSPICE templates (as long as they have the necessary permissions to do so), as well as search for activities already created by other museum curators by tags or template types.
- Support for user login and registration within the inSPICE Server.

The activity dashboard can be accessed at http://spice.padaonegames.com/dashboard which also gives access to example of instances of the supported use cases. Next Sections describe these modules in detail.

3.1. Creation Steps. Shared Steps

While not all of the flows included in inSPICE are directly configurable by museum curators, a subset of them can be considered "instantiable" in the sense that they expose a series of interfaces to facilitate the creation of specific activities based on certain base patterns. These flows are what we refer to as "activity templates" within inSPICE, and the activity instances based on them are objects based on a shared general activity schema:

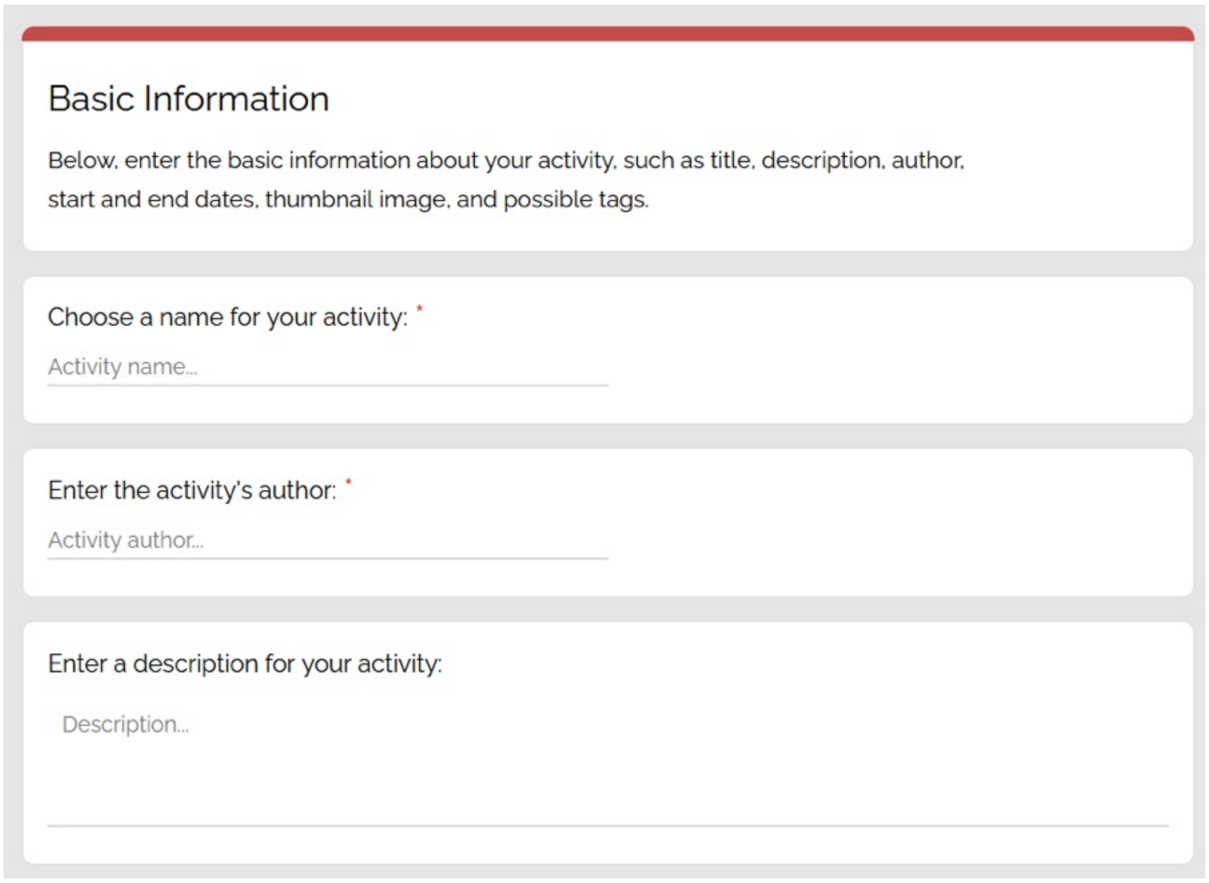
```
export interface ActivityInstance {
  activityType: SupportedActivity;
  activityTitle: string;
  activityAuthor: string;
  beginsOn: Date;
  endsOn: Date;
  _id: string;
  description?: string;
  imageSrc?: string;
  tags?: string[];
  intendedUsers?: IntendedUser[];
}

export const supportedActivities = [
  'Treasure Hunt',
  'GAM Game',
  'Multistage Form'
] as const;

export type SupportedActivity = typeof supportedActivities[number];
```

At the time of writing, the framework has 3 flows that can be considered "templatizable", with definitions for their activities following schemas that extend the above interface (Treasure Hunt, GAM Game and Multistage Form).

On the other hand, any creation flow of an activity definition that adheres to a template is given by a structure of "linked steps" using a custom React module that allows to define ordered pages constituting a configuration form. An example of the use of this structure for the case of the configuration of a GAM Game type of activity would be:

```
<Steps
  config={config}
  genState={activityDefinition}
  setGenState={setActivityDefinition}
>
  <Step title='Basic Information' component={ActivityInstanceBasicInfoStep} />
  <Step title='Stage Settings' component={ConfigureStageParamsStep} />
  <Step title='Select Artworks' component={SelectArtworksStep} />
</Steps>
```

In this context, a GAM Game type activity is defined through a form of 3 sequential steps (Basic Configuration, Stage Settings and Select Artworks), each of which modifies the general state of the definition, given by activityDefinition. The steps included in this way as components within Steps are always self-contained and are limited to querying and modifying the key-value pairs of the general state of the form, so that they can be reused between creation forms that share common items. The most obvious example of this sharing occurs with the ActivityInstanceBasicInfoStep step component, intended to "populate" all the fields present in the ActivityInstance interface mentioned above. As already mentioned, since all templates are derived from this interface, all activities will have at least the data included in it, and can therefore make use of this initial step to manage the completion of this data. This interface can be seen in figures Figure 7. Specify title, author and general description., Figure 8. Specify image thumbnail and tags. andFigure 9. Specify starting/closing dates for activity..


Basic Information

Below, enter the basic information about your activity, such as title, description, author, start and end dates, thumbnail image, and possible tags.

Choose a name for your activity: *

Activity name...

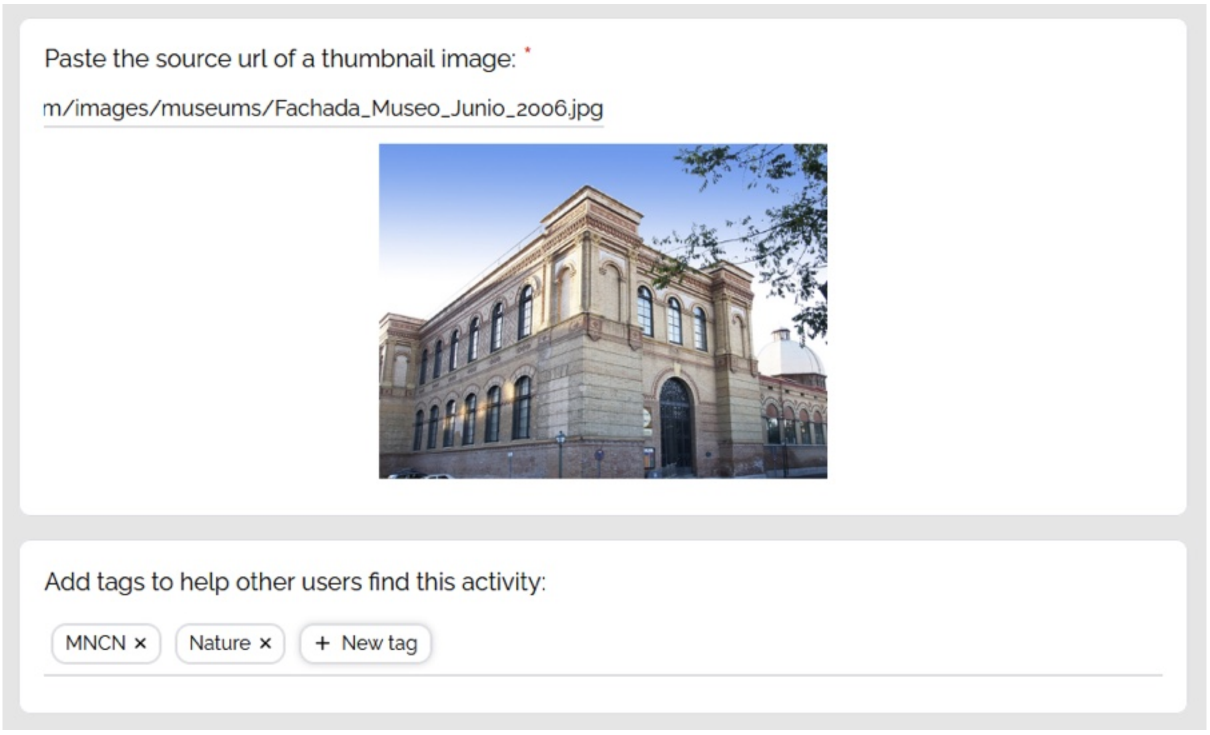
Enter the activity's author: *

Activity author...

Enter a description for your activity:

Description...

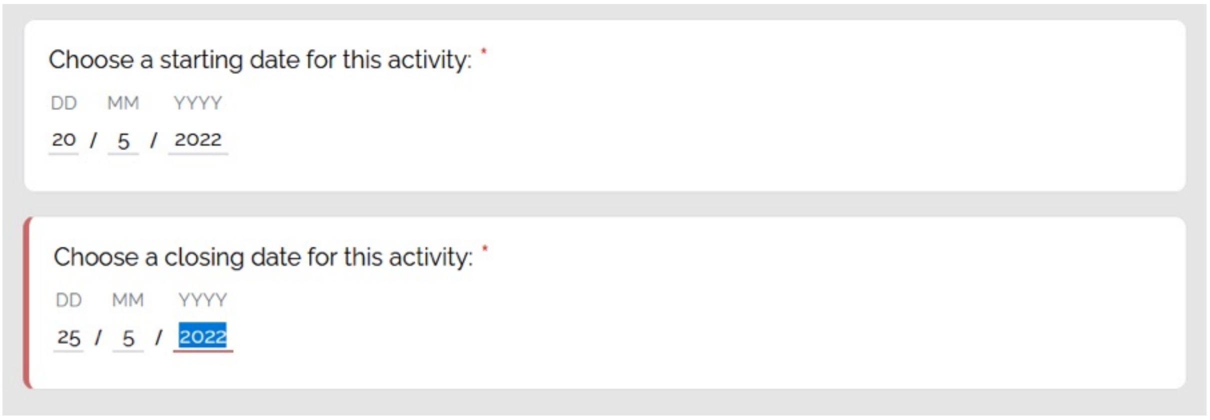
Figure 7. Specify title, author and general description.


Paste the source url of a thumbnail image: *

m/images/museums/Fachada_Museo_Junio_2006.jpg

Add tags to help other users find this activity:

Figure 8. Specify image thumbnail and tags.


Choose a starting date for this activity: *

DD MM YYYY
20 / 5 / 2022

Choose a closing date for this activity: *

DD MM YYYY
25 / 5 / 2022

Figure 9. Specify starting/closing dates for activity.

Additionally, the SelectArtworksStep step can be used to manage the construction of a list of works to be selected from those available in the catalogue of the museum on which the activity is to be held when the corresponding template requires a certain selection step (Treasure Hunts/ Find Artwork and GAM Game make use of this). This component can be found in Figure 10. General Select Artworks Step..

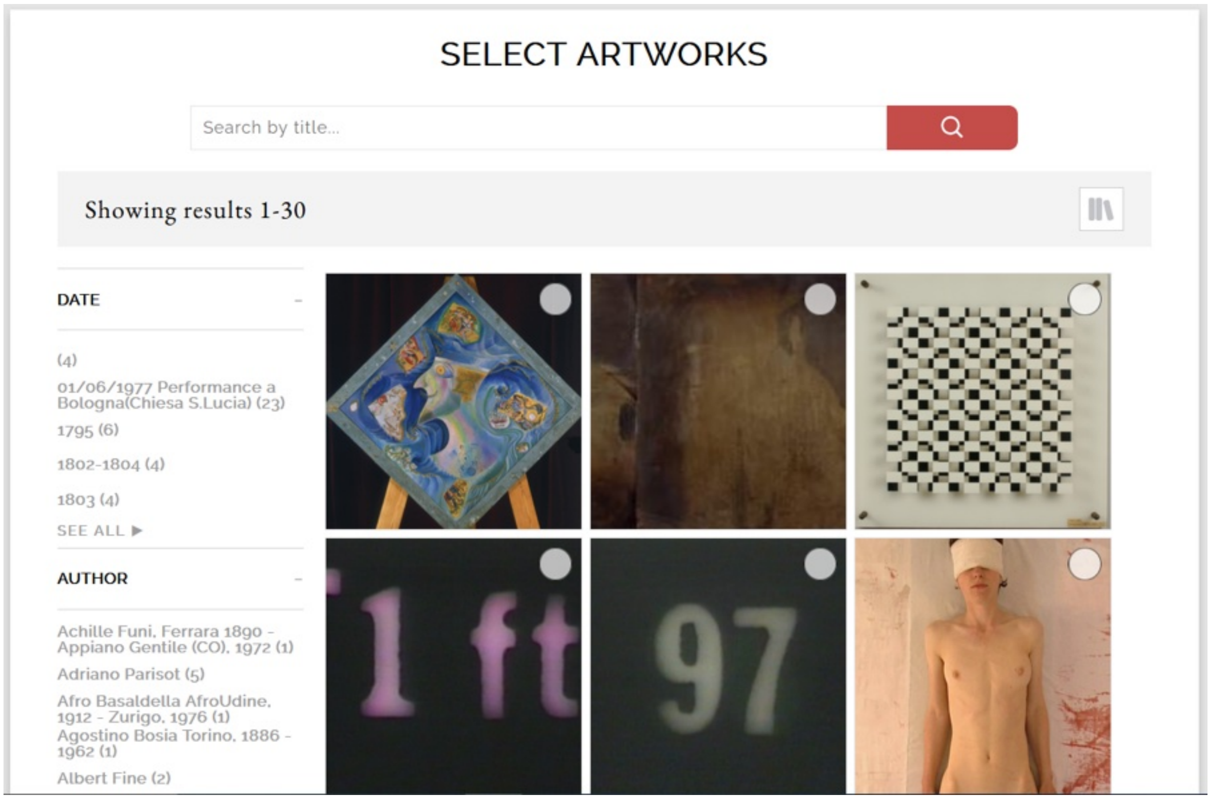

SELECT ARTWORKS
Search by title...
Showing results 1-30
DATE
(4)
01/06/1977 Performance a Bologna(Chiesa S.Lucia) (23)
1795 (6)
1802-1804 (4)
1803 (4)
SEE ALL ▶
AUTHOR
Achille Funi, Ferrara 1890 - Appiano Gentile (CO), 1972 (1)
Adriano Parisot (5)
Afro Basaldella AfroUdine, 1912 - Zurigo, 1976 (1)
Agostino Bosia Torino, 1886 - 1962 (1)
Albert Fine (2)

Figure 10. General Select Artworks Step.

3.2. (MNCN) Treasure Hunt Authoring

3.2.1. Introduction and Background

The activity represented by this template, which we will refer to as **Find Artwork** from now on, proposes a system in which museum visitors and users can create customized treasure hunts based on a series of artworks specified by museum curators, incorporating clues and personal multimedia material for each of the stages and artworks in their game. In this way, the activity is naturally framed as a form of collecting as well as a potential form of interpretation (see D2.1, D2.2 and D2.3), depending on how it is specified by the museum curator. The treasure hunts created by users can then be completed by other users and visitors, and analyzed by museum curators to draw conclusions and statistics of various kinds. With this background, the general process of the activity can be summarized in the following fundamental points:

The museum curator decides to create a new activity of type *Find Artwork*, either from scratch or by editing an existing activity of the same type in the system. To do this, they use a **creation interface**, in which the application guides them through a series of steps in which the curator must specify the necessary fields and settings to define the activity to be created: start and end date, types of inputs to be allowed, works that will be available for selection when a user creates a new treasure hunt, minimum and maximum number of works to be selected, etc. Once this data has been completed, the definition is sent to the persistence layer for future use.

During the period in which the activity is available, a user of the system can decide to take part in it and create a new treasure hunt. This can happen for instance from a link provided via the museum's activity website, or shared through social channels. In principle, the user must be registered in the system to be able to create a new treasure hunt. To do this, the user will utilize an **activity consumption interface**, in which the application guides the user through a series of panels in which they must add stages to the game following the restrictions set by the curator when defining the activity. For each stage, the user must select an artwork from the available ones, and write a series of short clues that help the player to recognize the selected artwork among the given set of works. Additionally, the user must enter some form of multimedia materials (text, audio, images, etc.), according to the curator's specifications, with a commentary or reflection on the selected piece. This material will be shown to the treasure hunt players upon successfully finding the appropriate artwork in the corresponding phase. Once a treasure hunt has been defined in its entirety, the definition is sent to the persistence layer so that it can be analyzed in the citizen curation process or played by other users.

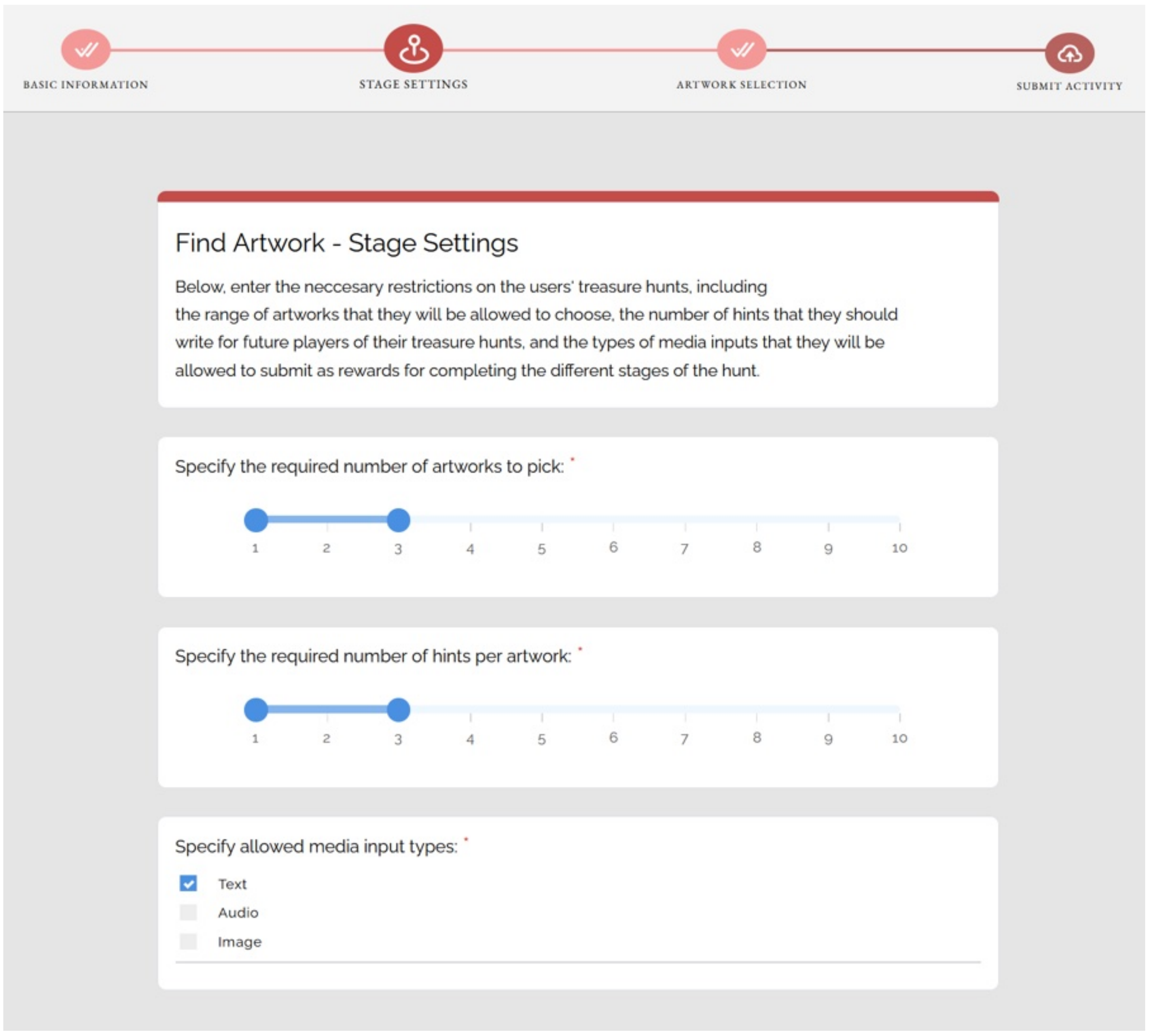
At that moment, the user receives a public link that can be shared with friends and acquaintances through social channels to allow them to play their treasure hunt. To do this, players will use a **treasure hunt consumption interface**, in which the application guides the player through the different phases defined by the user who created the treasure hunt.

3.2.2. Creation Interface - Main Flow

The creation interface presents 3 fundamental steps, 2 of which are shared with other activity templates (Basic Activity Information and Artwork Selection). The final goal of this flow is the creation of a definition object for an activity instance that conforms to the Find Artwork template format, derived from Activity Instance and therefore sharing all the minimum definition fields of a general activity (title, description, author, start and end dates, thumbnail, etc). The format of a valid definition object is given by the following interface:

```
export interface FindArtworkActivityDefinition extends ActivityInstance {
  activityType: 'Treasure Hunt';
  minStages: number;
  maxStages: number;
  minCluesPerStage: number;
  maxCluesPerStage: number;
  allowedInputs: AllowedInputs[]; // 'Text' | 'Audio' | 'Image'
  huntDefinitionsDatasetUuid: string;
  artworksDatasetUuid: string;
  artworks: string[];
}
```

The fields regarding the min/max stages allowed, minimum/maximum number of clues per stage, and types of inputs that are acceptable by treasure hunts creators when specifying what to show players of their creations upon successful completion of a stage are filled in through the "stage settings" step interface, which can be found in Figure 11.



Find Artwork - Stage Settings

Below, enter the neccesary restrictions on the users' treasure hunts, including the range of artworks that they will be allowed to choose, the number of hints that they should write for future players of their treasure hunts, and the types of media inputs that they will be allowed to submit as rewards for completing the different stages of the hunt.

Specify the required number of artworks to pick: *

1 2 3 4 5 6 7 8 9 10

Specify the required number of hints per artwork: *

1 2 3 4 5 6 7 8 9 10

Specify allowed media input types: *

- [x] Text
- [ ] Audio
- [ ] Image

Figure 11

The fields defined in the Activity Instance interface are populated through the first shared step interface, "Basic Information", common to all activity instance creation flows, while the artworks field is populated in the "Artwork Selection" step interface, which is again common to all activity instance creation flows that present some kind of general selection of artworks or artifacts over which to define the activity. The elements of this array are strings containing the id of the referenced artworks as stored within the persistence layer of the associated collection in the LDH.

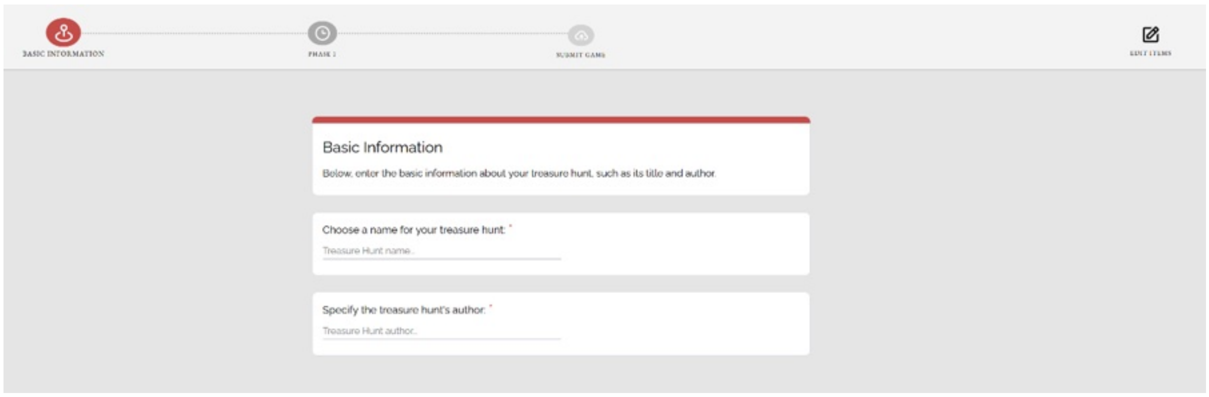
Once it is detected that all mandatory fields have been filled in by the user, the interface proceeds to enable a "Submit Activity" button, which calls the corresponding request from the Find Artwork management service to store the resulting definition in the persistence layer.

3.2.3. Activity Consumption Interface - Main Flow (Create Treasure Hunt)

The purpose of the create treasure hunt interface is to provide a workflow for visitors to engage with an instance of a Find Artwork activity, thus allowing them to create their own treasure hunts to be used later on in the citizen curation phase. In more concrete terms, the ultimate goal of this interface is to facilitate the creation of a treasure hunt definition object that adheres to the following format:

```
export interface TreasureHuntDefinition {
  _id: string;
  treasureHuntAuthor: string;
  treasureHuntTitle?: string;
  activityId: string;
  stages: StageData[];
}

export interface StageData {
  artworkId: string; // selected artwork's Id in LDH
  clues: string[]; // clues/ hints to show to the player after completing the
stage
  multimediaData: TreasureHuntMutimediaData[]; // rewards to show after
finding the artwork
}

export type TreasureHuntMutimediaData =
  | { kind: 'Text', text: string }
  | { kind: 'Audio', src: string }
  | { kind: 'Image', src: string }
  ;
```

The flow of creating a treasure hunt begins with a first step in which the user enters the most basic information of their treasure hunt, consisting of the name they wish to give to their creation and a user identifier (treasureHuntTitle and treasureHuntAuthor fields, respectively). This very simple step can be found in Figure 12.



Basic Information

Below, enter the basic information about your treasure hunt, such as its title and author.

Choose a name for your treasure hunt: *

Treasure Hunt name..

Specify the treasure hunt's author: *

Treasure Hunt author..

Figure 12

The next step, denoted by stages named "PHASE X", guide the application user through the process of specifying the content that each of the stages of their treasure hunt will contain. Initially, the interface displays a number of mandatory phases equivalent to that specified by the museum curator in the minStages field of their activity definition, and these can be added or removed (as long as the number of phases after modification remains within the allowed limits) by means of the "Edit Items" option. This enables an alternative editing mode on the treasure hunt layout, as shown in Figure 5.

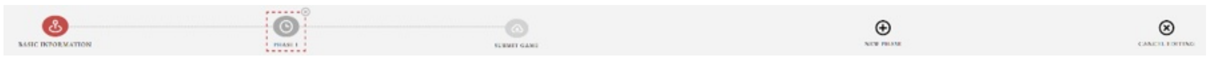


Figure 10

Each of these phases corresponds to an element of the *stages* array in the definition of a treasure hunt, and can be built following a sequence of sub-phases, accessible by clicking on any of the "PHASE X" selectors:

1. Selection of artwork to be found. Used to assign value to field *artworkId* within the treasure hunt definition. User may choose any artwork from the *artworks* array in the activity definition.

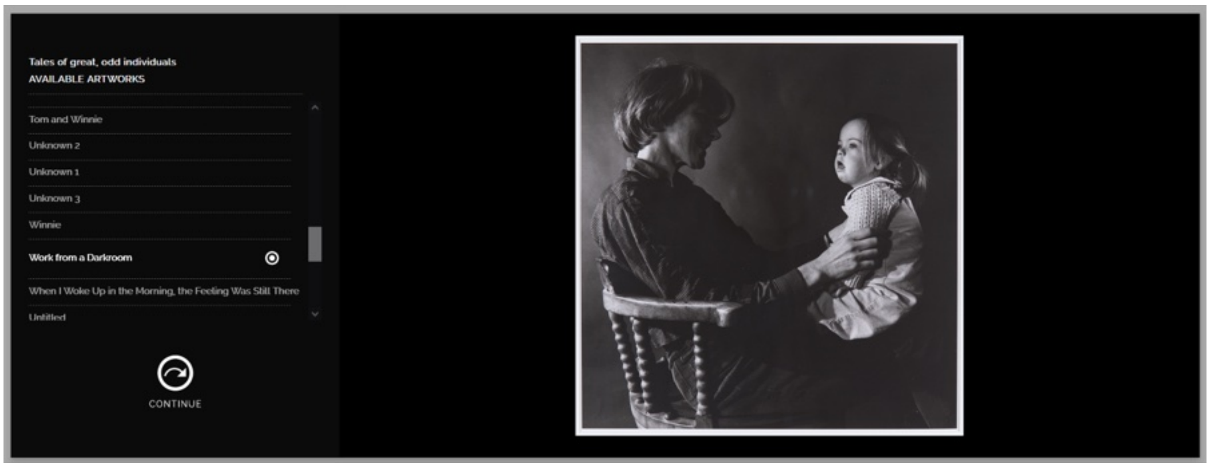
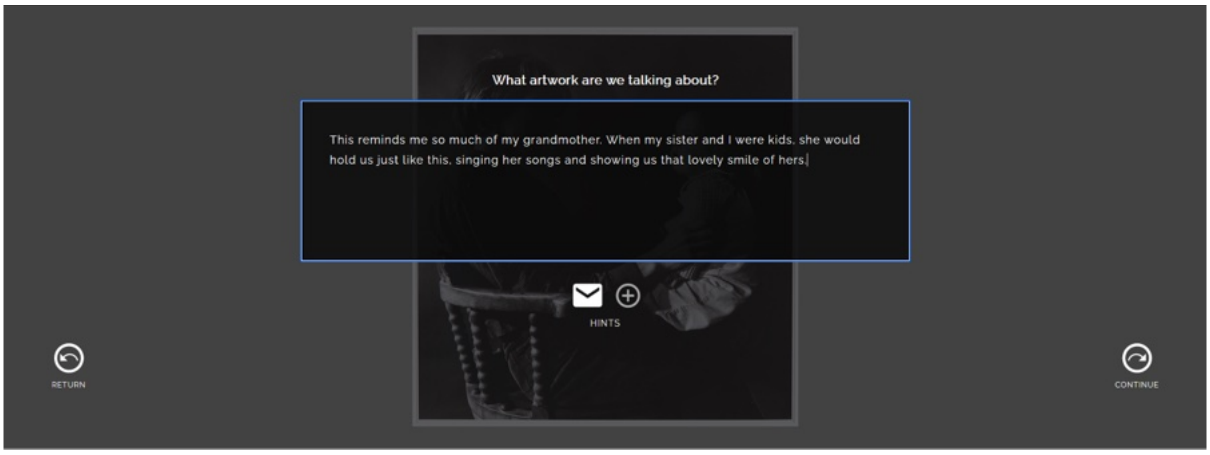


Figure 13

2. Introduction and composition of clues to be shown to the player of the treasure hunt. Used to assign values to field *clues* within the treasure hunt definition. Minimum and maximum number of clues for activity apply here.

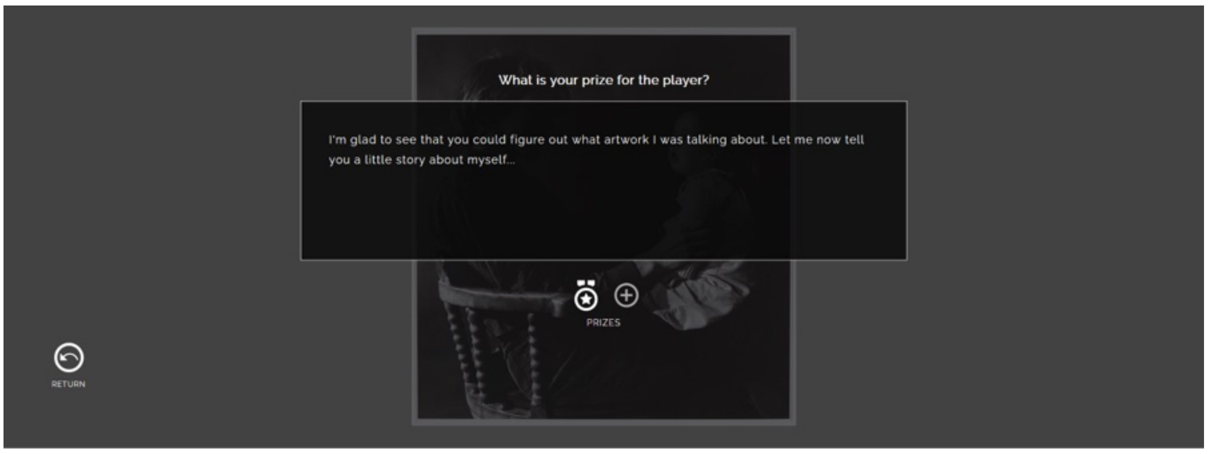

What artwork are we talking about?

This reminds me so much of my grandmother. When my sister and I were kids, she would hold us just like this, singing her songs and showing us that lovely smile of hers.

Figure 14

3. Introduction and composition of the multimedia content that the player will see upon completion of the phase (when finding the artwork). Used to assign values to field *multimediaData* within the treasure hunt definition. The example shown in figure 9

would correspond to an activity where *allowedInputs* is set to a single-value list containing the type "Text", with a minimum length of one item.



I'm glad to see that you could figure out what artwork I was talking about. Let me now tell you a little story about myself...

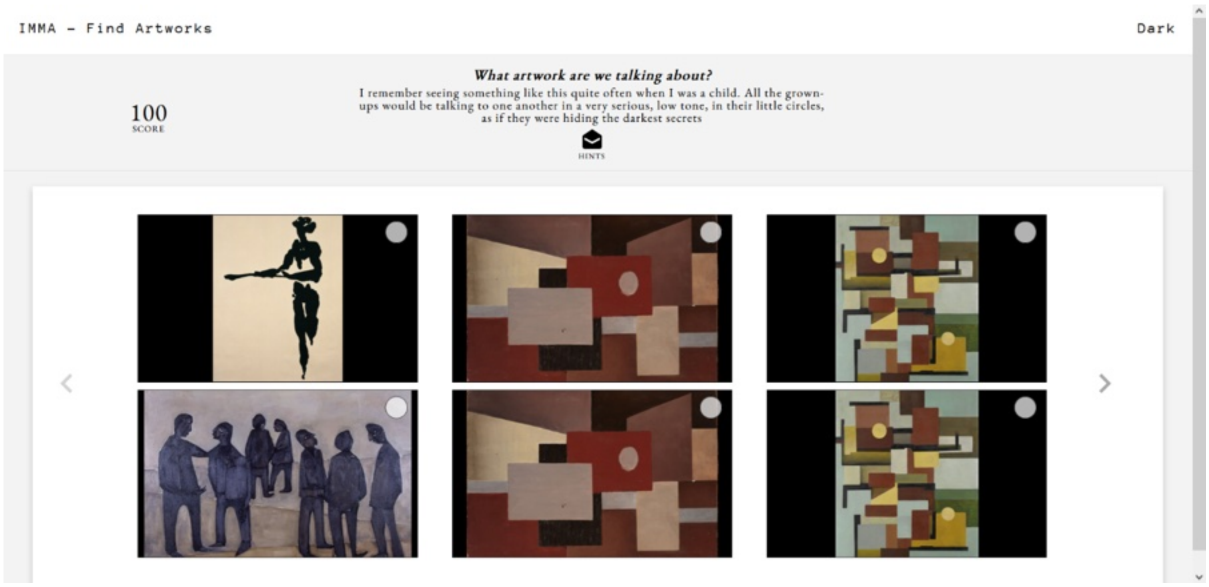
Figure 15

Once it is detected that all mandatory fields have been filled in by the user (basic information and complete definitions for all the required stages) the interface proceeds to enable a "Submit Game" button, which calls the corresponding request from the Find Artwork management service to store the resulting definition in the persistence layer.

3.2.4. Treasure Hunt Consumption Interface - Play Treasure Hunt

The purpose of the play treasure hunt interface is to provide a workflow for visitors to engage with a treasure hunt that has been previously created by another user, and thus be able to try the experience along with the associated content. This interface is not intended to generate any type of definition record, as was the case in the two previous flows, and is simply meant to guide the user through the interactive experience. The flow has the following fundamental stages:

1. Presentation of the works from which the user will be able to choose in each of the game rounds, together with the first hint of the phase. In this context, the user will be able to additionally visualize a scoreboard with the number of points accumulated so far, as well as a panel with the available clues, the first one unlocked by default, and the rest locked behind a cost in game points.


IMMA - Find Artworks

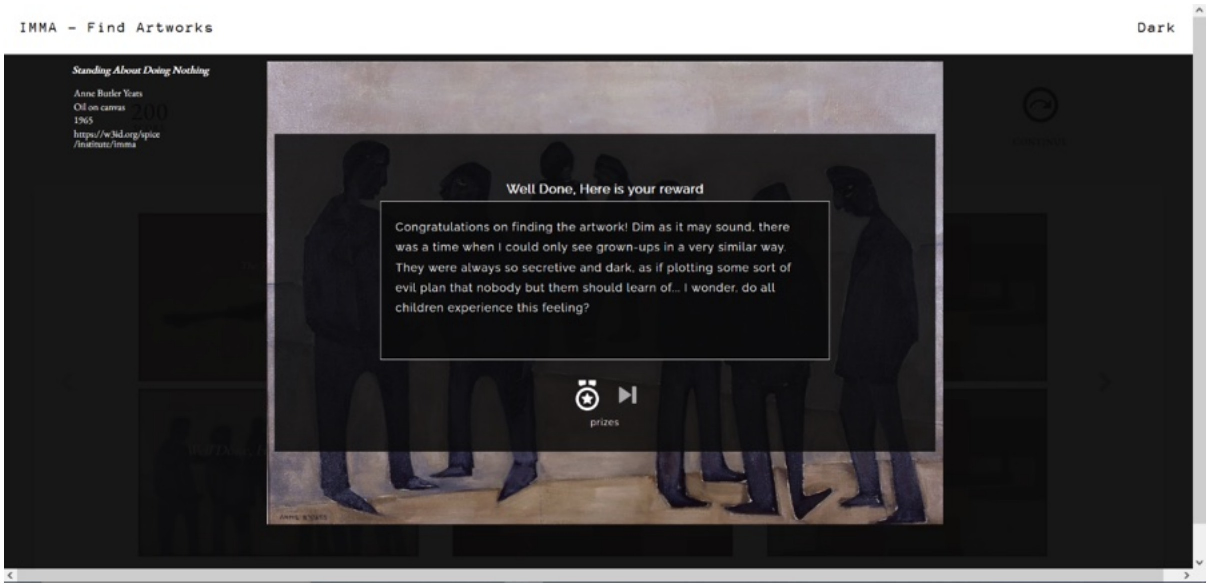
Dark

100
SCORE

What artwork are we talking about?
I remember seeing something like this quite often when I was a child. All the grown-ups would be talking to one another in a very serious, low tone, in their little circles, as if they were hiding the darkest secrets

Figure 16

2. Selection of works by the user as an attempt to figure out which piece is referred to by the clues provided. At this point the user can choose an incorrect piece, thereby deducting points from their scorer, request an additional clue in exchange for a penalty in the form of score points as indicated under the icon associated with that particular clue, or select the correct piece, which will trigger the flipping of all available pieces (and a score increase) and enable the option to go on to consume the content set as a prize by the designer of the treasure hunt.


IMMA - Find Artworks

Dark

Standing About Doing Nothing
Anne Butler Yeats
Oil on canvas
1965
https://w3id.org/spice/institute/imma

Well Done, Here is your reward

Congratulations on finding the artwork! Dim as it may sound, there was a time when I could only see grown-ups in a very similar way. They were always so secretive and dark, as if plotting some sort of evil plan that nobody but them should learn of... I wonder, do all children experience this feeling?

Figure 17

3. Consumption of multimedia content provided by the creator of the treasure hunt. By clicking on the prize icon of the correct work, the user will be able to see a list of multimedia elements or texts with the corresponding content. At the end of this

visualization, the user can proceed to the next stage of the treasure hunt by clicking on the “Continue” button.

3.2.5. Definitions management and access to resources

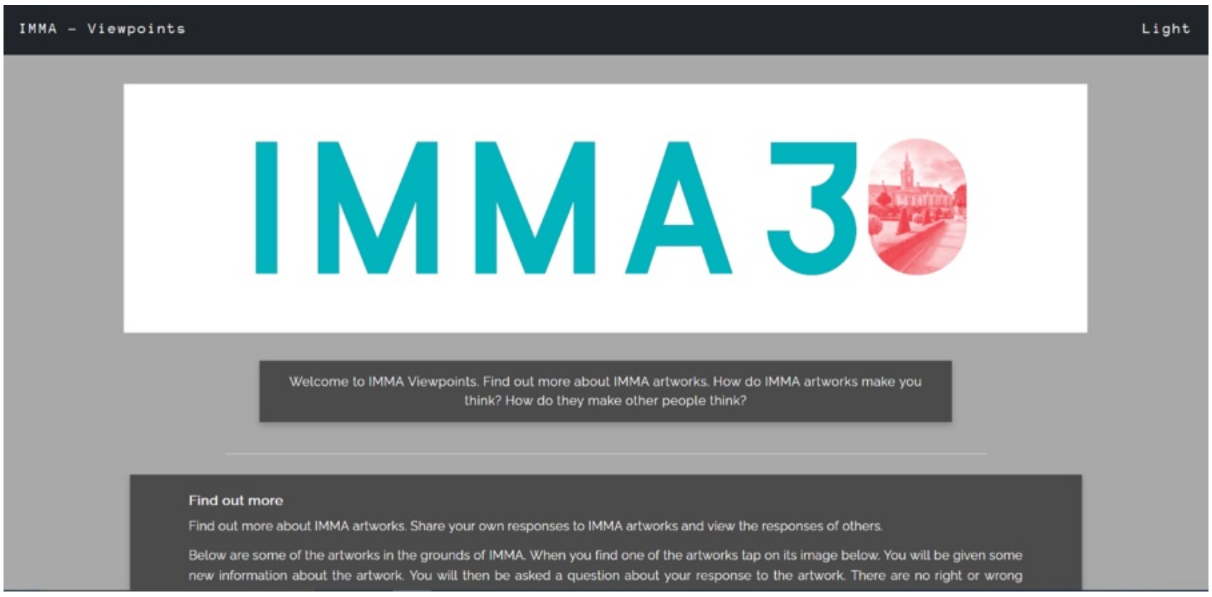
Each of the above user flows requires an activity or treasure hunt definition, respectively, in order to operate. Architecture-wise, what happens is that the higher-order components associated with each of these flows are linked to paths within the web application that include information regarding the id of the definition object to be consumed at the corresponding time. Thus, for example, to access the flow for creating a treasure hunt based on an activity instance of type find artwork with id "*6203b48b1ea1d52ccf1c6da9*", the relative path “*/find-artwork/consumer/create/6203b48b1ea1ea1d52ccf1c6da9*" would have to be used. In this path, the associated component would proceed to make a request through the Find Artwork service to the inSPICE server, which in turn would query the LDH to determine if an instance with that id exists in the relevant collection. If so, the object will be sent back to the client side, where it will be used to generate the interfaces described above with the specifications of the downloaded file.

3.3. (IMMA) Viewpoints

3.3.1. Activity Overview

The activity represented by this template, which we will refer to as **Viewpoints** from now on, proposes a system in which museum visitors and users can contribute their views on a selection of works in the collection, as well as explore the views of other users on the same works. This is done by means of a fixed list of standardized questions/answers to which users can react with the aim of providing a guided entry point when drafting their views. With this background, the general process of the activity can be summarized in the following fundamental points:

- The user of the activity decides to enter the web application and explore the different works included in it. To do this, the user engages with an exploration interface, in which the application presents a brief explanation of the process to follow (Figure 18) and displays a panel with thumbnails of the different works with which the user will be able to interact (Figure 19). The user may interact with any of these pieces by clicking on its thumbnail. In addition, placing the cursor over any of the thumbnails displays a semi-transparent panel on top of it with the basic data of the work (title, author, date).



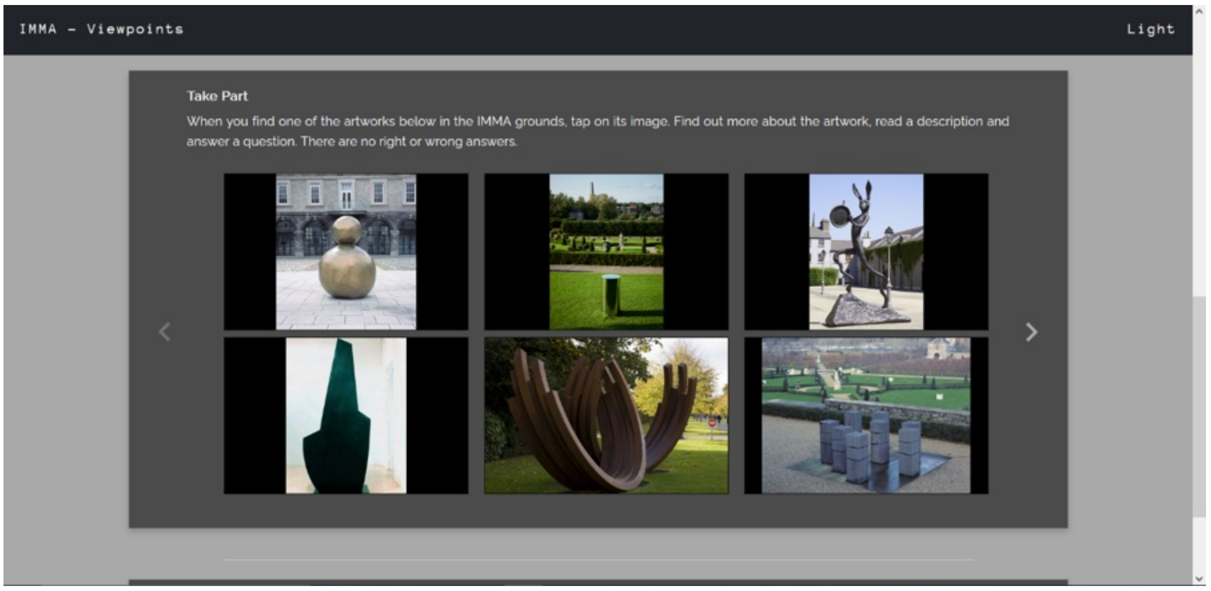
Welcome to IMMA Viewpoints. Find out more about IMMA artworks. How do IMMA artworks make you think? How do they make other people think?

Find out more

Find out more about IMMA artworks. Share your own responses to IMMA artworks and view the responses of others.

Below are some of the artworks in the grounds of IMMA. When you find one of the artworks tap on its image below. You will be given some new information about the artwork. You will then be asked a question about your response to the artwork. There are no right or wrong

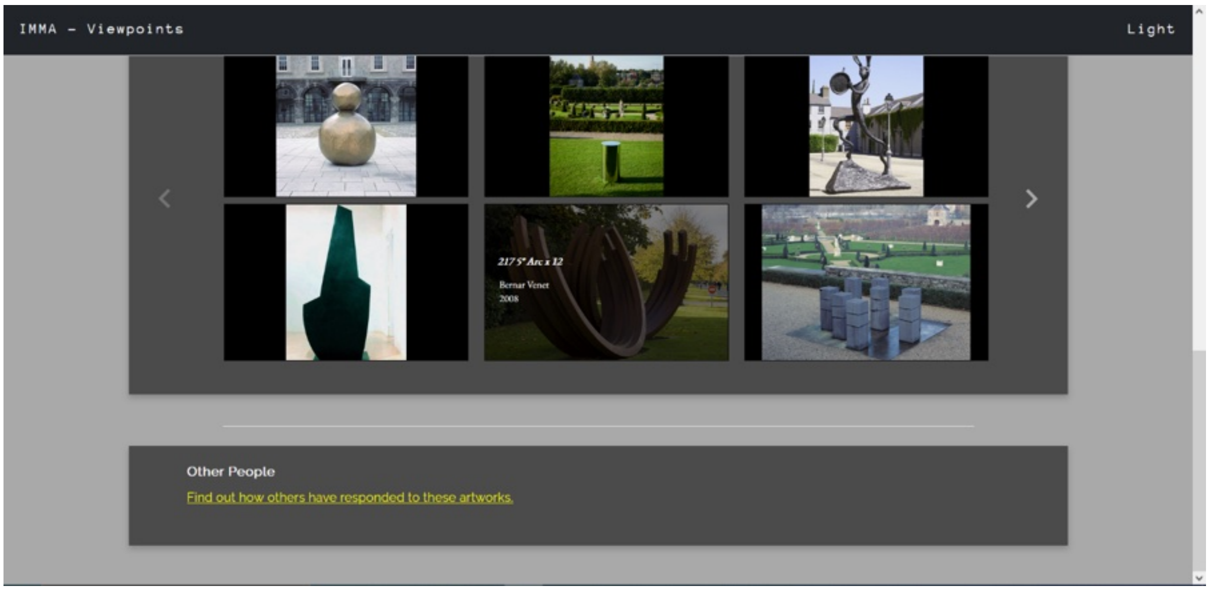
Figure 18



Take Part

When you find one of the artworks below in the IMMA grounds, tap on its image. Find out more about the artwork, read a description and answer a question. There are no right or wrong answers.

Figure 19

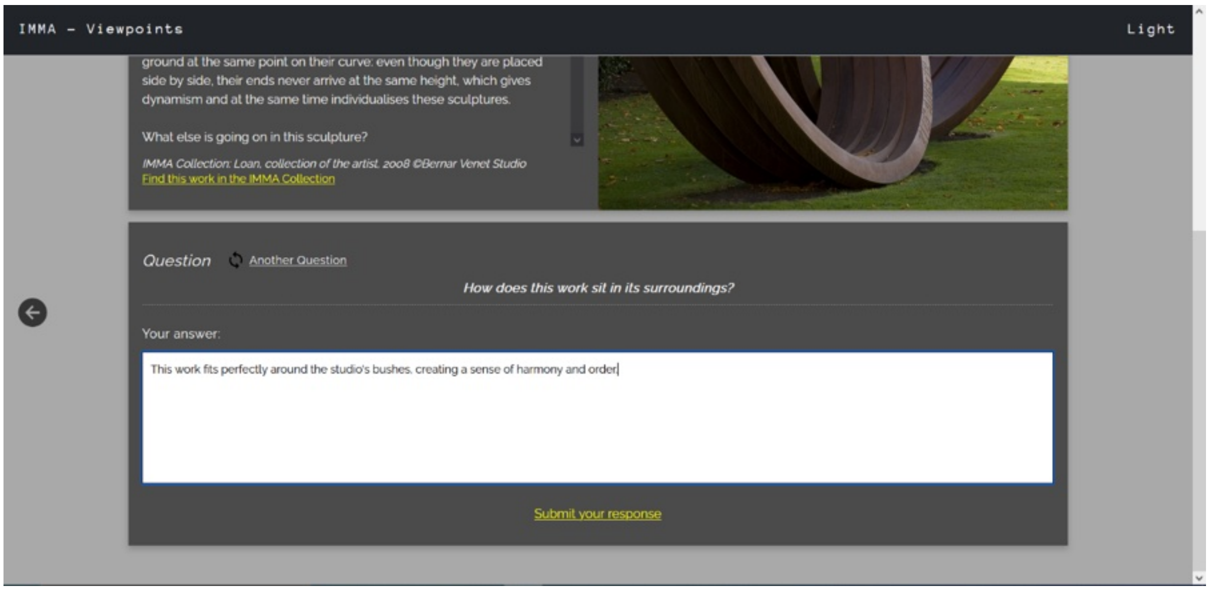


Other People

Find out how others have responded to these artworks.

Figure 20

- Upon entering the artwork's page, the application displays a first component in which the most important details of the piece are condensed through a brief text and/or an audio recording with any explanations that may be deemed pertinent. This also includes a larger and more detailed image of the work than that of the thumbnail in the previous interface, as well as information regarding the location, author, title, date, or materials used to create the work, among other details.



ground at the same point on their curve: even though they are placed side by side, their ends never arrive at the same height, which gives dynamism and at the same time individualises these sculptures.

What else is going on in this sculpture?

IMMA Collection: Loan, collection of the artist. 2008 ©Bernar Venet Studio

Find this work in the IMMA Collection

Question Another Question

How does this work sit in its surroundings?

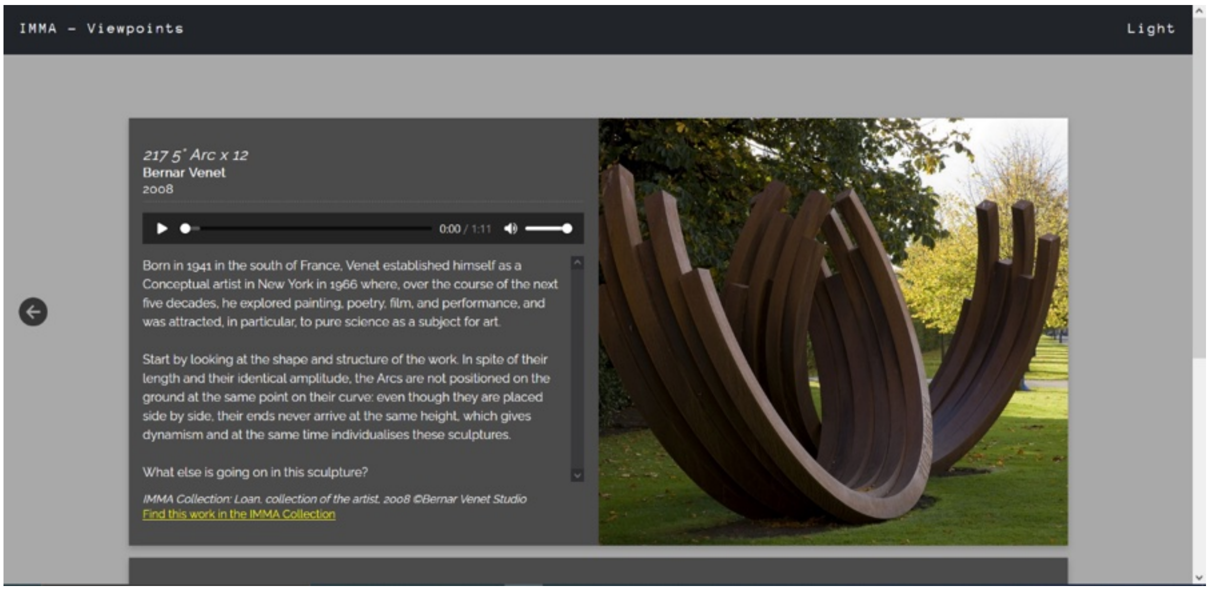
Your answer:

This work fits perfectly around the studio's bushes, creating a sense of harmony and order

Submit your response

Figure 21

- At the bottom of this artwork details screen, it is possible to find an interactive panel that randomly displays standardized questions that seek to elicit the user's opinion or point of view on the artwork, with the option to load a new question if the user is not sufficiently comfortable with the current prompt. The user is then able to use the text box under the "Your Answer" section to enter a thought in response to the question being asked. The answers entered here are sent to the persistence layer for future consumption and analysis.



217 5˚ Arc x 12
Bernar Venet
2008

Born in 1941 in the south of France, Venet established himself as a Conceptual artist in New York in 1966 where, over the course of the next five decades, he explored painting, poetry, film, and performance, and was attracted, in particular, to pure science as a subject for art.

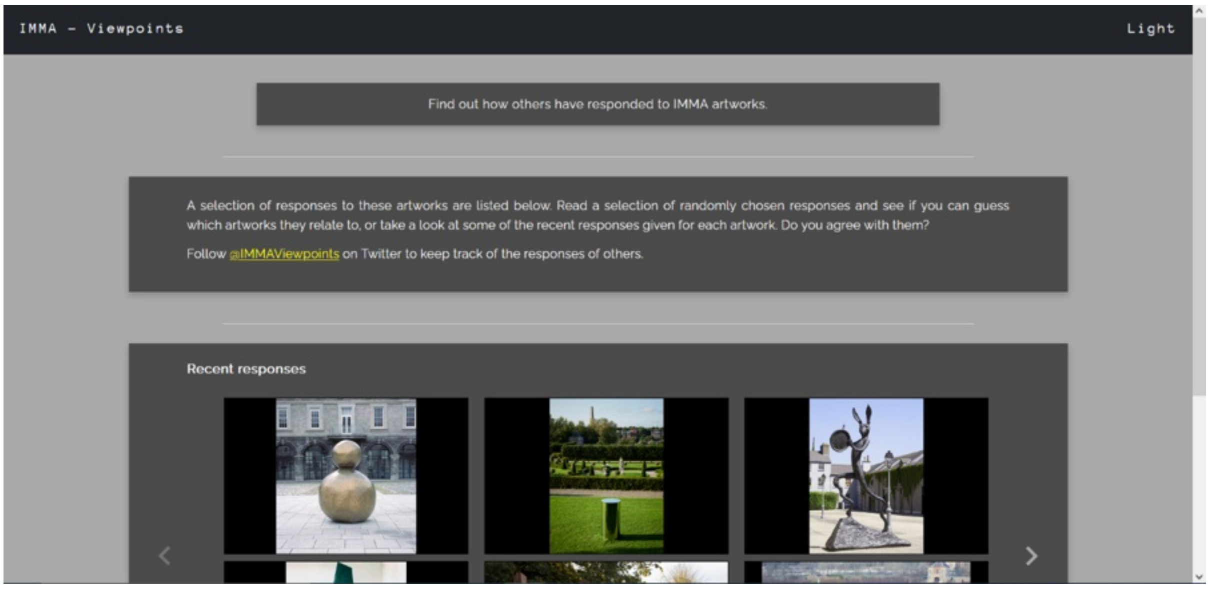
Start by looking at the shape and structure of the work. In spite of their length and their identical amplitude, the Arcs are not positioned on the ground at the same point on their curve: even though they are placed side by side, their ends never arrive at the same height, which gives dynamism and at the same time individualises these sculptures.

What else is going on in this sculpture?

IMMA Collection: Loan, collection of the artist, 2008 ©Bernar Venet Studio
Find this work in the IMMA Collection

Figure 22

- Additionally, from the home screen it is possible to access an additional flow intended to present all user interactions with the works in the collection ("Find out how others have responded to these artworks"). In the associated interface, the user may visualize a new artwork selector, again by means of thumbnails, so that clicking on each one of them opens a popup with a list of question-answer style cards including the history of interactions of other users with the selected artwork.



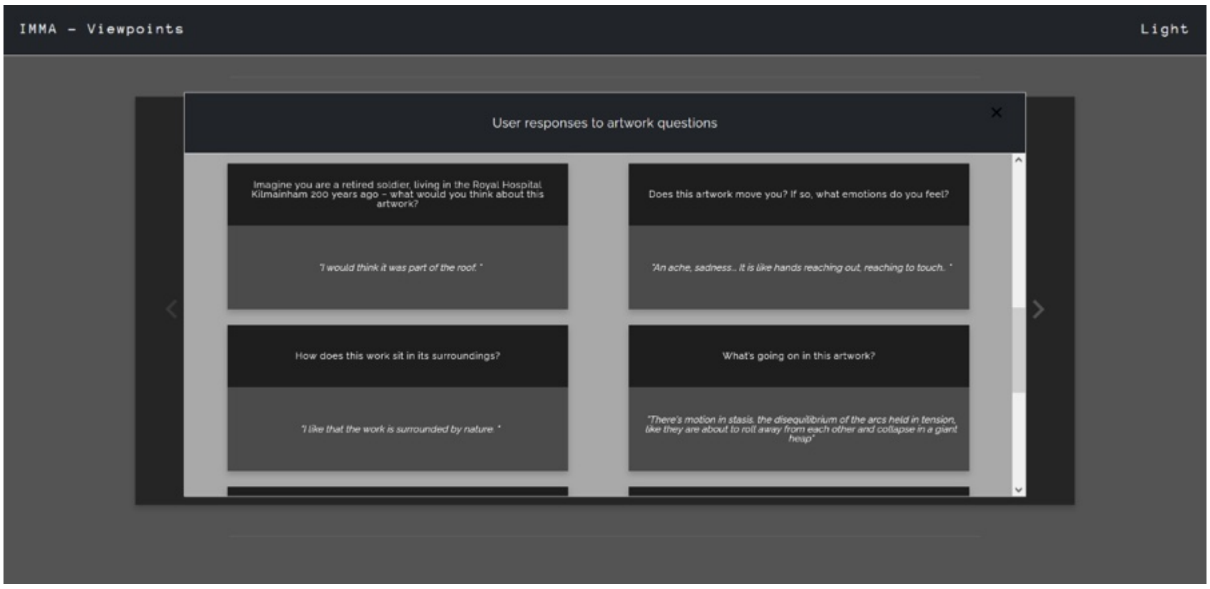
Find out how others have responded to IMMA artworks.

A selection of responses to these artworks are listed below. Read a selection of randomly chosen responses and see if you can guess which artworks they relate to, or take a look at some of the recent responses given for each artwork. Do you agree with them?

Follow @IMMAViewpoints on Twitter to keep track of the responses of others.

Recent responses

Figure 23



User responses to artwork questions

Imagine you are a retired soldier, living in the Royal Hospital Kilmainham 200 years ago – what would you think about this artwork?

"I would think it was part of the roof. "

Does this artwork move you? If so, what emotions do you feel?

"An ache, sadness... It is like hands reaching out, reaching to touch. "

How does this work sit in its surroundings?

"I like that the work is surrounded by nature. "

What's going on in this artwork?

"There's motion in stasis. the disequilibrium of the arcs held in tension, like they are about to roll away from each other and collapse in a giant heap"

Figure 24

3.4. (GAM) Gam Game

3.4.1. Introduction and Background

The activity represented by this template, which we will refer to as **GAM Game** from now on, revolves around the notion of storytelling. Through this activity type, users can create short stories by collecting and sequencing the artworks from the museum collection, and add a personal response to each of the artworks in the story. Stories can be created at any time before, during and after the museum visit.

Personal responses can consist of simple *tags*, text *comments* (created from a catalogue of templates: [the artwork] makes me feel …, [the artwork] reminds me of …, etc.) and *emojis*. More than one element per each category (or none) can be added to each artwork in the story. The user can also upload a single image from their device as a response to an artwork.

Stories can be saved in the user's personal space associated to their account and later deleted (but not edited). In order to identify them, and also as part of the creative process, users are prompted to give a title to each story as a precondition to saving it.

After creating each story, the user receives a set of recommendations of artworks based on the emotions assigned by the DEGARI reasoning service (see D6.3) from WP6 to the artworks in the story (in turn, these associations are inferred by DEGARI from the text content which accompanies the artwork: artwork description from the collection record, user comments, etc.). Recommendations are of two types: they concern artworks associated with the same or similar emotions, and artworks associated with emotions of opposite polarity. If accepted, a recommended artwork will be attached to the story.

Once shared, stories are associated with the artworks in the museum collections, and can be browsed by the other users, who can express their appreciation to them through likes, in the style of social media.

With this context in mind, in the following sections we will elaborate on each of the application flows in more detail, including screenshots to illustrate their use in a more visual way.

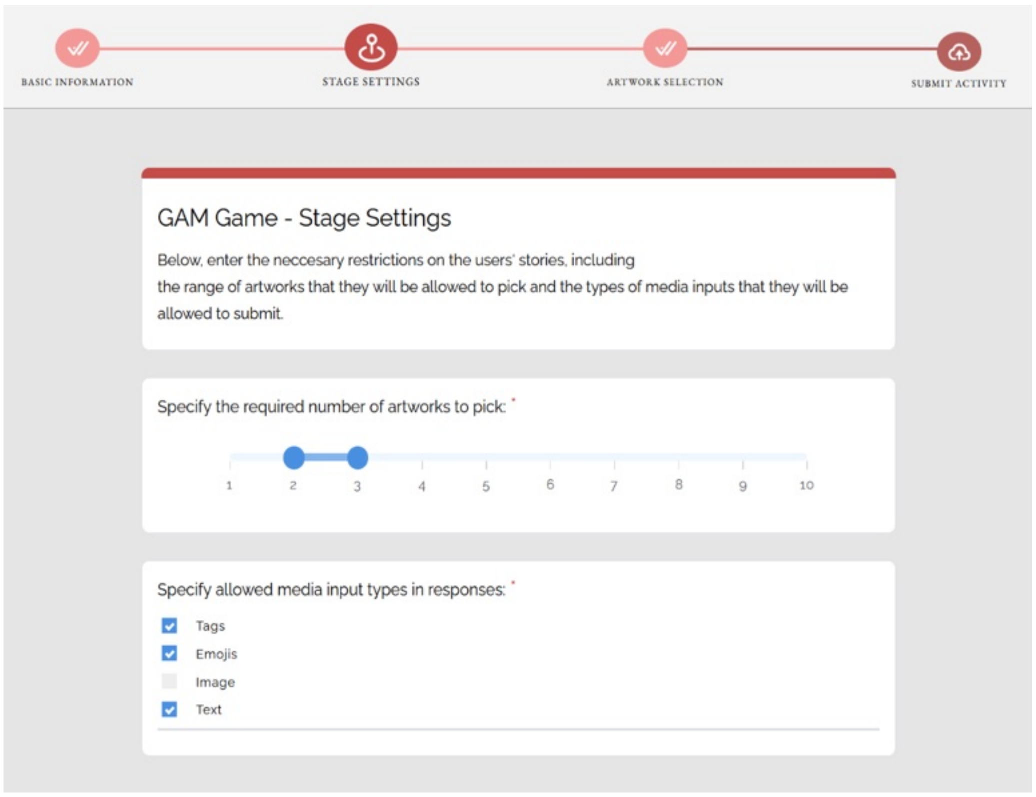
3.4.2. Activity Creation Interface - Main Flow

As in the context of the Find Artwork template, this activity includes a previous step in which the museum curator must go through a configuration process for their particular instance of the GAM Game, including the works they wish to incorporate into the activity and defining the different hyperparameters to configure the end user's experience. As in the previous case, it is necessary to go through 3 fundamental steps, with the first and the last one (Basic Information and Artwork Selection) shared with general activities and activities with artwork selections, respectively.

The final goal of this flow is the creation of a definition object for an activity instance that conforms to the GAM Game template format, derived from Activity Instance and therefore sharing all the minimum definition fields of a general activity (title, description, author, start and end dates, thumbnail, etc). The format of a valid definition object is given by the following interface:

```
export interface GamGameActivityDefinition extends ActivityInstance {
  activityType: 'GAM Game',
  minArtworks: number; // min number of artworks per story
  maxArtworks: number; // max number of artworks per story
  allowedResponseTypes: AllowedResponseType[]; // "Tags" | "Emojis" | "Image"
| "Text"
  storyDefinitionsDatasetUuid: string;
  artworksDatasetUuid: string;
  artworks: string[];
}
```

The fields regarding the min/max artworks allowed per story and the types of responses that are acceptable by story creators when specifying what interactions to add to their stories are filled in through the "stage settings" step interface, which can be found in Figure 25.



GAM Game - Stage Settings

Below, enter the neccesary restrictions on the users' stories, including the range of artworks that they will be allowed to pick and the types of media inputs that they will be allowed to submit.

Specify the required number of artworks to pick: *

Specify allowed media input types in responses: *

- [x] Tags
- [x] Emojis
- [ ] Image
- [x] Text

Figure 25

Once it is detected that all mandatory fields have been filled in by the user, the interface proceeds to enable a "Submit Activity" button, which calls the corresponding request from the GAM Game management service to store the resulting definition in the persistence layer. The filling of the common fields is handled in the exact same way as in the case of the Find Artwork activity creation flow via shared interfaces.

3.4.3. Activity Consumption Interface

The purpose of the explore GAM Game interface is to provide a workflow for visitors to engage with an instance of a GAM Game activity, thus allowing them to create their own stories to be used later on in the citizen curation phase, as well as to browse and view the stories created by other users taking part in the activity. In more concrete terms, the ultimate goal of this interface is to facilitate both the creation and the visualization of GAM Game Story objects that adhere to the following format:

```
export interface GamGameStoryDefinitionData {
  /** Id of this story within the database (needed for navigation and fetching
by id) */
  _id: string;
  /** Link to image that should be displayed when browsing stories */
  imageSrc?: string;
  /** Title of the story */
  title: string;
  /** Id of the activity that this story is assigned to */
  activityId: string;
  /** List of parts containing selected artworks and multimedia data */
  parts: GamGameStoryPart[];
}

/** Definition of a GAM Game story part, including selected artwork and user-
generated multimedia data */
export interface GamGameStoryPart {
  /** Id of the selected artwork within the database */
  artworkId: string;
  /** Multimedia data associated with a given story (texts, emojis, tags) */
  multimediaData: GamGameStoryPartMutimediaData;
}

/** Multimedia data associated with a given story (texts, emojis, tags) */
export interface GamGameStoryPartMutimediaData {
  /** What template is being used for the associated text. This is the part
that the user cannot modify directly,
   * instead needing to select a fixed text as a start for their answers */
  textTemplate: string;
  /** Text written by the user about the artwork */
  text: string;
  /** List of emojis that the user has placed on top of the artwork (+
positions) */
  emojis: StoryPartEmoji[];
  /** List of tags that the user has placed on top of the artwork (+
positions) */
  tags: StoryPartTag[];
}

/** Basic interface for any element that can be placed on top on an artwork */
export interface StoryPartOverlayElement {
  locationX: number; // in [0, 1]
  locationY: number; // in [0, 1]
}

export interface StoryPartTag extends StoryPartOverlayElement {
  tag: string;
}

export interface StoryPartEmoji extends StoryPartOverlayElement {
  emoji: Emoji;
}

export const availableEmoji = [
  '🤩',
  '🥰',
  '😱',
  '😴',
  '🤢',
  '😥',
  '😌',
  '🧐'
] as const;
```

```
export type Emoji = typeof availableEmoji[number];
```

Regarding this definition, it is pertinent to mention that the information related to tags and emojis placed by users over the artwork images is stored jointly with the coordinates (as width and height ratios) in which such elements were placed. Additionally, the admissible emojis are limited to a fixed selection, so as to facilitate both the operation of the emotional reasoning services and the emoji choices by the users.

When loading an activity page associated to a specific GAM Game definition id, the user can find a series of sub-pages and options to interact with both the activity and the artworks included in it (the user can switch between them with the help of the drop-down menu from the top left corner of the interface, Figure 26):

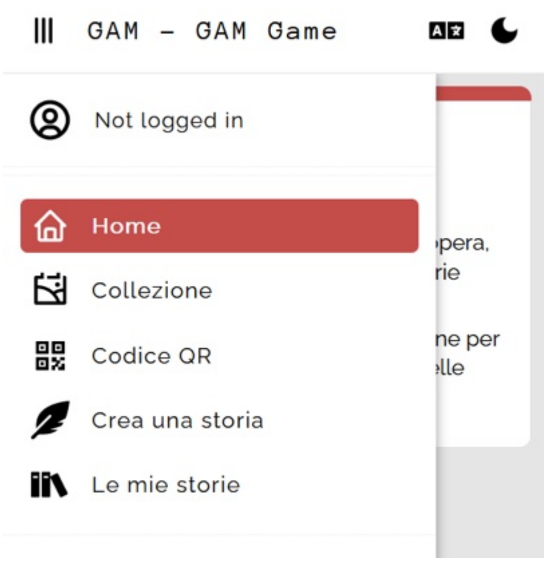
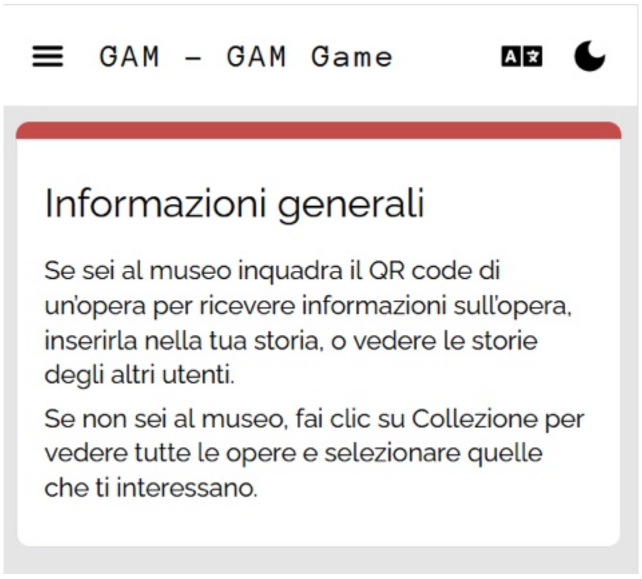


Figure 26

- **General Information.** Main page of the activity, which provides a brief description explaining the process to follow to interact with the different options.


GAM – GAM Game

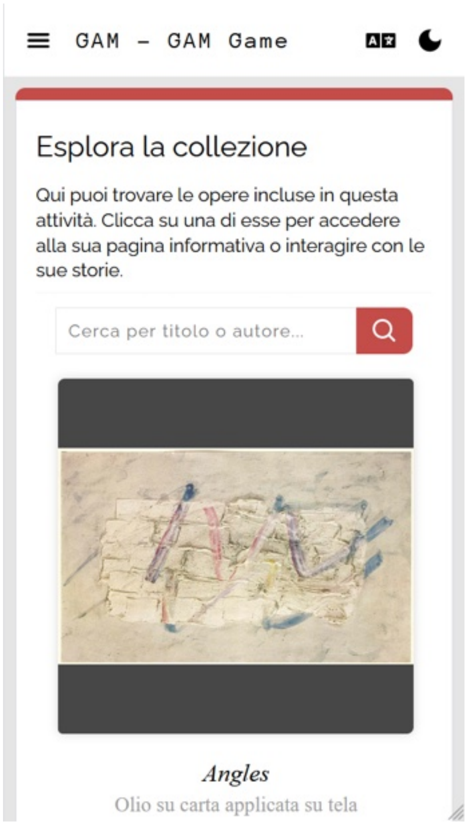
Informazioni generali

Se sei al museo inquadra il QR code di un'opera per ricevere informazioni sull'opera, inserirla nella tua storia, o vedere le storie degli altri utenti.

Se non sei al museo, fai clic su Collezione per vedere tutte le opere e selezionare quelle che ti interessano.

Figure 27

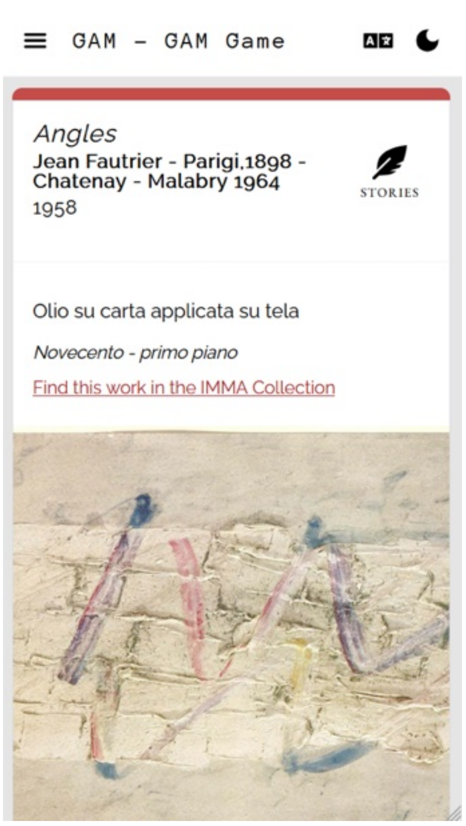
- **Explore the collection.** This page is intended to enable the exploration of the different artworks included in the activity and view their basic information (title, author, dating, general description, etc.), as well as a list of all the stories posted by the users of the activity that contain the corresponding artwork. All browsing flows include a search bar to facilitate the process of finding a work or story by title or author. The "collection browsing" interface can be found in Figure 28. “Explore the Collection” Interface., while some examples of interfaces intended to display the details of a particular work and a listing of stories that include it can be seen in Figure 29. “View Artwork Detail” Interface. and Figure 30. “Browse Stories” interface., respectively.



Esplora la collezione

Qui puoi trovare le opere incluse in questa attività. Clicca su una di esse per accedere alla sua pagina informativa o interagire con le sue storie.

Figure 28. "Explore the Collection" Interface.



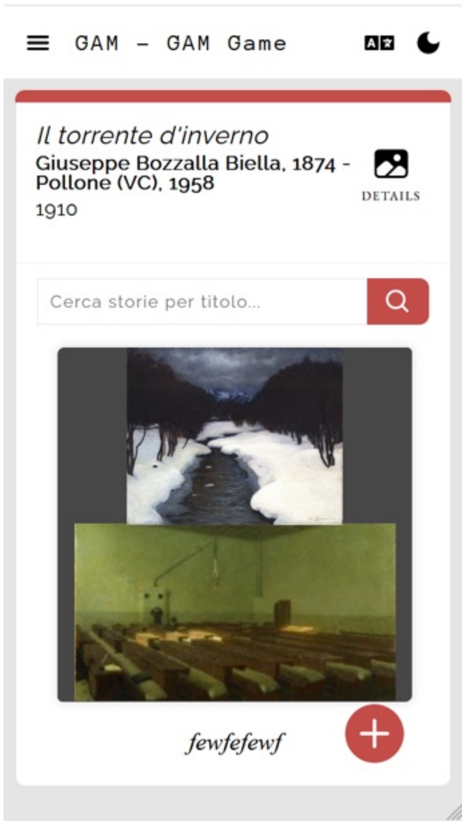
Angles
Jean Fautrier - Parigi,1898 - Chatenay - Malabry 1964
1958

Olio su carta applicata su tela

Novecento - primo piano

Find this work in the IMMA Collection

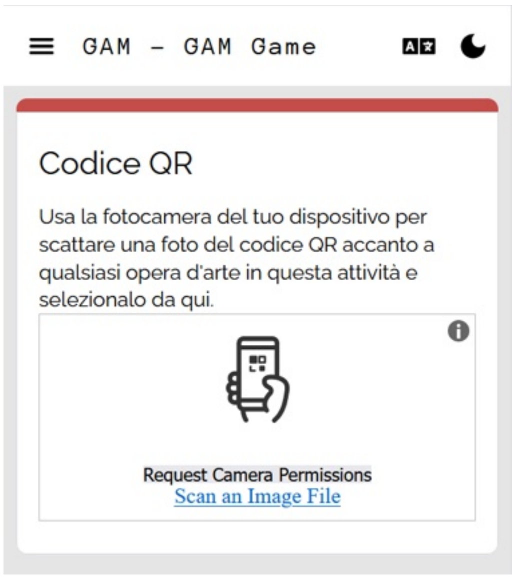
Figure 29. "View Artwork Detail" Interface.



Il torrente d'inverno
Giuseppe Bozzalla Biella, 1874 - Pollone (VC), 1958
1910

Figure 30. "Browse Stories" interface.

- **QR Code**. If the museum supports QR codes to identify the works included in the collection, it is possible to use them to navigate directly to the activity's entry dedicated to an artwork by clicking on the "Scan QR" option in the menu. Once here, the user can use the camera of their mobile device to point to the QR of the corresponding work and, if it is available in the activity, go on to explore its entry in the application. An example of this interface can be seen in Figure 31.



GAM – GAM Game

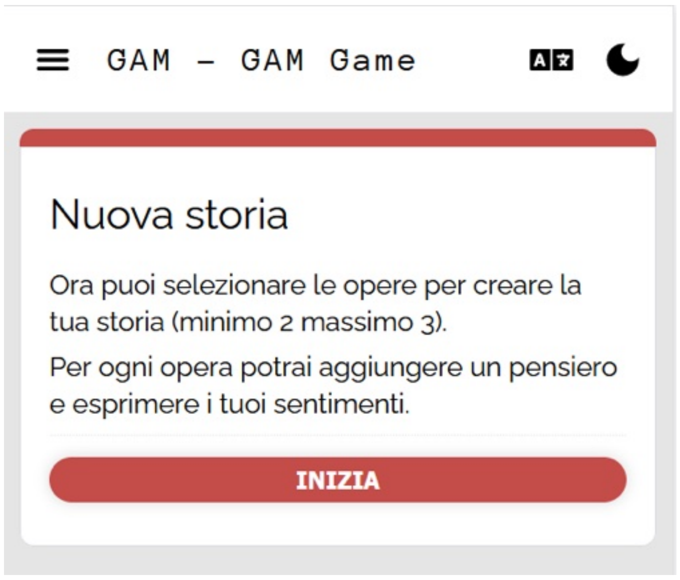
Codice QR

Usa la fotocamera del tuo dispositivo per scattare una foto del codice QR accanto a qualsiasi opera d'arte in questa attività e selezionalo da qui.

Request Camera Permissions
Scan an Image File

Figure 31

- **Create a story.** This functionality can be accessed directly from the menu, through the "Create a story" option, or from the story browsing interface associated to a given work, by means of the lower right button with the "+" icon. In the latter case, the story will be started with the selected work as the first piece. The story creation workflow is in turn structured in a series of sequential phases that will be described below.
  - **Introduction and general explanation of the process.** Gives the user a brief description of how to proceed during the creation of a story. Clicking on "Begin" starts the main creation flow.



GAM – GAM Game

Nuova storia

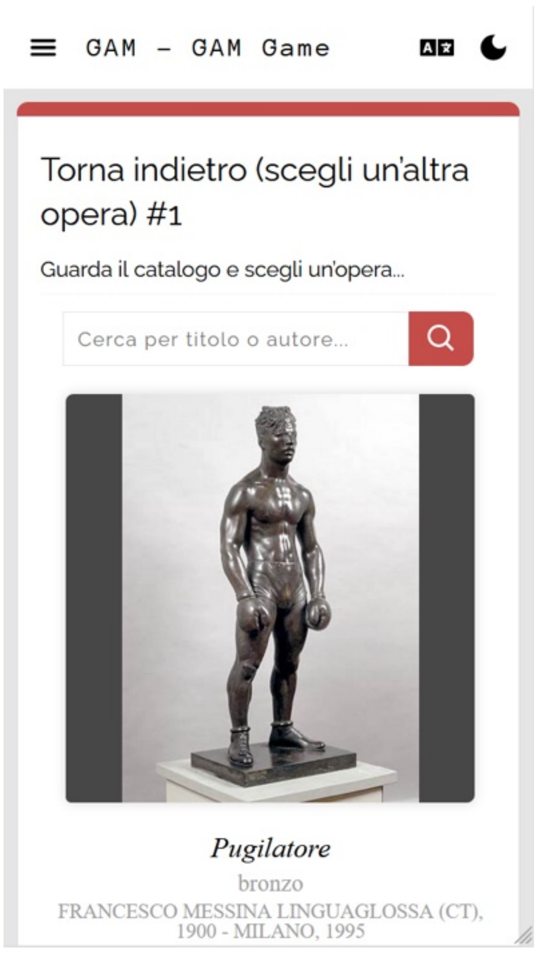
Ora puoi selezionare le opere per creare la tua storia (minimo 2 massimo 3).

Per ogni opera potrai aggiungere un pensiero e esprimere i tuoi sentimenti.

INIZIA

Figure 32

- **Select Artwork.** The user is presented with a list of works included in the activity, in a random order, from which they must choose the next artwork they wish to include in their story. Clicking on one of the works leads to the next phase, after receiving a confirmation from the user that this work is indeed the one they wish to include.



GAM – GAM Game

Torna indietro (scegli un'altra opera) #1

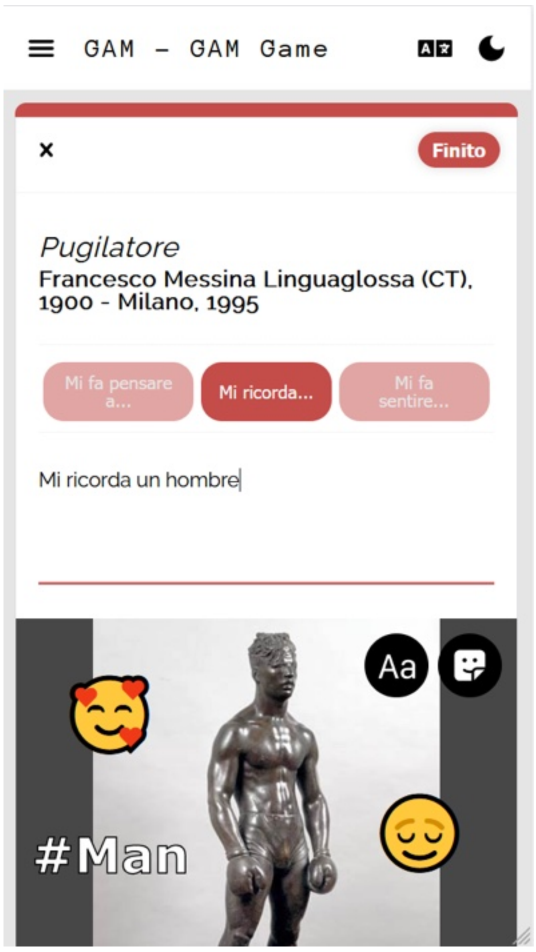
Guarda il catalogo e scegli un'opera...

Cerca per titolo o autore...

Pugilatore
bronzo
FRANCESCO MESSINA LINGUAGLOSSA (CT), 1900 - MILANO, 1995

Figure 33

- **Introduction of multimedia elements and user interactions with the artwork**. At this stage the user will be able to add tags and/or emojis over the image of the work to express their emotions and thoughts about it, as well as answers to "questions" or standard "templates", which take the form of pre-started sentences to guide the user in what they are expected to write as textual interaction ("It makes me feel...", "It reminds me of...", "It makes me think..."). These templates can be chosen by the user from the three available. The elements that are placed on the image can be moved freely by the user by dragging them with the mouse (if used from a computer browser) or with the finger (if using a touch device). An example of this interface can be found in Figure 34. Once some kind of response to the selected template has been included, a "finish" button is enabled to continue with the story creation process.

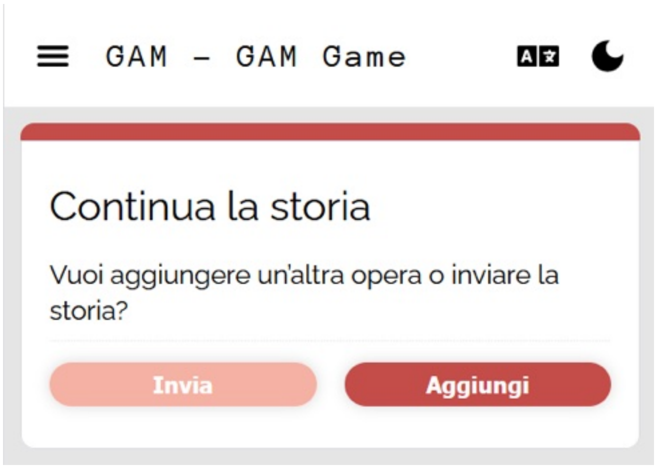


Pugilatore
Francesco Messina Linguaglossa (CT), 1900 - Milano, 1995

Mi ricorda un hombre

Figure 34

- **Prompt for the user on whether to add more works to the story or submit it as is.** After finishing each of the interactions with the works, and assuming that the maximum number of works in the activity specification has not been reached, the user will now see an interface asking if they wish to continue adding entries to the story or send it as it is. If the story had reached the maximum number of elements allowed, instead of this screen the user would go directly to the final submission and title selection screen, whereas if the story did not yet have the minimum number of elements required, this screen would not appear, instead showing a step to choose the next work.

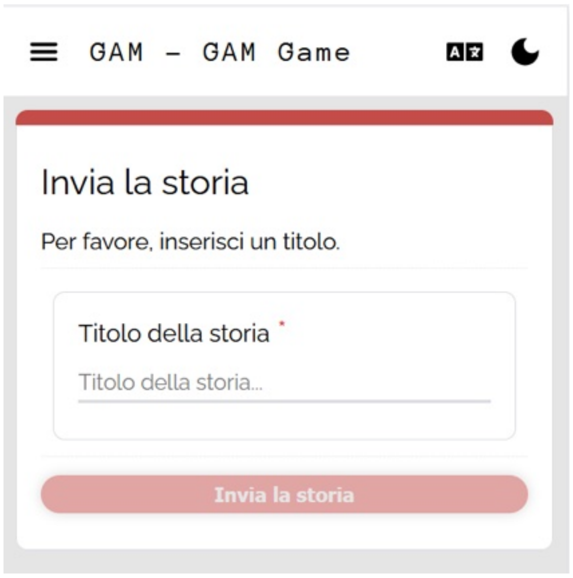

GAM – GAM Game

Continua la storia

Vuoi aggiungere un'altra opera o inviare la storia?

Figure 35

- **Introduction of the story title**. Before submitting their story, the user must type a meaningful title, which will be the one displayed to other users in all story browsing screens. This interface can be seen in Figure 36.


GAM – GAM Game

Invia la storia

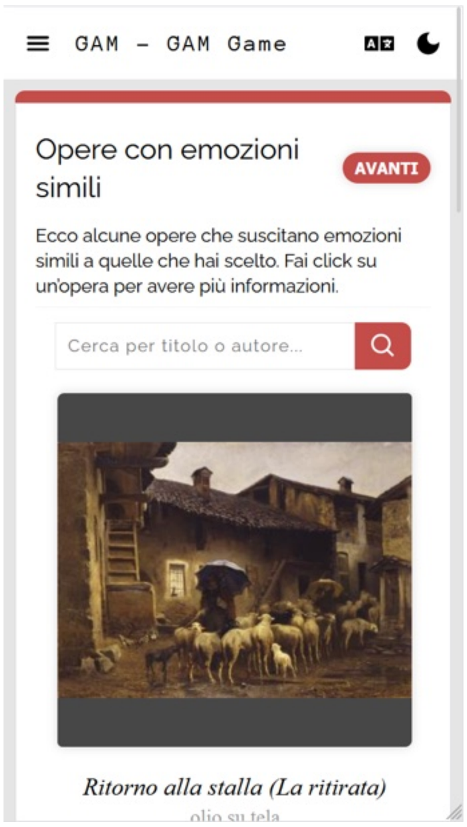
Per favore, inserisci un titolo.

Titolo della storia *

Titolo della storia...

Figure 36

- **Recommendation step for works that trigger emotions similar to those of artworks chosen by the user**. After completing the creation of a new story, the user will receive a series of recommendations for works similar to their own in terms of emotions triggered. An example of this interface can be seen in Figure 37.



Opere con emozioni simili

Ecco alcune opere che suscitano emozioni simili a quelle che hai scelto. Fai click su un'opera per avere più informazioni.

Ritorno alla stalla (La ritirata)

olio su tela

Figure 37

- **Recommendation step for works that trigger emotions opposite to those of artworks chosen by the user.** Similarly, the WP6 emotional reasoner is used to recommend works that trigger opposite emotions to those of the selected artworks.

  After this process, the work is sent to the persistence layer, and the user is redirected to the start screen of the activity.

- **My stories.** If the user is registered within inSPICE and has created a story with their active profile, they will be able to view a list of their own stories, with the same appearance as when browsing other users' stories about a selected work. From this interface it is also possible to delete stories already created by the user.
- **View a story saved within the application.** This flow can be accessed either from the "my stories" option, if the user wants to visualize their own story, or from the list of stories in the context of a specific artwork page, by clicking on the story the user wants to visualize. In this flow, the user is simply shown each of the works chosen by the creator of the story along with their interactions (emojis, tags or text in templates). The user can advance between the works of the story by clicking on the "Next" button. Some examples of this process are shown below in Figure 38, Figure 39, Figure 40 and Figure 41.

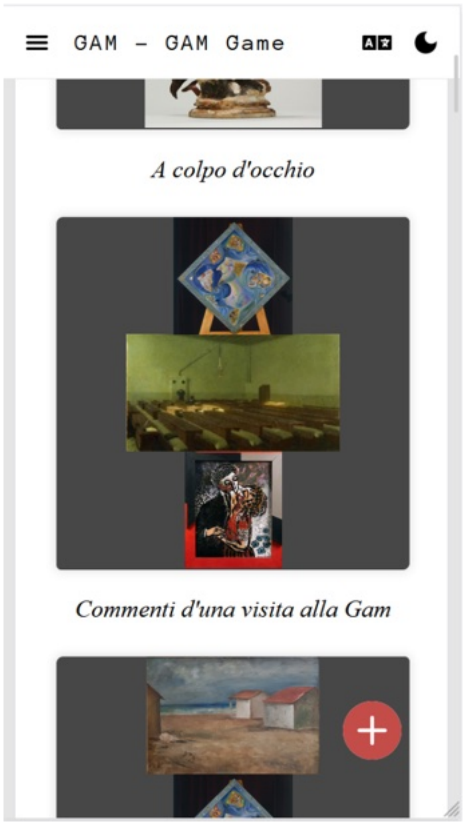
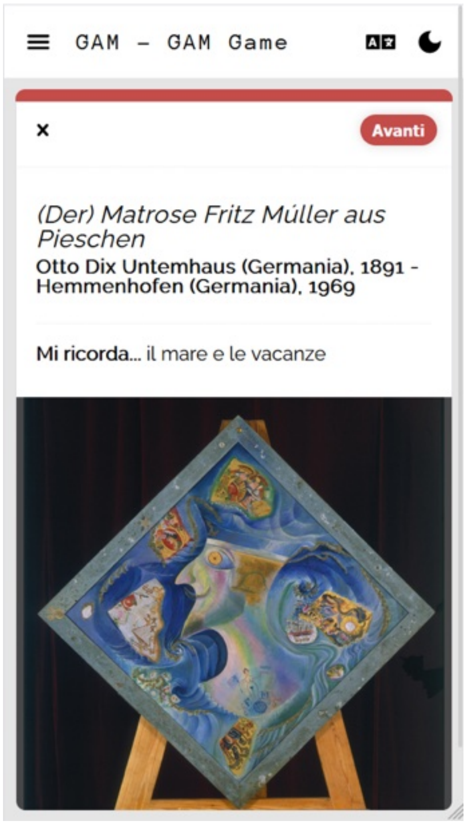


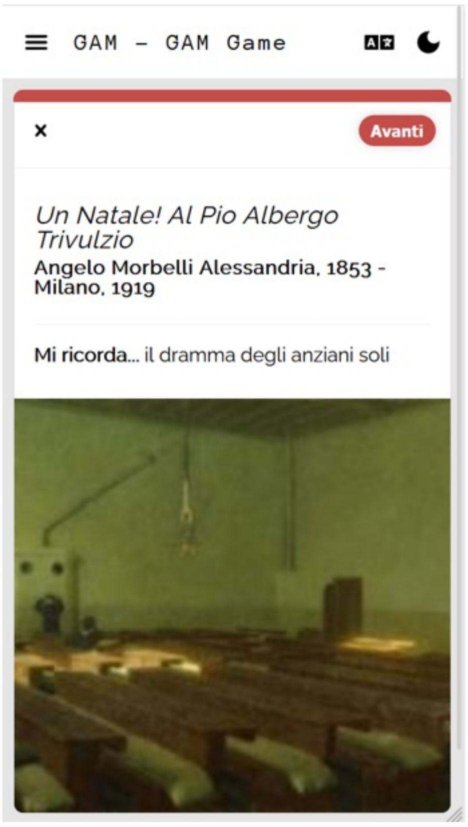
Figure 38



(Der) Matrose Fritz Müller aus Pieschen
Otto Dix Untemhaus (Germania), 1891 - Hemmenhofen (Germania), 1969

Mi ricorda... il mare e le vacanze

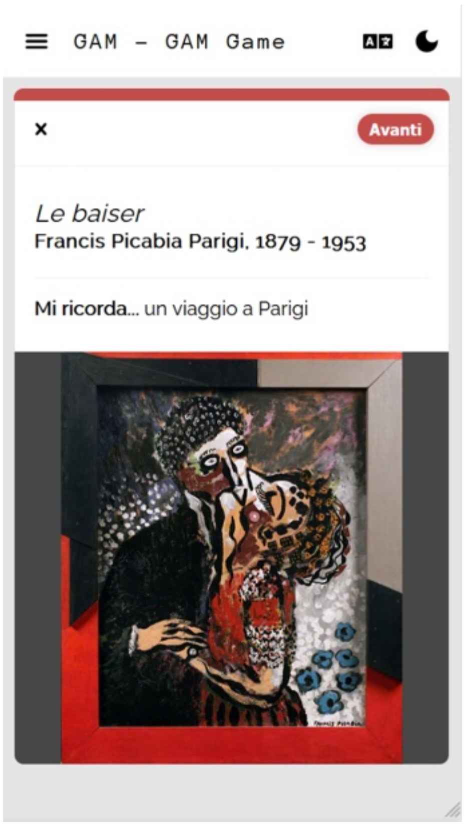
Figure 39



Un Natale! Al Pio Albergo Trivulzio
Angelo Morbelli Alessandria, 1853 - Milano, 1919

Mi ricorda... il dramma degli anziani soli

Figure 40



Le baiser
Francis Picabia Parigi, 1879 - 1953

Mi ricorda... un viaggio a Parigi

Figure 41

3.5. (MNCN/ Hecht) Multi-stage Form Creation

The activity represented by this template, which we will refer to as **Multistage Form** from now on, revolves around the notion of curating use-case specific forms that can then be completed by museum users/ visitors in different contexts. Through this type of activity, museum curators can design interactive forms with fields of various types to request users' opinions and responses to different prompts. This web application model is used extensively in the Hecht Museum use cases.

However, the same form creation workflow can be integrated into other seemingly unrelated use cases, such as the one at the National Museum of Natural Sciences in Madrid. In this case, the museum makes use of interactive treasure hunts (tablet applications created in Unity 3D) through which the visitor must explore the museum in search of artifacts based on in-game clues, complete mini-games, and answer questions about the different topics covered in the treasure hunt. The structure of such an activity can actually be constructed in exactly the same way in which a form is generated in this template, specifying phase by phase what type of prompt, content or mini-game is to be presented to the user at each point in the scavenger hunt, with the configuration of the phase, assuming it is a configurable stage. The definitions generated from this process can then be accessed through requests from the Unity application to the inSPICE server, and used to build a treasure hunt that adheres to those specifications. This interaction flow is shown in Figure 42. It is to be mentioned that these flows and interactions are still in the process of development at the time of writing.

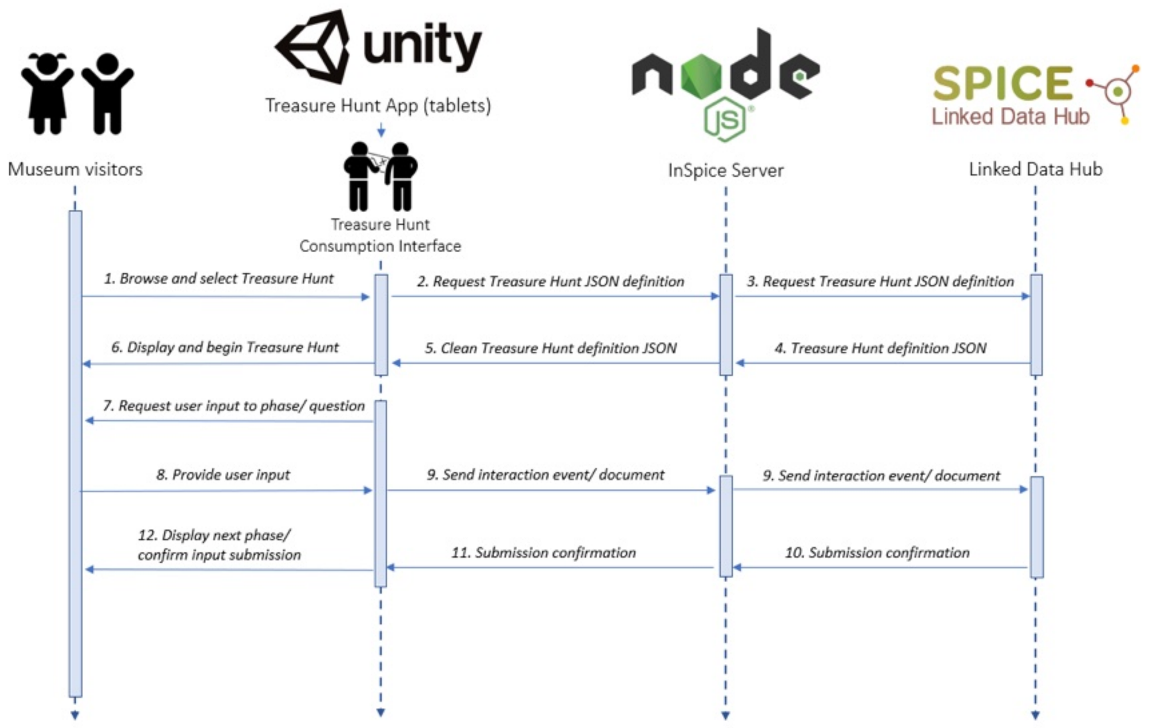

unity
Treasure Hunt App (tablets)
Museum visitors
Treasure Hunt Consumption Interface
node
InSpice Server
SPICE Linked Data Hub
Linked Data Hub
1. Browse and select Treasure Hunt
2. Request Treasure Hunt JSON definition
3. Request Treasure Hunt JSON definition
4. Treasure Hunt definition JSON
5. Clean Treasure Hunt definition JSON
6. Display and begin Treasure Hunt
7. Request user input to phase/ question
8. Provide user input
9. Send interaction event/ document
9. Send interaction event/ document
10. Submission confirmation
11. Submission confirmation
12. Display next phase/ confirm input submission

Figure 42

The creation flow associated with this template is dedicated to the creation of a definition object that conforms to the following schema derived from the basic activity interface:

```
export interface MultistageFormActivityDefinition extends ActivityInstance {
  activityType: 'Multistage Form',
  stages: MultistageFormStage<MultistageFormSupportedPayload>[];
  formResponsesDatasetUuid: string;
}

export type MultistageFormSupportedPayload =
  | { type: 'short-text', payload: ShortTextFieldDefinition }
  | { type: 'long-text', payload: LongTextFieldDefinition }
  | { type: 'multiple-choice', payload: MultipleChoiceFieldDefinition }
  | { type: 'likert-scale', payload: LikertScaleFieldDefinition }
  | { type: 'checkbox', payload: CheckboxGroupFieldDefinition }
  | { type: 'range', payload: RangeFieldDefinition }
  | { type: 'calendar', payload: never }
  | { type: 'tags', payload: TagsFieldDefinition }
  ;
```

Here, stages is a list of MutistageFormStages, objects that include information about each of the stages of a form, including title, description and, most importantly, a list of definitions of items to include within the stage. The items to be included within any given stage are given by objects that conform to one of the types allowed within MultistageFormSupportedPayload (these definitions may be modified to meet the particular needs of a specific use case, the ones included here are only intended to serve as examples of common items within a form). In this way the forms are built as sequences of phases, in turn composed by sets of items (questions, multimedia content, etc) that will be shown to the user in each phase.

The interface for the creation of this object is shown in Figure 43. Creation interface for a web-based multistage form. (in use cases involving web forms).

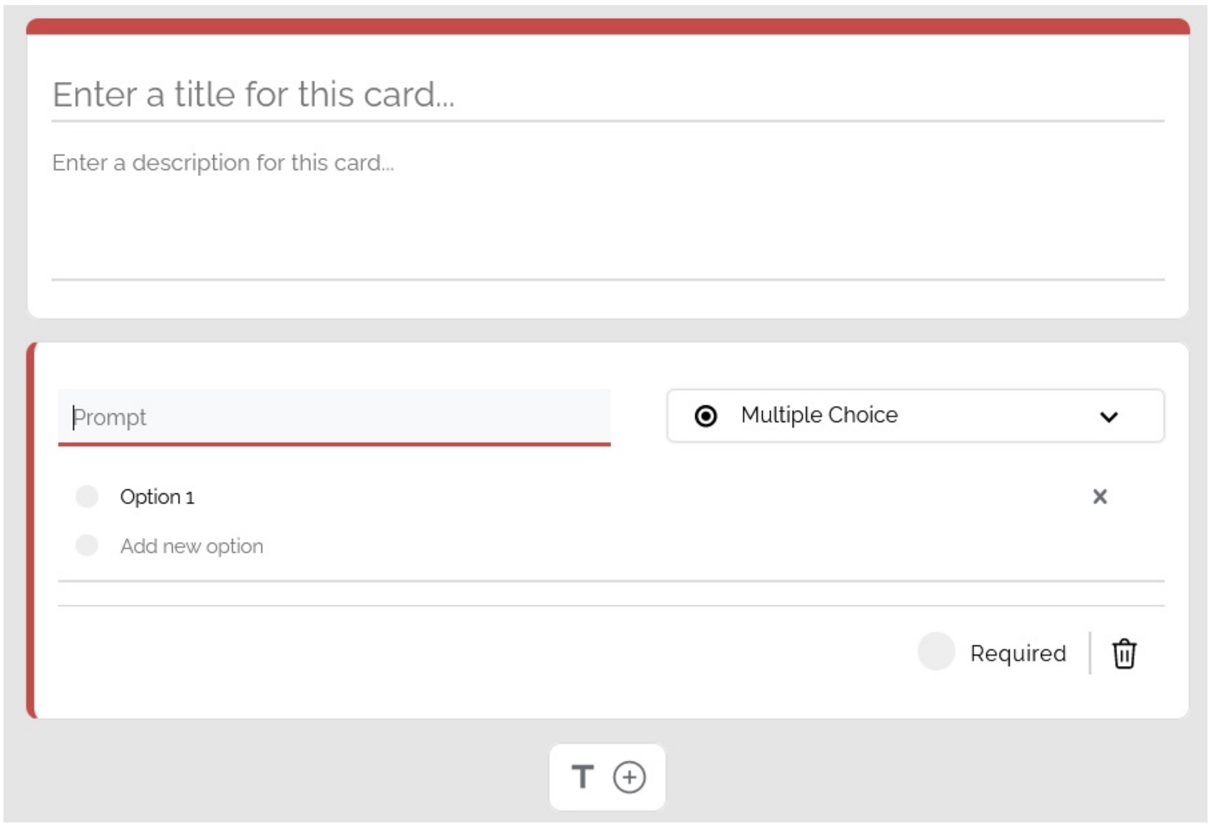
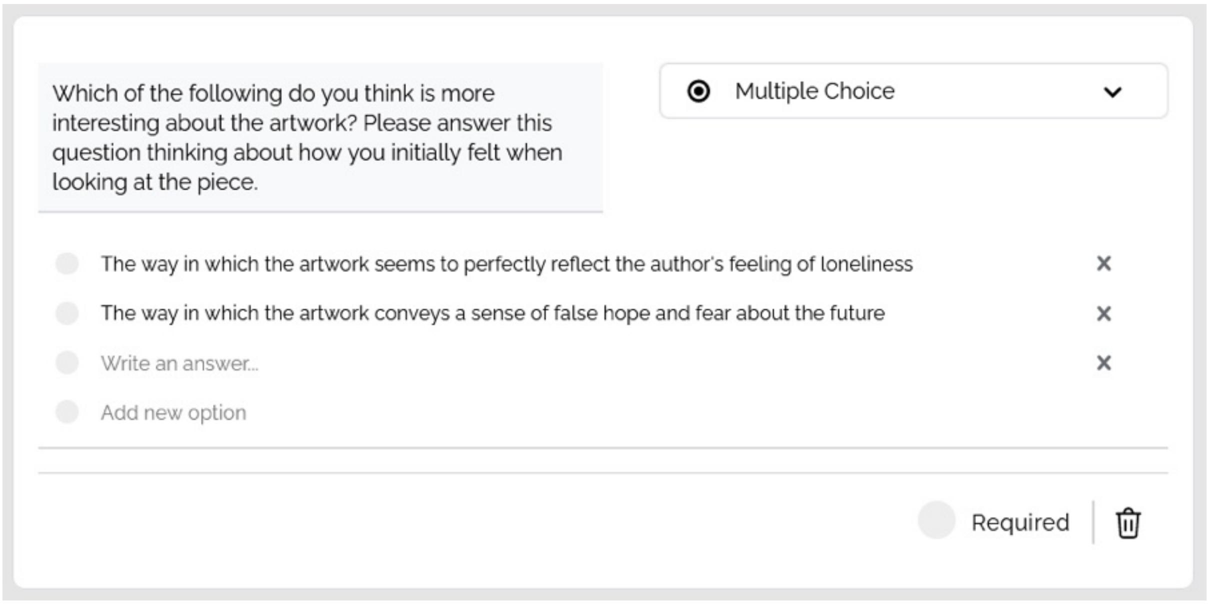

Enter a title for this card...
Enter a description for this card...
Prompt
Multiple Choice
Option 1
Add new option
Required
T

Figure 43. Creation interface for a web-based multistage form.

Essentially, for each stage, the application presents the user with a description and title card of the phase, which will be displayed at the beginning of the associated consumption interface, and an action bar through which it is possible to add new items to the phase, both textual information and questions or prompts for the user ("T" and "+" icons in the figure).

When adding an interactive question from this action bar, the user can choose between the different types of items allowed in the activity by means of the drop-down in the upper right corner of the card, and edit the necessary fields for the selected item type. An example of this can be seen in figures Figure 44. Editing tool for a Multiple Choice prompt. and Figure 45. Detail of the prompt type dropdown. for the case of a Multiple Choice type item.

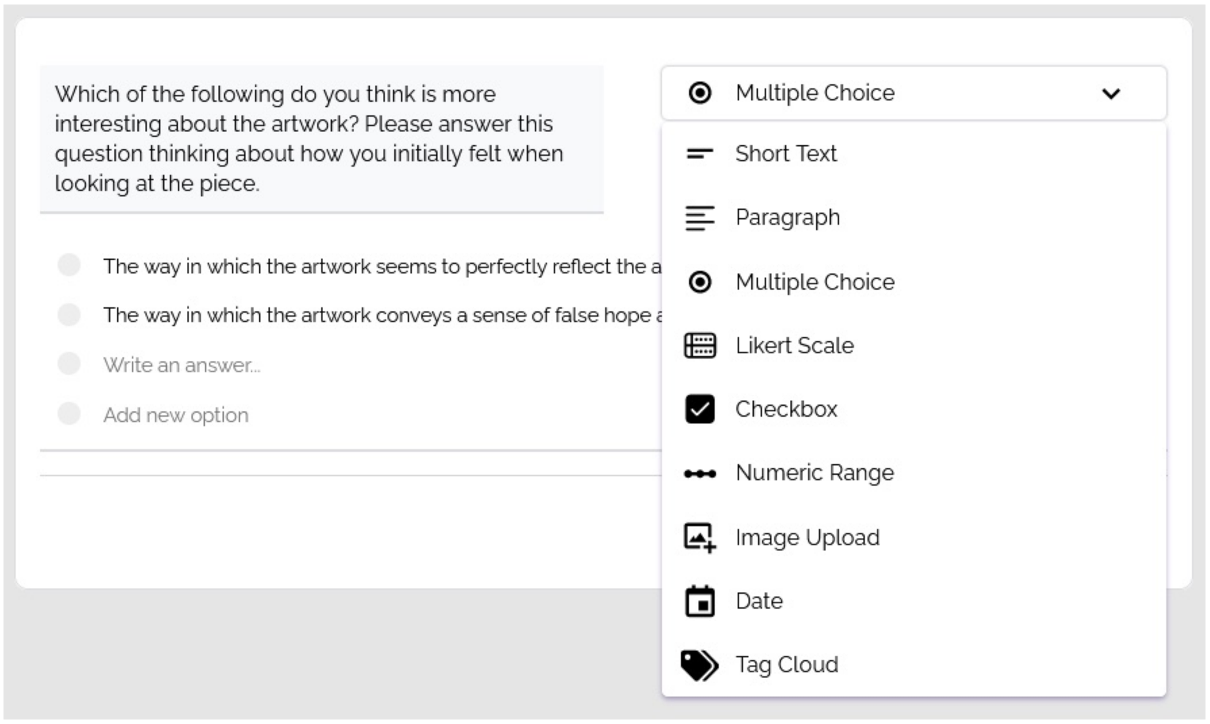

Which of the following do you think is more interesting about the artwork? Please answer this question thinking about how you initially felt when looking at the piece.

Multiple Choice

- The way in which the artwork seems to perfectly reflect the author's feeling of loneliness
- The way in which the artwork conveys a sense of false hope and fear about the future
- Write an answer...
- Add new option

Required

Figure 44. Editing tool for a Multiple Choice prompt.


Which of the following do you think is more interesting about the artwork? Please answer this question thinking about how you initially felt when looking at the piece.

- The way in which the artwork seems to perfectly reflect the a
- The way in which the artwork conveys a sense of false hope a
- Write an answer...
- Add new option

Multiple Choice

- Short Text
- Paragraph
- Multiple Choice
- Likert Scale
- Checkbox
- Numeric Range
- Image Upload
- Date
- Tag Cloud

Figure 45. Detail of the prompt type dropdown.

Additionally, all cards that include dynamic prompts can be marked as required or removed freely by means of the options placed in the lower right corner of the cards.

The consumption flow of the definitions generated through this tool is under active development at the time of writing this document.

3.6. (MNCN) Collection Browsers

The inSPICE framework additionally provides, albeit not in the form of a template, the necessary infrastructure to easily generate browsers of museum artwork or artifact collections, with support for searches by tags and keywords. At the time of writing, this functionality is enabled for the use case of the National Museum of Natural Sciences of Madrid (MNCN), and enables the browsing and exploration of a sub-collection of about 100 museum artifacts for which detailed explanations are available. The associated interface can be seen in Figure 46. Collection browser interface for a subset of artworks in MNCN..

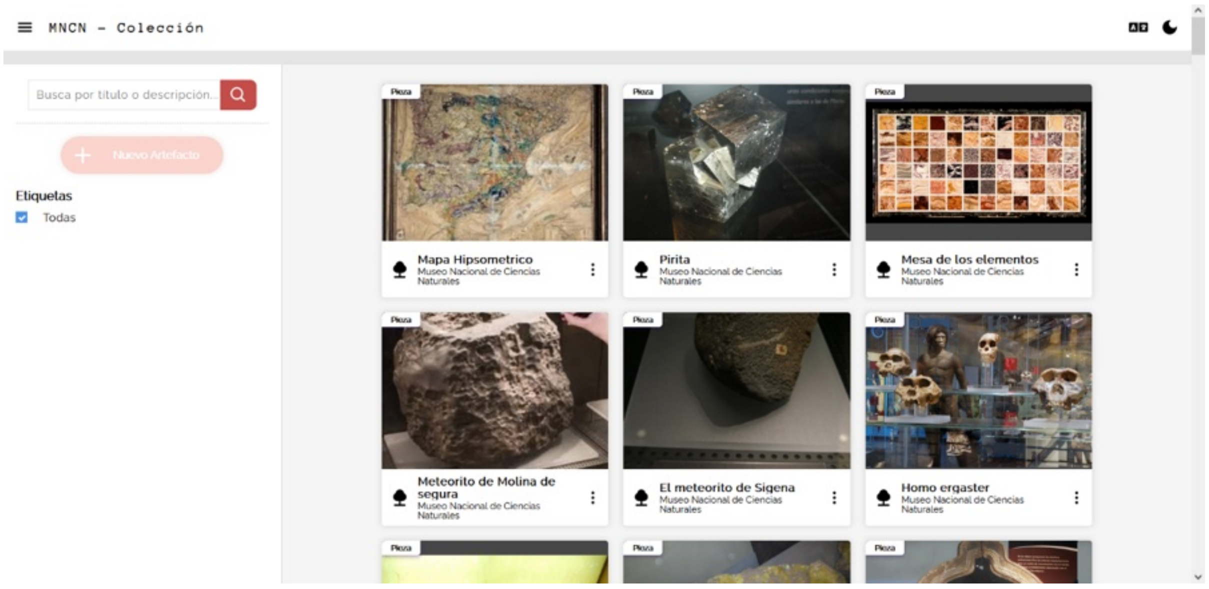

MNCN - Colección
Busca por título o descripción...
Nuevo Artefacto
Etiquetas
Todas
Pieza
Mapa Hipsometrico
Museo Nacional de Ciencias Naturales
Pirita
Museo Nacional de Ciencias Naturales
Mesa de los elementos
Museo Nacional de Ciencias Naturales
Meteorito de Molina de segura
Museo Nacional de Ciencias Naturales
El meteorito de Sigena
Museo Nacional de Ciencias Naturales
Homo ergaster
Museo Nacional de Ciencias Naturales

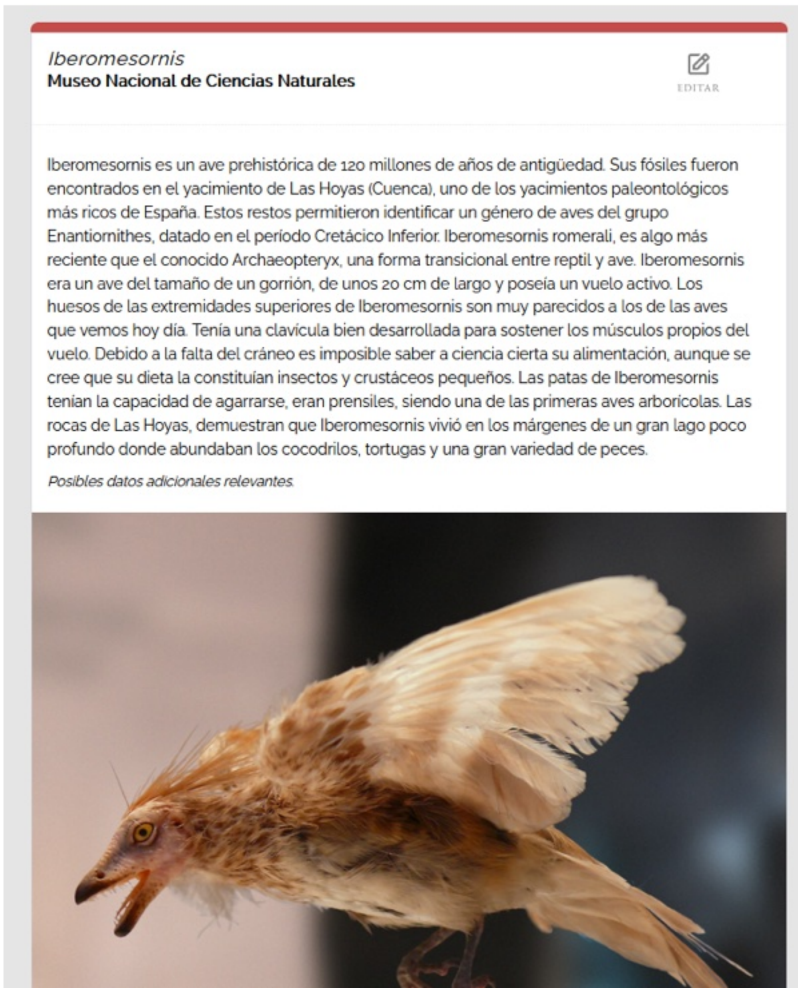
Figure 46. Collection browser interface for a subset of artworks in MNCN.

The artifacts displayed in the browser and managed by this flow conform to the following specification:

```
export interface BrowserArtifact {
  title: string;
  description: string;
  tags?: string[];
  imageSrc: string;
  _id: string;
}
```

While this definition is seemingly limiting, in practice expanding it to include additional optional fields is straightforward by simply extending the above definition and adding support for such fields in the artifact detail components.

By clicking on each of the artifacts, the application redirects the user to a piece detail page, which displays an enlarged version of the selected object’s image, as well as the full title, labels and a text dedicated to the description of the element. An example of these interfaces can be found in Figure 47. Artifact detail interface..


Iberomesornis
Museo Nacional de Ciencias Naturales

Iberomesornis es un ave prehistórica de 120 millones de años de antigüedad. Sus fósiles fueron encontrados en el yacimiento de Las Hoyas (Cuenca), uno de los yacimientos paleontológicos más ricos de España. Estos restos permitieron identificar un género de aves del grupo Enantiornithes, datado en el período Cretácico Inferior. Iberomesornis romerali, es algo más reciente que el conocido Archaeopteryx, una forma transicional entre reptil y ave. Iberomesornis era un ave del tamaño de un gorrión, de unos 20 cm de largo y poseía un vuelo activo. Los huesos de las extremidades superiores de Iberomesornis son muy parecidos a los de las aves que vemos hoy día. Tenía una clavícula bien desarrollada para sostener los músculos propios del vuelo. Debido a la falta del cráneo es imposible saber a ciencia cierta su alimentación, aunque se cree que su dieta la constituían insectos y crustáceos pequeños. Las patas de Iberomesornis tenían la capacidad de agarrarse, eran prensiles, siendo una de las primeras aves arborícolas. Las rocas de Las Hoyas, demuestran que Iberomesornis vivió en los márgenes de un gran lago poco profundo donde abundaban los cocodrilos, tortugas y una gran variedad de peces.

Posibles datos adicionales relevantes.

Figure 47. Artifact detail interface.

Additionally, and assuming that the user of the flow has the necessary permissions within the system, the above interfaces allow the user to manually edit the content of the artifacts included in the collection, delete them, or add new elements at will. A detail of the context menu with these options can be seen in figure Figure 48. Actions menu for an artifact in the collection browser..

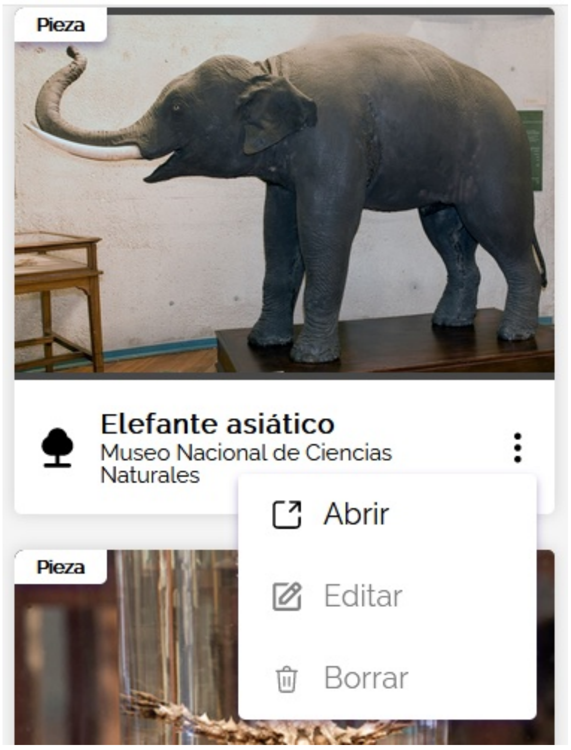


Figure 48. Actions menu for an artifact in the collection browser.

3.7. Activity Dashboard

In order to facilitate the exploration and creation of activity instances based on the templates described in the previous sections, the framework provides an activity dashboard as shown in Figure 49. Activity Dashboard sample..

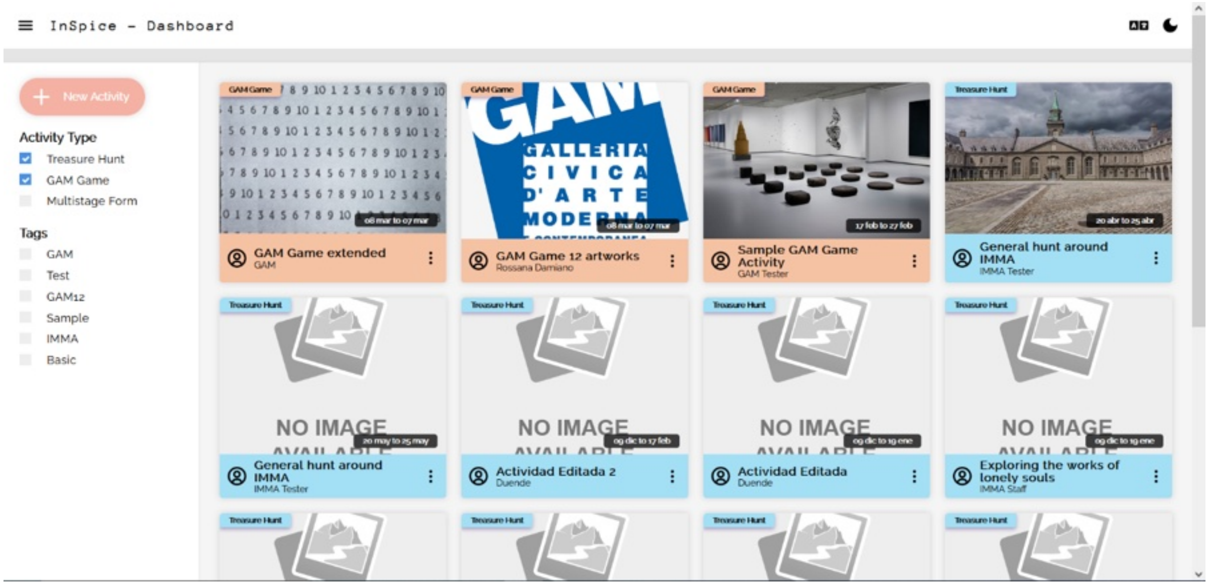


Figure 49. Activity Dashboard sample.

This interface offers support for searching and filtering activities by tags or template types that are used as a base and, as was the case with the collection browsers described in the previous section, by clicking on the three dots icon in the lower right corner of each thumbnail, the application displays a contextual menu (Figure 50. Actions panel for a sample activity within the Dashboard.) from which the activity instance can be opened (via a link that can be used to distribute it to the activity participants), edited (assuming the user is the creator of the activity), duplicated (which opens a creation flow associated with the template with all the fields filled in with the data of the selected activity by default), or deleted (again, if the user is the creator).

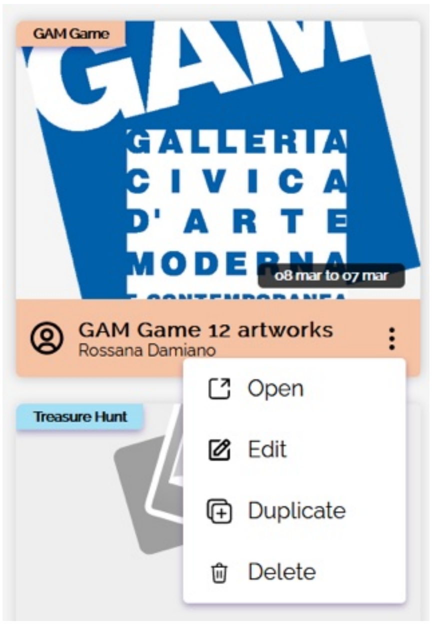


Figure 50. Actions panel for a sample activity within the Dashboard.

In order to add and manage an activity instance within this interface, it is necessary to manually add the sources (in this case, UUIDs of LDH collections) from which to extract activity definitions in the configuration of the web application. During the initial loading of this flow, the application will take this list of sources and make the relevant requests to the inSPICE server to retrieve the associated definitions, which must in all cases adhere to the basic specification of an activity described in Section 3.1.

If an activity definition object fetched from one of the specified sources adheres to the above interface, the application will perform an additional check to verify that the object is consistent with the specific activity instance definition associated with the value specified in the activityType field (for example, if activityType is 'GAM Game', the object will be validated against the GamGameActivityDefinition schema). All activities that pass the validation filter will be accessible from the dashboard.

Finally, by clicking on the "New Activity" button, the application will display a popup with the list of templates enabled for creating new activities within the current instance of inSPICE. Clicking on any of the templates will redirect the user to the relevant creation flow to generate a new activity definition based on that template. An example of this action can be seen in Figure 51, with two templates enabled for creation (Treasure Hunt and GAM Game).

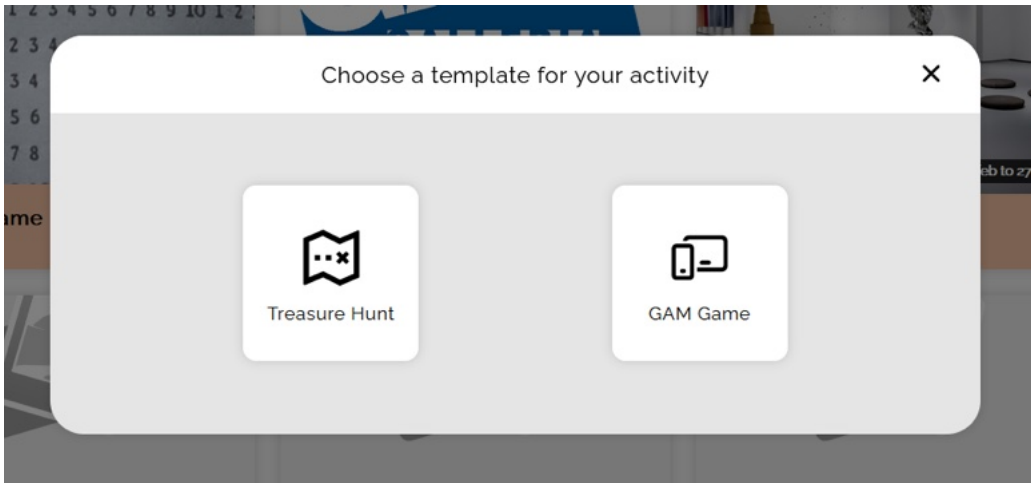


Figure 51

4. Components with general usage

4.1. Artwork Selection

4.1.1. Artwork Search Results

This component is responsible to show the result(s) of a search for an artwork in a grid. I the number of the results is a large number the grid will be break into pages. At any moment on page will be active and it is possible to navigate through these pages. Artworks can be selected from the search result and if need more information about each artwork is available by clicking on it.

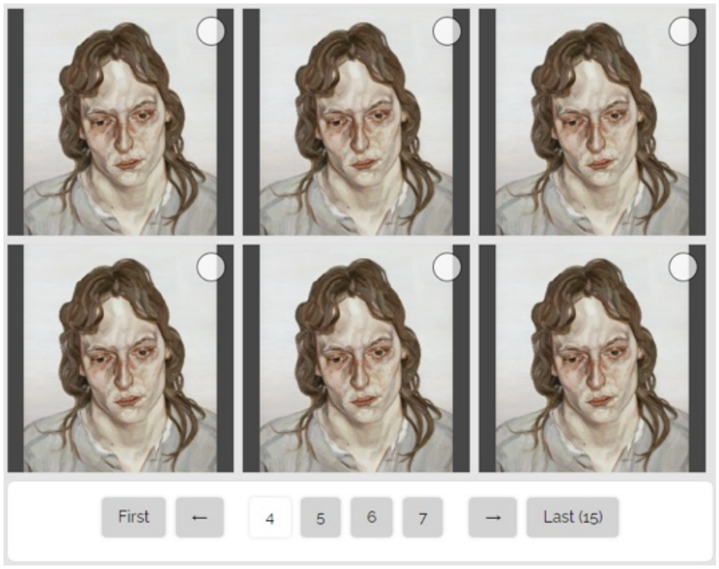


Figure 52. Artwork search result.

Props needed for component:

```
export interface ArtworkSearchResultsProps {
  /** List of artworks adhering to the ArtworkData interface, each of them
   * containing at least an artwork’s id, author, title, and image source.*/
  artworks: ArtworkData[];
  /** List of artwork ids that the user has already selected. */
  selectedArtworks: string[];
  /** Currently displayed page. */
  page?: number;
  /** Total number of indexed pages to be displayed. */
  pageTotal?: number;
  /** Callback to the parent of this panel indicating that an artwork has been
   * selected (Added to selection). */
  onArtworkSelected?: (artwork: ArtworkData) => void;
  /** Callback to the parent of this panel indicating that an artwork has been
   * deselected (Removed from selection). */
  onArtworkDeselected?: (artworkId: string) => void;
  /**
   * Callback to the parent of this panel indicating that an artwork has been
   * clicked from the panel.
   * Note that this is different from selection, as clicking on an artwork
   * just means that the user wishes
   * to explore it (e.g. to get more information about it).
   */
  onArtworkClicked?: (artworkId: string) => void;
  /** Callback to the parent of this panel indicating that the user wishes to
   * change the currently displayed page. */
  onPageChange?: (page: number) => void;
};
```

4.1.2. Artwork Selected Card

This is a simple component to showcase the details of a single selected Artwork object, containing the ArtworkData interface. This artwork can be flipped and there is a callback in case of deselection.

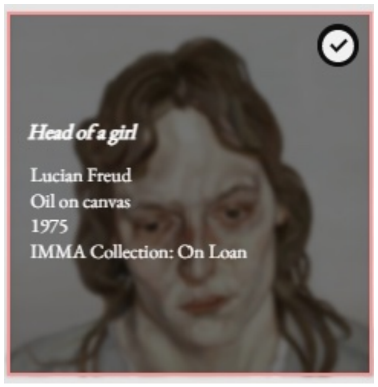


Figure 53. Selected artwork.

Props needed for component:

```
export interface ArtworkSelectedCardProps {
  /** Boolean indicating if the artwork card is flipped or not. */
  flipped: boolean;
  /** The artworkData object containing the card's information. */
  artworkData: ArtworkData;
  /** Callback to notify the parent when the card is deselected. */
  onArtworkDeselected: () => void;
};
```

4.1.3. Artwork Selection Card

This component is very similar to the ArtworkSelectedCard with one important difference. This component is not giving the details of the artwork, it is just for only for showing it to the user. The selected card later can be passed to the ArtworkSelectedCard for giving more of its details to the user.

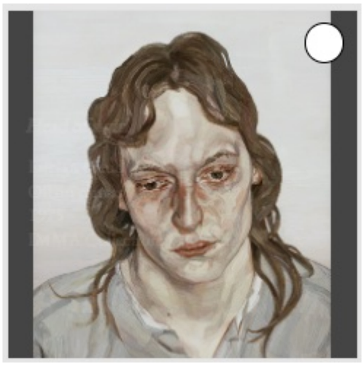


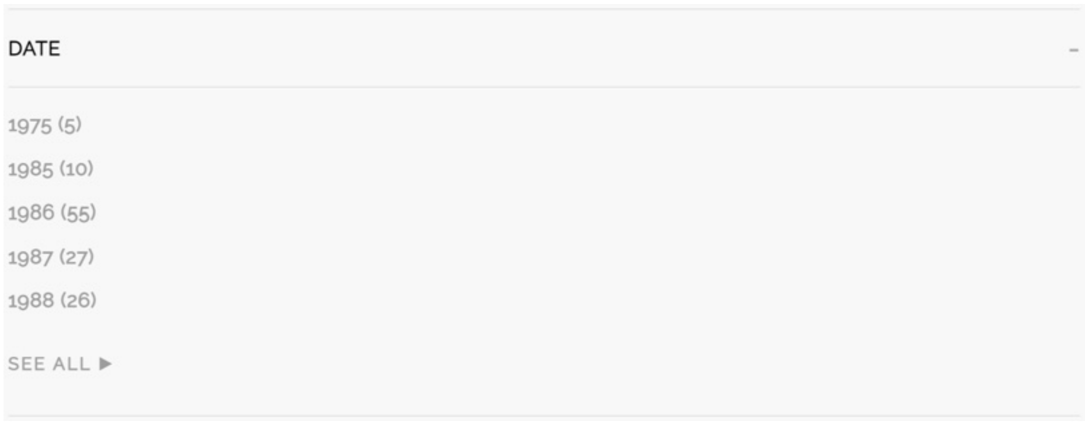
Figure 54. Unselected artwork.

Props needed for component:

```
export interface ArtworkSelectionCardProps {
  /** Artwork data to be used to render this component.
   *  Contains the Artwork’s information. */
  artworkData: ArtworkData;
  /** Whether the card should be displayed if is selected or not. */
  selected: boolean;
  /** Callback to the parent of this panel indicating that this artwork has
   *  been selected (Added to selection) */
  onCardSelected?: () => void;
  /** Callback to the parent of this panel indicating that this artwork has
   *  been deselected (Removed from selection) */
  onCardDeselected?: () => void;
  /**
   * Callback to the parent of this panel indicating that this card has been
   * clicked.
   * Note that this is different from selection, as clicking on an artwork
   * just means that the user wishes
   * to explore it (e.g. to get more information about it).
   */
  onCardClicked?: () => void;
};
```

4.1.4. Filter Field

This component is being used to show a set of selectable filters to the user on a specific field. These filters are shown in a list which can be limited to a desired number. If this is the case a “See All” option is provided which will open a popup on click to demonstrate all the available options with no limits.


DATE –

1975 (5)
1985 (10)
1986 (55)
1987 (27)
1988 (26)

SEE ALL ▶

Figure 55. A filter fields.

Props needed for component:

```
export interface FilterFieldProps {
  /** Display name of the filter category. */
  filterField: string;
  /** Array of filter options for the given category. */
  filterOptions: string[];
  /** Array of numbers that are used in order as ids for the filters. */
  filterCounts: number[];
  /** Whether to render a bottom border (this is useful when displaying a
   *  clicked sequence of filter categories). */
  bottomBorder?: boolean;
  /** Maximum number of options to be shown to the user before expanding the
   *  clicked list */
  maxOptionsShown?: number;
  /** Callback to the parent component specifying that a specific filter has
   *  clicked been selected within this category. */
  onFilterSelected: (filter: string) => void;
};
```

4.1.5. Filter Panel

A dictionary containing all different, unique filter fields that the user might access, as well as the different values that each of those filters considers. E.g. "Date" -> ["1997", "1998", ...]. Each of the entries provided here will be used to generate a filtering subpanel with up to maxOptionsShown entries each.

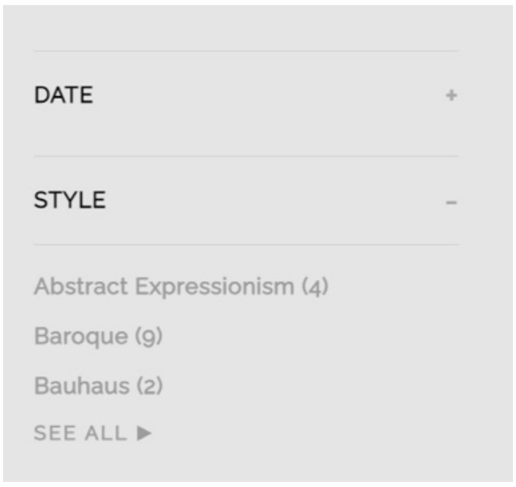


Figure 56

Props needed for component:

```
export interface FilterPanelProps {
  /**
   * Dictionary containing all different, unique filter fields that the user
   * might access, as well as the different values that each of those filters
   * considers.
   * E.g. "Date" -> ["1997", "1998", ...].
   * Each of the entries provided here will be used to generate a filtering
   * subpanel with up to maxOptionsShown
   * entries each.
   */
  uniqueFilterFields: Map<string, { value: string; count: number }[]>;
  /**
   * Maximum number of entries to display for each filter field. Additional
   * entries may be inspected by using the see all button at the bottom of
   * each field's subpanel.
   */
  maxOptionsShown: number;
  /**
   * Callback to notidy parent component of the selection of a filter within
   * the panel.
   * Receives two arguments as parameters, namely field and filter, to denote
   * the name of the field for which a filter was selected (e.g. 'Date') and
   * the value of the filter applied (e.g. '1997'), respectively.
   */
  onFilterApplied?: (field: string, filter: string) => void;
};
```

4.1.6. Filter Pop Up

A popup component that will be shown if the user clicks on a “See All” button on a filter list. The same set of filters will be available to user in a popup, however in full view with no limits in the number of shown filters. The probs for this component are very similar to the

“FilterField” except there is no maximum number of filters here. There is also a callback to open or close the popup.

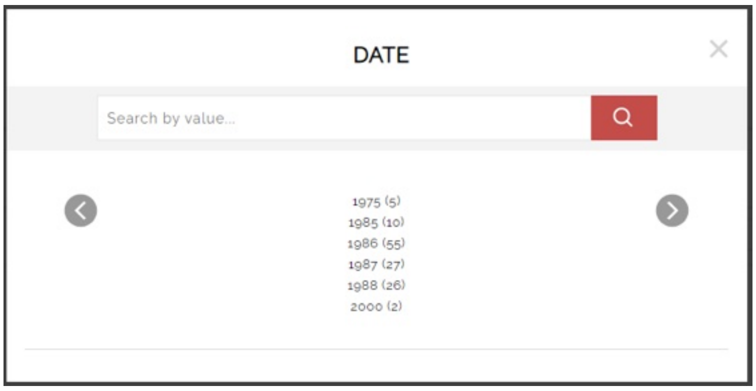


Figure 57. Filter popup.

Props needed for component:

```
export interface FilterPopupProps {
  /** Display name of the filter category. */
  filterField: string;
  /** Array of filter options for the given category. */
  filterOptions: string[];
  /** Array of numbers that are used in order as ids for the filters. */
  filterCounts: number[];
  /** Callback to the parent component specifying that a specific filter has
   *  been selected within this category. */
  onFilterSelected?: (filter: string) => void;
  /** Callback to the parent component specifying whether we want to close
   *  this component. */
  setFilterPopupOpen?: (opened: boolean) => void;
};
```

4.1.7. Search and Select Many Artworks

Component to display a collection of artworks and allow for basic selection, filtering, search and navigation functionalities.

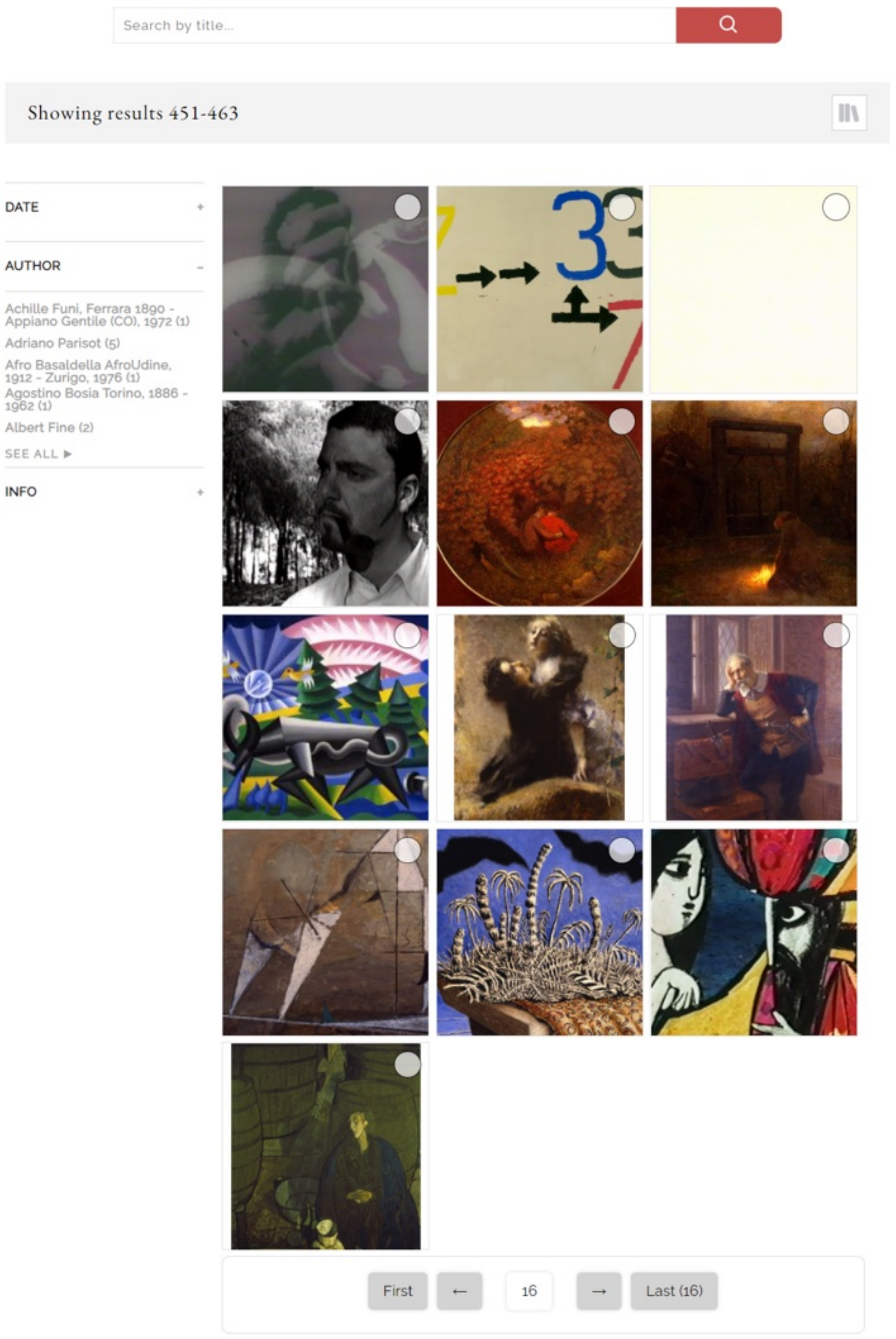

Search by title...
Showing results 451-463
DATE +
AUTHOR -
Achille Funi, Ferrara 1890 - Appiano Gentile (CO), 1972 (1)
Adriano Parisot (5)
Afro Basaldella AfroUdine, 1912 - Zurigo, 1976 (1)
Agostino Bosia Torino, 1886 - 1962 (1)
Albert Fine (2)
SEE ALL ▶
INFO +
First
←
16
→
Last (16)

Figure 58

Props needed for component:

```
export interface SearchAndSelectManyArtworksProps {
  /**
   * Artworks that should be displayed by this Component. This is not intended
   * to be a comprehensive list of all artworks returned from a big query,
   * but rather a slice of a query's results, or the result of a paginated
   * query.
```

```
 */
displayedArtworks: ArtworkData[];
/**
 * Currently displayed page and total number of pages considered, as well
 * as number of items per page. This allows the consumer of this component
 * to specify the state of the current query and provide some visual
 * feedback to the end-user on page location and navegation.
 */
pageData?: { currentPage: number, pageTotal: number, itemsPerPage: number };
/**
 * Callback to notify parent component of the action of a user clicking
 * on a change page button. If pageData is not provided, this will never
 * be called as the end-user will have no way of performing said action.
 */
onPageChanged?: (pageNumber: number) => void;
/**
 * Artworks that should appear as selected within this panel. Note that
 * it is necessary to provide a complete list of enriched artworks and
 * not just a list of ids, as this component does not keep an internal
 * state to store artworks that are not currently displayed in
 * displayedArtworks.
 */
selectedArtworks: ArtworkData[];
/**
 * Callback to the parent of this panel indicating that an artwork has
 * been selected within thepanel. This applies only to artworks that are
 * currently displayed (from displayedArtworks), which is why this callback
 * limits itself to providing the id of the artwork as only parameter.
 */
onArtworkSelected?: (artwork: ArtworkData) => void;
/**
 * Callback to the parent of this panel indicating that an artwork has
 * been deselected from the panel. This applies to both deselecting from
 * currently displayed artworks and from panel of currently selected
 * artworks (that is, it reflects changes to both displayed and selected
 * Artworks).
 */
onArtworkDeselected?: (artworkId: string) => void;
/**
 * Callback to the parent of this panel indicating that an artwork has
 * been clicked from the panel. Note that this is different
 * from selection, as clicking on an artwork just means that the user
 * wishes to explore it (e.g. to get more information about it).
 */
onArtworkClicked?: (artworkId: string) => void;
/**
 * Filter applied in this search, if any. This is used to render a list of
 * filters on top of the current search display and allow users to see
 * which filters have been applied until now and remove them if needed.
 * Trying to remove a filter will trigger the onFilterRemoved callback,
 * if provided.
```

```
   */
  appliedFilter?: GetArtworksFilter;
  /**
   * Callback to the parent of this panel indicating that the user has
   * performed a search action, with the text introduced in the
   * searchbox when requesting the search. Searchbox text IS stateful,
   * and will be kept in the internal state of the component.
   */
  onSearchPerformed?: (searchText: string) => void;
  /**
   * Dictionary containing all different, unique filter fields that the
   * user might access, as well as the different values that each of
   * those filters considers. E.g. "Date" -> ["1997", "1998", ...].
   * Each of the entries provided here will be used to generate a filtering
   * subpanel with up to maxOptionsShown entries each.
   */
  uniqueFilterFields?: Map<string, { value: string; count: number }[]>;
  /**
   * Callback to notify parent component of the selection of a filter within
   * the filter panel. Receives two arguments as parameters, namely field
   * and filter, to denote the name of the field for which a filter was
   * selected (e.g. 'Date') and the value of the filter applied
   * (e.g. '1997'), respectively.
   */
  onFilterApplied?: (field: string, filter: string) => void;
  /**
   * Callback to notify parent component of the deselection of a filter
   * within this component. Receives two arguments as parameters, namely
   * field and filter, to denote the name of the field for which a filter
   * was deselected (e.g. 'Date') and the value of the filter removed
   * (e.g. '1997'), respectively.
   */
  onFilterRemoved?: (field: string, filter: string) => void;
  /**
   * Callback to notify parent component of the deselection of all filters
   * within this component.
   */
  onClearFilters?: () => void;
  /**
   * Can be used to render a loading overlay on top of this component.
   * Useful while fetching the required artworks from the API. If not
   * specified, it will be treated as false.
   */
  loading?: boolean;
};
```

4.1.8. Select Artwork

Panel used to display a list of available artworks to choose from, as well as to provide a full-size display of the selected item’s image. This list is provided to the user in the form of a scrollable sequence of artworks' titles, which may be clicked on to select the artwork and trigger a callback with its id as parameter to notify the component's parent of the performed selection. Only one artwork may be selected at any given time, and the image pointed at by its source will be displayed on the right sub-panel of this component. Selecting an artwork will NOT change the currently selected item unless selectedArtwork prop is modified in parent (component is not stateful). If no artwork is selected, then the following page will be showed.

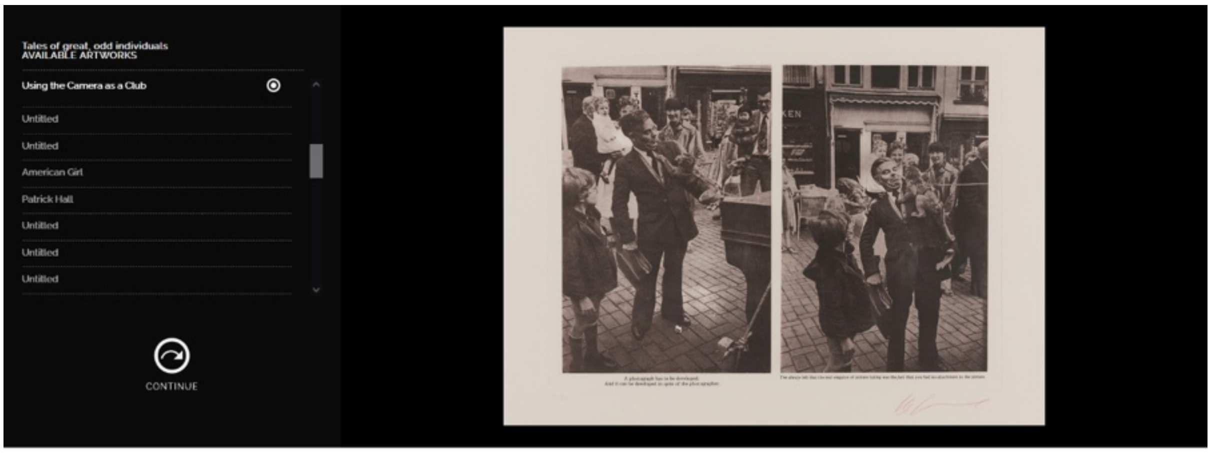
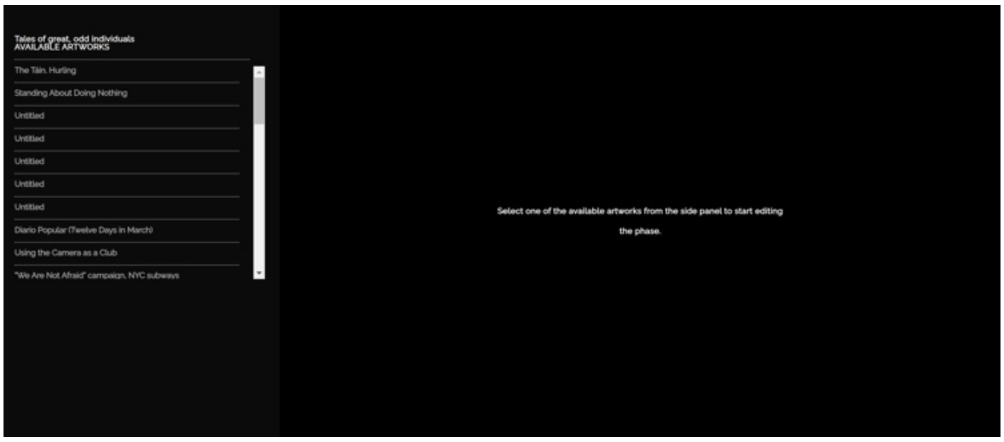


Figure 59. Artwork selection with a selected artwork.


Tales of great, odd individuals
AVAILABLE ARTWORKS
The Táin. Hurling
Standing About Doing Nothing
Untitled
Untitled
Untitled
Untitled
Untitled
Diario Popular (Twelve Days in March)
Using the Camera as a Club
"We Are Not Afraid" campaign, NYC subways

Select one of the available artworks from the side panel to start editing the phase.

Figure 60. Artwork selection page in empty mode.

Props needed for component:

```
export interface SelectArtworkProps {
  /**
   * Array of objects adhering to the ArtworkData interface, each of them
   * containing at least an artwork's id, author, title, and image source.
   * Image source will be used to render the selected artwork's preview.
   */
  imagesData: ArtworkData[];
  /**
   * Id of the currently selected artwork, or undefined if none of them is
   * selected at the time. Must match the id of one of the artworks declared
   * in imagesData (otherwise behaviour will be the same as when passing in
   * undefined in this prop).
   */
  selectedArtwork: string | undefined;
  /**
   * Callback that will be used whenever an artwork is selected in the
   * list of available artworks, with the id of the selected artwork as a
   * parameter.
   * @param artworkId unique identifier of selected artwork within imagesData.
   */
  onArtworkSelected: (artworkId: string) => void;
  /**
   * Callback that will be triggered when the continue button on the panel
   * is clicked.
   */
  onNextClicked?: () => void;
  /**
   * Title to display on top of the list of available artworks.
   */
  titleText: string;
};
```

4.1.9. Selected Artworks Pop Up

This component serves as a popup to show a set of selected artworks in a grid view. The purpose is to give the users an opportunity to view all their selections in a single page and be able to edit them by removing the undesired artworks.

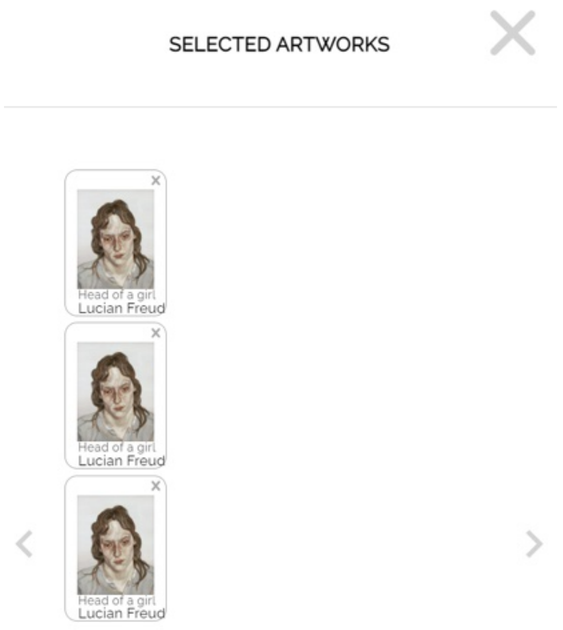


Figure 61

Props needed for component:

```
export interface SelectedArtworksPopupProps {
  /** Artworks that should be displayed by this Component. This is not
   * intended to be a comprehensive list of all artworks returned from
   * a big query, but rather a slice of a desired handpicked selection cards.
   */
  artworks: ArtworkData[];
  /** Callback with the id of an artwork as a parameter, which the user wants
   * to remove from the list.
   */
  onArtworkRemoved: (artworkId: string) => void;
  /** Callback to the parent to close/open the popup. */
  setPopupOpen: React.Dispatch<React.SetStateAction<boolean>>;
};
```

4.1.10. Show Filters

Component to display an array of applied filters for a given artwork search. Provides an option to remove any given filter from the array, as well as a button to clear all filters simultaneously.

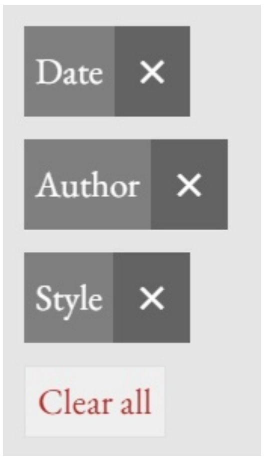


Figure 62

Props needed for component:

```
export interface ShowFiltersProps {
  /**
   * Currently applied filters. each entry must contain the name of the field
   * over which the corresponding filter is applied, and the value of the
   * filter itself. Fields will not be displayed, but they are needed to
   * notify the parent component about the removal of a given filter.
   */
  filters: { field: string, filter: string }[];
  /**
   * Callback to notify parent component of the deselection of a filter
   * within this component. Receives two arguments as parameters, namely
   * field and filter, to denote the name of the field for which a filter
   * was deselected (e.g. 'Date') and the value of the filter removed
   * (e.g. '1997'), respectively.
   */
  onFilterDelete?: (field: string, filter: string) => void;
  /**
   * Callback to notify parent component of the deselection of all filters
   * within this component.
   */
  onClear?: () => void;
};
```

4.2. Forms

4.2.1. Checkbox Group Input

This component is used to show a group of selectable items. This group will have a name as the "fieldText" and each checkbox is provided with a label. A callback is responsible for the functionality of the check change.

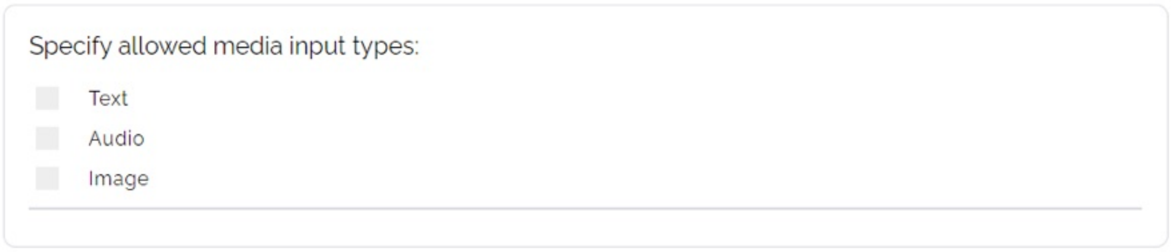

Specify allowed media input types:
- Text
- Audio
- Image

Figure 63. List of available checkboxes.

Props needed for component:

```
export interface CheckBoxGroupInputProps {
  /** Name of the checkbox field. */
  fieldText: string;
  /** Array where each component has a label for a separate checkbox. */
  labelList: string[];
  /** Array where it's saved if each label is checked or not. */
  checked: boolean[];
  /** Array containing labels of the checkboxes that are checked by default.
   */
  initialAllowedInputTypes?: string[];
  /** Callback that is responsible for the action after checkbox
   * toggles. It accepts the checkbox label as its input.
   */
  onCheckBoxToggled: (label: string) => void;
```

4.2.2. Checkbox Input

This component builds a checkbox with a default value, a label, variable box size and font.

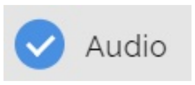


Figure 64. Checkbox selected.

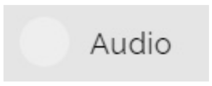


Figure 65. Checkbox unselected.

Props needed for component:

```
export interface CheckBoxInputProps {
  /** Default value of the checkbox. */
  checked?: boolean;
  /** Label of the checkbox. */
  labelText: string;
  /** String for the size of the checkbox in pixels. If not used then by
   *  default it will be “25px”. */
  boxSize?: string;
  /** The style which the checkbox will be presented with */
  style?: 'radio' | 'checkbox';
  /** String containing the font of the label of the checkbox. */
  textFont?: FlattenSimpleInterpolation;
  enabled?: boolean;
  /** Callback with the state of the checkbox as a parameter and then
   *  reverses it. */
  onCheckedChange: (checked: boolean) => void;
};
```

4.2.3. Integer Range Slider

This Component is responsible for making a range of numbers. This range will have a [min, max] set as the start and end point of the range (e.g. [0,100]). Also, it is possible to choose initial numbers inside the range as the primary start and end points (e.g. [10,50]). There is another optional feature available as “step” to set the gaps when changing the values.

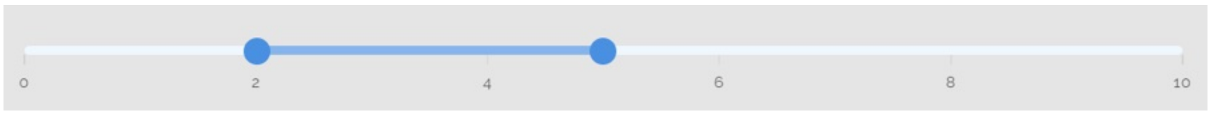


Figure 66. Range slider with step size 2, from 0 to 10.

Props needed for component:

```
export interface IntegerRangeSliderProps {
  /** Minimum value of the range. */
  min: number;
  /** Maximum value of the range. */
  max: number;
  /** Value for the size of the jumps when changing the range.  */
  step?: number;
  /** Initial value for the minimum range. */
  initialMin?: number;
  /** Initial value for the maximum range. */
  initialMax?: number;
  /** Callback to inform the parent of the component that the minimum value
   *  has changed. */
  onMinValueChange?: (min: number) => void;
  /** Callback to inform the parent of the component that the maximum value
   *  has changed. */
  onMaxValueChange?: (max: number) => void;
};
```

4.2.4. Search Bar

This component builds a simple search bar. An optional string as the primary background text can be used whenever value is set to an empty string, or undefined. The search bar will inform the component parent the search query has changed.

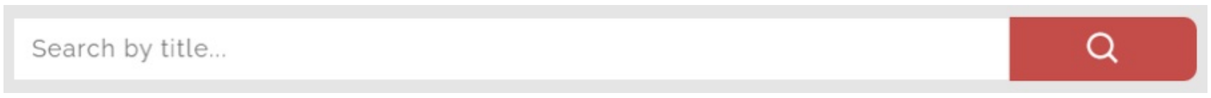


Figure 67. Search Bar used in menus.

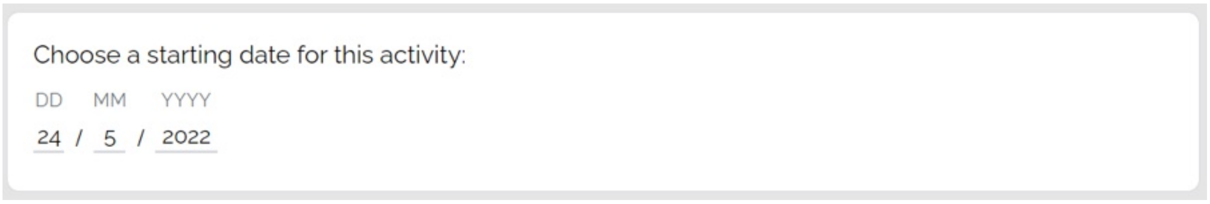
Props needed for component:

```
export interface SearchBarProps {
  /**
   * Callback to the parent of this panel indicating that the user has
   * performed a search action, with the text introduced in the searchbox
   * when requesting the search. Searchbox text is stateful, and will
   * be kept in the internal state of the component.
   */
  onSearchPerformed?: (searchText: string) => void;
  /** Text to display when the value is empty or undefined */
  placeholder?: string;
};
```

4.3. Cards

4.3.1. Calendar Input Card

A box to get a date as in input from the user. The top text will indicate what is this date about and if it is mandatory to provide it or not (by showing a red star). In the case that the date is mandatory and the user does not give an input, proper error text will be shown at the button of the box.



Choose a starting date for this activity:

DD MM YYYY
24 / 5 / 2022

Figure 68. Calendar input card.

Props needed for component:

```
export interface CalendarInputCardProps {
  /** Text rendered on top of the component as a
   * prompt for the user, indicating what date they should choose.
   */
  promptText: string;
  /** Callback to notify the parent when the value of the input field has
   *  changed.*/
  onChange?: (date: Date | undefined) => void;
  /** Callback to notify the parent when the enter key is pressed while the
   *  component is focused. */
  onEnterPress?: () => void;
  /** Default date that will be shown on the loading page before the user
   *  inserts any date. */
  initialDate?: Date | undefined;
  /** Whether this field is considered required within the overall form
   *  (used to display an asterisk). */
  required?: boolean;
  /* True if user tried to submit the form without filling a required field */
  requiredAlert?: boolean;
}
```

4.3.2. Checkbox Group Input Card

This component provides a list of check boxes as options that the user can choose from. At the top of the component a text will be shown to inform the user what is the selection about. If having at least one input is required a red star will be shown in front of the text. Moreover if the user does not provide at least one input in this case, proper error message will be shown below the options.

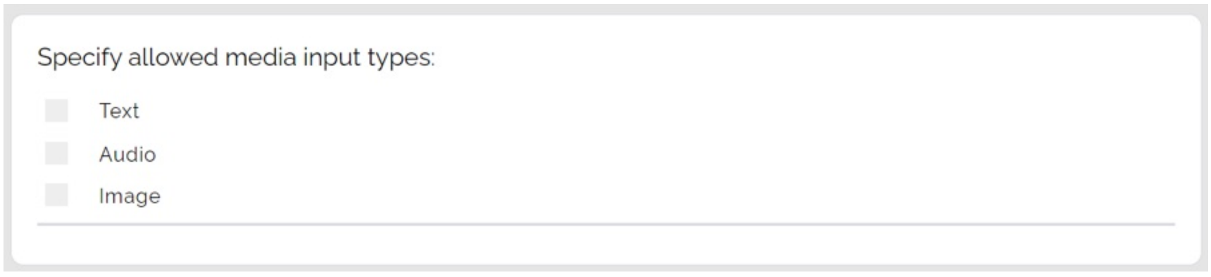

Specify allowed media input types:
- Text
- Audio
- Image

Figure 69. Checkbox group input card.

Props needed for component:

```
export interface CheckBoxGroupInputCardProps {
  /** Main text rendered on top of the component as a prompt for the user,
   *  indicating what they must check in the field */
  promptText: string;
  /** Callback to use whenever the value of a checkbox changes. Recieves the
   *  name of the field as a parameter. */
  onFieldToggle?: (field: string) => void;
  /** Array which contains the name of the fields checked. */
  checked?: string[];
  /** Array containing the name of all the checkboxes. */
  fields: string[];
  /** Variable wich indicates if the field is considered required within the
   *  overall form (used to display an asterisk) */
  required?: boolean;
  /** Alert used when the user tries to submit the form without entering a
   *  required value. As feedback the appearance of the card is modified. */
  requiredAlert?: boolean;
}
```

4.3.3. Image Upload Card

This component can be used in order to ask for a file from the user. Proper text message needs to be given on the top of this card. Similar to the previous cards there is a red start indicating if uploading a file is mandatory or not.

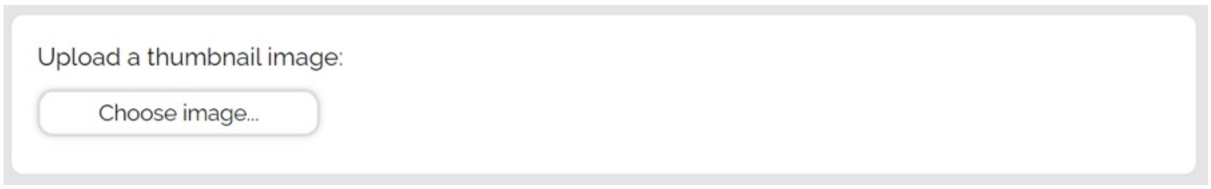


Figure 70. Image upload card.

Props needed for component:

```
export interface ImageUploadCardProps {
  /** Main text rendered on top of the component as a prompt for the user,
   *  indicating what they must upload. */
  promptText: string;
  /** Callback to use whenever the value of the input field is changed. */
  onChange?: (file: File | undefined) => void;
  /** Callback to use whenever the enter key is pressed while the component is
   *  focused. */
  onEnterPress?: () => void;
  /** Default file path that will be loaded on the page before the user
   *  uploads any file. */
  initialSrc?: string;
  /** Default file that will be uploaded on the page before the user uploads
   *  any file. */
  initialFile?: File;
  /** Whether this field is considered required within the overall
   *  form (used to display an asterisk). */
  required?: boolean;
  /* True if the user tried to submit the form without filling a
   *  required field. */
  requiredAlert?: boolean;
}
```

4.3.4. Long Text Input Card

A multiline textbox to take an input from the user in a text format. The place holder will be shown on the text bar if there is no input from the user (e.g. “Description...”). The red star indicates that this is an obligatory field which in case of remaining empty will result into a error message (e.g. “This question is required.”). The maximum length for the number of characters that the user may type can be limited.

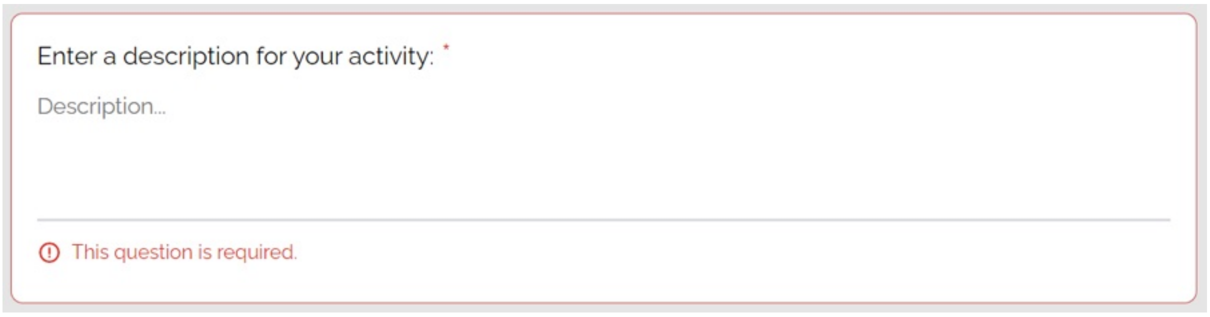

Enter a description for your activity: *
Description...
This question is required.

Figure 71. Long text input card.

Props needed for component:

```
export interface LongTextInputCardProps {
  /** Main text rendered on top of the component as a prompt for the
   *  user, indicating what they must type into the field. */
  promptText: string;
  /** Text to display when the value is empty or undefined. */
  placeholder?: string;
  /** Callback used whenever the value of the input field is changed. */
  onChange?: (value: string) => void;
  /** Callback used whenever the “Enter” key is pressed while the
   *  component is focused. */
  onEnterPress?: () => void;
  /** Maximum number of characters to allow within the input area. */
  maxLength?: number;
  /** Current value of the input field. Needs to be changed after onChange
   *  events to be kept in sync with internal state. */
  value?: string;
  /** Whether this field is considered required within the overall
   *  form (used to display an asterisk). */
  required?: boolean;
  /** Modifies the appearance of the card to reflect that the user
   *  tried to submit the form without entering a value for this field. */
  requiredAlert?: boolean;
}
```

4.3.5. Range Input Card

This component is a slide bar for taking a range of numbers in. The text on the top of the box will indicate what is the range of numbers about and the range can have minimum and maximum limits. Changing each of these values may have a separate call back.

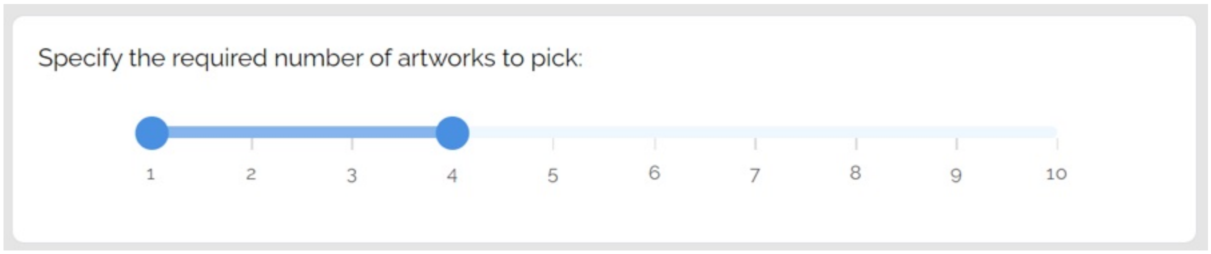


Figure 72. Numerical range input card.

Props needed for component:

```
export interface RangeInputCardProps {
  /** Main text rendered on top of the component as a prompt for the
   *  user, indicating what is the range that they have to choose. */
  promptText: string;
  /** Minimum value, is the start point of the range. */
  min: number;
  /** Maximum value possible, is the end point of the range. */
  max: number;
  /** Initial value for the range. */
  initialMin?: number;
  /** Initial maximum value for the range. */
  initialMax?: number;
  /** Callback to use when the minimum value is changed. */
  onMinValueChange?: (value: number) => void;
  /** Callback to use when the maximum value is changed. */
  onMaxValueChange?: (value: number) => void;
  /** If this field is considered required within the overall
   *  form (used to display an asterisk). */
  required?: boolean;
  /* True if user tried to submit the form without filling a required field */
  requiredAlert?: boolean;
}
```

4.3.6. Short Text Input Card

This component is the single line form of the “LongTextInputCard.tsx” with the same attributes.

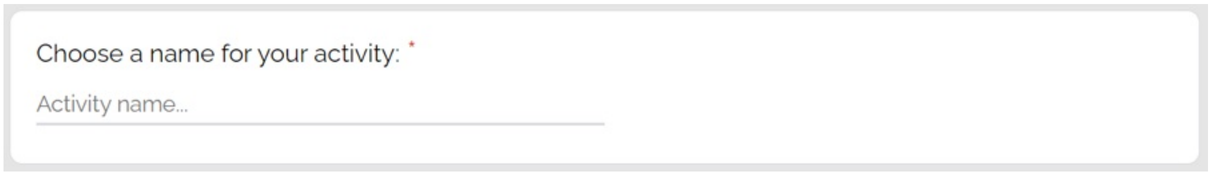


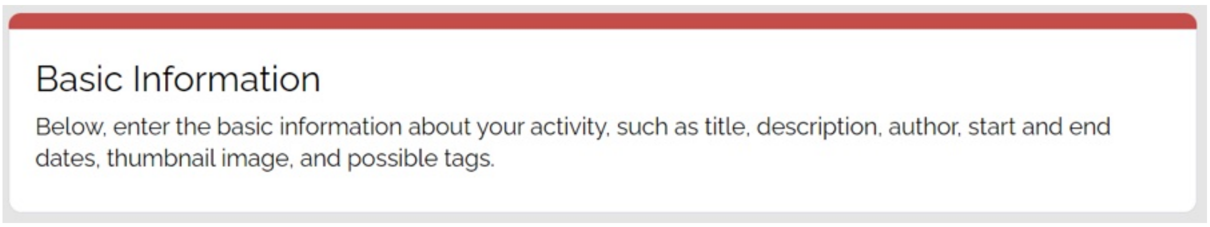
Figure 73. Short text input card.

Props needed for component:

```
export interface ShortTextInputCardProps {
  /** Main text rendered on top of the component as a prompt for the user,
   *  indicating what they must type into the field. */
  promptText: string;
  /** Text to display when the value is empty or undefined. */
  placeholder?: string;
  /** Callback to the parent used whenever the value of the input field is
   *  changed. */
  onChange?: (value: string) => void;
  /** Callback to the parent when the "Enter" key is pressed while the
   *  component is focused. */
  onEnterPress?: () => void;
  /** Maximum number of characters allowed within the input area. */
  maxLength?: number;
  /** Current value of the input field. Needs to be changed after onChange
   *  events in order to be synchronized with the internal state. */
  value?: string;
  /** If the field is required within the overall form (used to display
   *  an asterisk). */
  required?: boolean;
  /** If the user tries to submit the form without a required field this
   *  changes the appearance of the card. */
  requiredAlert?: boolean;
  /** Alert message to be displayed when required alert is set to true. */
  alertMessage?: string;
  /** Whether this field represents a password (should be hidden) */
  isPassword?: boolean;
  /** Proportion of container width to be used for input.
   *  50% (0.5) by default */
  width?: number;
}
```

4.3.7. Step Title Card

This component is designed for explaining and providing additional information. The component will have a title for the card and the information part. There is no input or interaction needed from the user.


Basic Information

Below, enter the basic information about your activity, such as title, description, author, start and end dates, thumbnail image, and possible tags.

Figure 74. Title card.

Props needed for component:

```
export interface StepTitleCardProps {
  /** Step's title. */
  stepTitle: string;
  /** Short description of what needs to be done within the step. */
  stepDescription?: string;
  /** Whether to enable action button at the top of the card. */
  enableAction?: boolean;
  /** Name of the action to be performed. If not specified, no action will
   *  be available (button won't be rendered). */
  actionName?: string;
  /** Callback to parent component indicating that the action has been
   *  requested by the user. */
  onActionCliked?: () => void;
  children?: React.ReactNode;
}
```

4.3.8. Tags Input Card

This component provides the user with a card to add tags to it. The title will explain what are the tags intended for and the user can add tags by clicking on the "New tag" button or remove the added tags one by one by clicking on the remove button on each one of them. The number of tags that the user can add or remove can be limited to a range by giving minimum and maximum numbers.

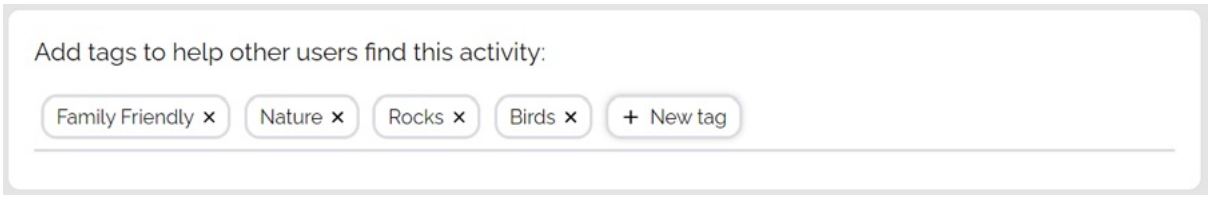


Figure 75. Tags input card.

Props needed for component:

```
export interface TagsInputCardProps {
  /** Main text rendered on top of the component as a prompt for the user,
   *  indicating what they must do with the tags manager. */
  promptText: string;
  /** Callback to use when a card has been created or removed. The
   *  parameter `value` provides a complete list of tags in use. */
  onChange?: (value: string[]) => void;
  /** Maximum number of tags allowed. */
  maxTags?: number;
  /** Minimum number of tags allowed. */
  minTags?: number;
  /** Current value of the input field. Needs to be changed after onChange
   *  events in order to be synchronized with the internal state. */
  value?: string[];
  /** If this field is considered required within the overall form (used to
   *  display an asterisk). */
  required?: boolean;
  /** Modifies the appearance of the card to reflect that the user tried to
   *  submit the form without entering a required value. */
  requiredAlert?: boolean;
}
```

4.4. Layouts

4.4.1. Content Card

A simple box with a title to be used as a parent for other more complex components. It is possible to put a title for this card, set a width and assign a maximum width for it.

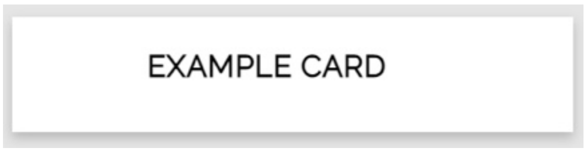


Figure 76

Props needed for component:

```
export interface CardProps {
  /** Title of the box. */
  cardTitle?: string;
  /** Alignment of the title. */
  titleAlign?: 'left' | 'right' | 'center' | 'justify';
  /** Width of the box. */
  width?: string;
  /** Maximum width possible for the card */
  maxWidth?: string;
  /** Content orientation of the card. */
  flexDirection?: 'row' | 'column';
  /** React node component that can act as a child to the box. */
  children?: React.ReactNode;
};
```

4.4.2. Loading Overlay

This component will show a message during the waiting time of a page load.

Figure 77. Loading overlay.

Props needed for component:

```
export interface LoadingOverlayProps {
  /** Message to be display. */
  message?: string;
};
```

4.5. Navigation

4.5.1. Activity Creation Overview Panel

Panel with a sequence of steps to be fulfilled. All the steps finished are rendered with a different colour as visual feedback for the user. The steps to be fulfilled are determined by the user.

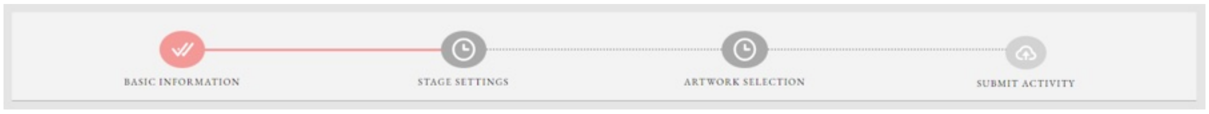


Figure 78. Representation of the steps.

Props needed for component:

```
export interface ActivityCreationOverviewPanelProps
                  extends NavigationComponentProps {
  /**
   * Minimum number of stages to allow when stage adding/removal is enabled.
   */
  minStages?: number;
  /**
   * Maximum number of stages to allow when stage adding/removal is enabled.
   */
  maxStages?: number;
  /**
   * List of available stages to select from. Each entry contains information
   * about the name of the stage and whether it has already been completed
   * by the user or not.
```

```
 * Also specifies if the stage can be removed when clicking on edit stages.
 * Names will be used to display the label for each stage.
 */
stages: { name: string, completed: boolean, canBeRemoved?: boolean }[];
/**
 * Caption/label for the last button at the end of the stage sequence,
 * generally used as a submission
 * button for the activity built during the creation process. If not
 * provided, this last button will not be rendered.
 */
finaItemCaption?: string;
/**
 * Caption/label used to add a stage panel button that will be used to
 * add new stages to the panel.
 * Has no effect if enableStageAddition is not active.
 */
addStagePanelText?: string;
/**
 * Caption/label used to edit stages panel button that will be used to
 * edit stages in the panel (remove/add/reorder).
 * Has no effect if enableStageAddition is not active.
 */
editStagesPanelText?: string;
/**
 * Caption/label used to cancel edit stages panel button that will be used
 * to disable edit mode (remove/add/reorder).
 * Has no effect if enableStageAddition is not active.
 */
cancelEditStagesPanelText?: string;
/**
 * Whether stage addition should be allowed for this activity. This
 * determines if the add stage panel is enabled when clicking on edit
 * stages. Disabled by default.
 */
enableStageAddition?: boolean;
/**
 * Whether stage removal should be allowed for this activity. This
 * determines if the remove stage icons are enabled when clicking on edit
 * stages. Disabled by default.
 */
enableStageRemoval?: boolean;
/**
 * Callback to notify parent component about the activity being submitted
 * (submit button clicked).
 */
onSubmitActivity?: () => void;
/**
 * Callback to notify parent component about a specific stage being
 * selected (by index).
 */
onStageSelected?: (index: number) => void;
```

```
  /**
   * Callback to notify parent component about the user trying to add a
   * new stage to the activity.
   */
  onAddNewStage?: () => void;
  /**
   * Callback to notify parent component about the user trying to remove a
   * stage from the activity (by index).
   */
  onRemoveStage?: (index: number) => void;
};
```

4.5.2. Page Bar

An informative component to show when there are multiple pages of items. There is always one active page and the possibility to move to the other pages.

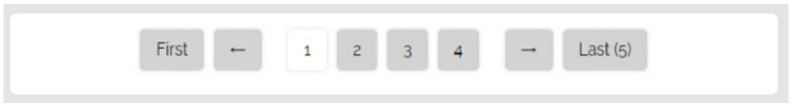


Figure 79. Page bar.

Props needed for component:

```
export interface PageBarProps {
  /** Currently displayed/selected page. */
  currentPage: number;
  /** Total number of pages to take into account */
  numberOfPages: number;
  /** Callback to notify the parent component about a page number being
   *  selected */
  onPageSelected: (page: number) => void;
};
```

4.5.3. Progress Line

An informative tab to show how many stages are there in a given sequence and which stage is active. At any given time only one stage is active and the related information of that step are being shown. It I possible to add new mid stages by clicking on the provided button. Each step will be given and number as its identifier starting from zero for the first stage and a Boolean to indicate if it is finished or not. In the following figure it can be seen that there are three stages and stage number “0” is the current Stage.

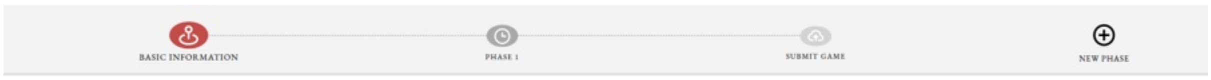


Figure 80. Sample progress line.

Props needed for component:

```
export interface ProgressLineProps {
  /**
   * Currently active item, which will be highlighted to set it apart from
   *  the rest of items.
   */
  currentItem: number;
  /**
   * List of available stages to select from. Each entry contains information
   * about the name of the stage and whether it has already been completed
   * by the user or not.
   * Also specifies if stage can be removed when clicking on edit stages.
   * Names will be used to display the label for each stage.
   */
  items: { name: string, completed: boolean, canBeRemoved?: boolean }[];
  /**
   * Caption/label for the last button at the end of the stage sequence,
   * generally used as a submission button for the activity built during
   * the creation process. If not provided, this last button will not
   * be rendered.
   */
  finalItemCaption?: string;
  /**
   * Whether we are in edit mode, allowing reorder/remove items
   * (if removalAllowed == true).
   */
  editing?: boolean;
  /**
   * Whether we are allow to remove elements in edit mode or not.
   */
  removalAllowed?: boolean;
  /**
   * Callback to notify the parent component about a specific item being
   * selected (by index).
   */
  onItemSelected?: (index: number) => void;
  /**
   * Callback to notify the parent component about the removal of an item
   * from the list by the user (by index).
   */
  onRemoveItem?: (index: number) => void;
  /**
   * Callback to notify the parent component about the task being submitted
   * (submit button clicked).
   */
  onSubmit?: () => void;
};
```

4.5.4. Stage Button Panel

Customizable button which can be enable or disable, being rendered differently in each case. It has three different funtionalities, it can be used for editing, adding or cancel. The text below is also customizable in case it's wanted to be written in other languages.

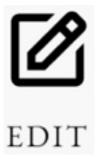


Figure 81. Stage button enabled.

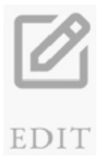


Figure 82. Stage button disabled.

Props needed for component:

```
export interface StageButtonPanelProps {
  /**
   * Whether this panel will react to user clicks (and be rendered as
   * if it does).
   */
  enabled: boolean;
  /**
   * Text to display beneath the panel's icon.
   */
  panelText?: string;
  /**
   * Icon to display in this panel.
   */
  panelIconType: 'edit' | 'add' | 'cancel';
  /**
   * Callback to parent component to notify about user clicking on
   * the stage button.
   */
  onButtonClicked?: () => void;
};
```

5. Components with specific usage

5.1. GAM Game

5.1.1. Artwork Decoration Panel

Customizable panel where it can be added tags and emojis to an artwork piece. The intention is to let the viewers express themselves and share with others how they felt about the artwork.

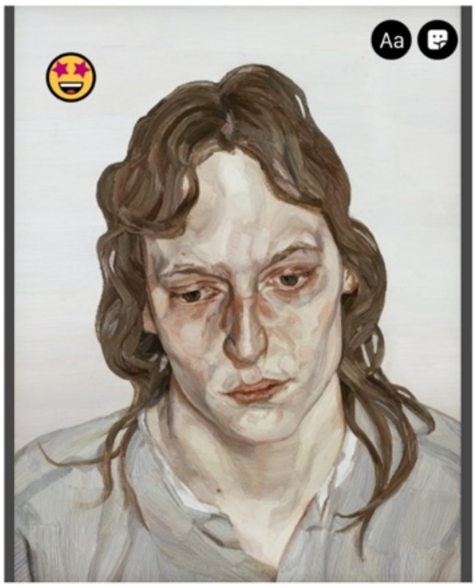


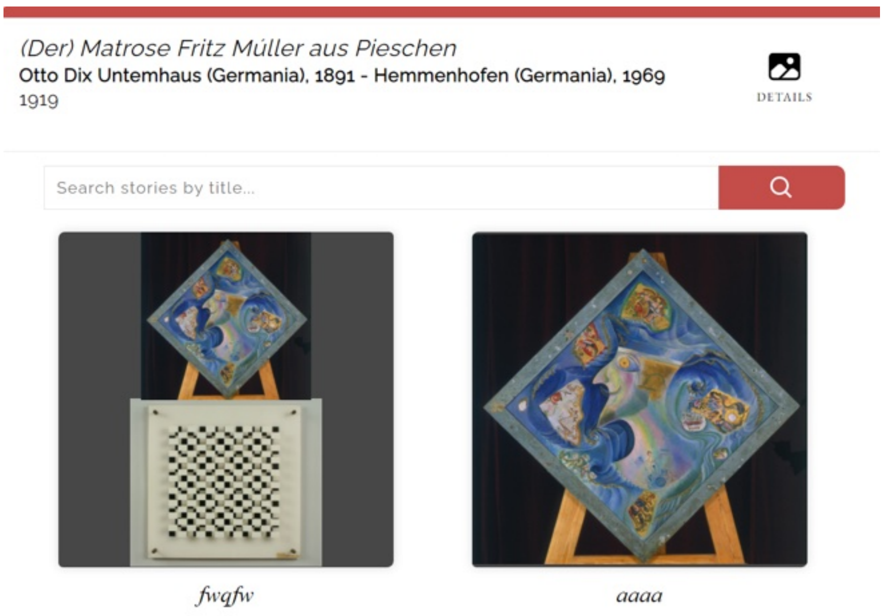
Figure 83. Artwork decorated with an emoji.

Props needed for component:

```
export interface ArtworkDecorationPanelProps {
  /** Controls if the panel can be edited. */
  editEnabled?: boolean;
  /** Path to find the source image of the artwork. */
  artworkSrc: string;
  /** Emojis shown in the panel. */
  emojis: StoryPartEmoji[];
  /** Tags shown in the panel. */
  tags: StoryPartTag[];
  /** Callback to the parent notifying when is added a new emoji. */
  onAddEmoji?: (emoji: Emoji) => void;
  /** Callback to the parent notifying when is added a new tag. */
  onAddTag?: (tag: string) => void;
  /**
   * Called with position expressed in relative terms (% of image's height
   * and width)
   */
  onMoveEmoji?: (index: number, pos: Position) => void;
  /**
   * Called with position expressed in relative terms (% of image's height
   * and width)
   */
  onMoveTag?: (index: number, pos: Position) => void;
}
```

5.1.2. Artwork Stories List

This component renders a list of stories from different users, all of them associated to a given artwork. Allows to share how each viewer perceives the artwork.



(Der) Matrose Fritz Müller aus Pieschen
Otto Dix Untemhaus (Germania), 1891 - Hemmenhofen (Germania), 1969
1919

Figure 84. Stories list browser interface.

Props needed for component:

```
interface ArtworkStoriesListProps {
  /** stories associated to this artwork */
  stories: GamGameStoryDefinitionData[];
  /** Artwork data to be used when rendering the stories */
  artworks: ArtworkData[];
  /** Specific artwork that all stories within this component will include */
  currentArtwork: ArtworkData;
  /** Callback to parent specifying that a given story (by id) has
   *  been selected by the user */
  onStorySelected?: (storyId: string) => void;
  /** Callback to parent specifying that the user wishes to create a
   *  new story */
  onCreateStoryClicked?: () => void;
  /** Callback to parent specifying that the user wishes to switch to
   *  details mode */
  onShowDetailsClicked?: () => void;
};
```

5.1.3. Artwork List

The artworks are rendered in a list with a search bar useful to find a specific piece of art. The artworks on the list can be selected.

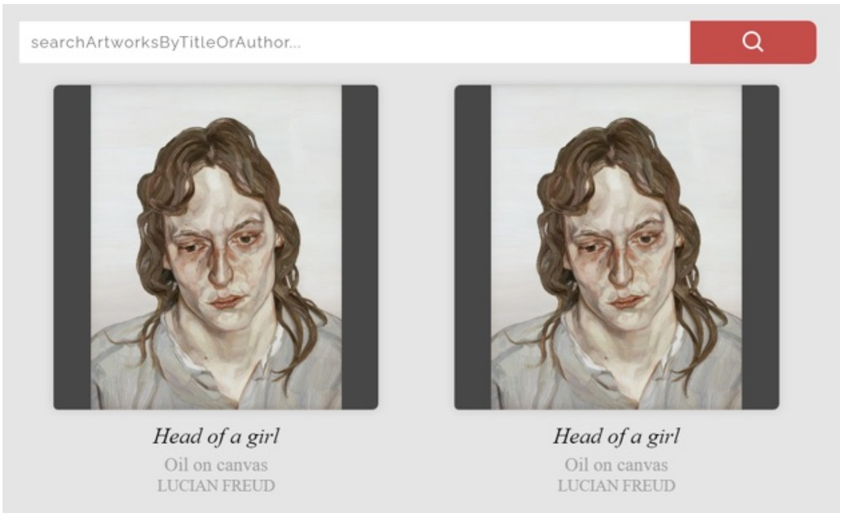


Figure 85. List of artworks.

Props needed for component:

```
export interface ArtworksListProps {
  /** Artworks to be rendered within this component */
  artworks: ArtworkData[];
  /** Callback to parent specifying that a given artwork with artwork has
   *  been selected */
  onArtworkSelected?: (artworkId: string) => void;
};
```

5.1.4. Create Story Part

A component of the user story creation flow, it is intended to specify a part of the given story based on the user's response to a series of templated prompts and, optionally, a set of interactions with the chosen artwork in the form of emojis or tags superimposed on the image.

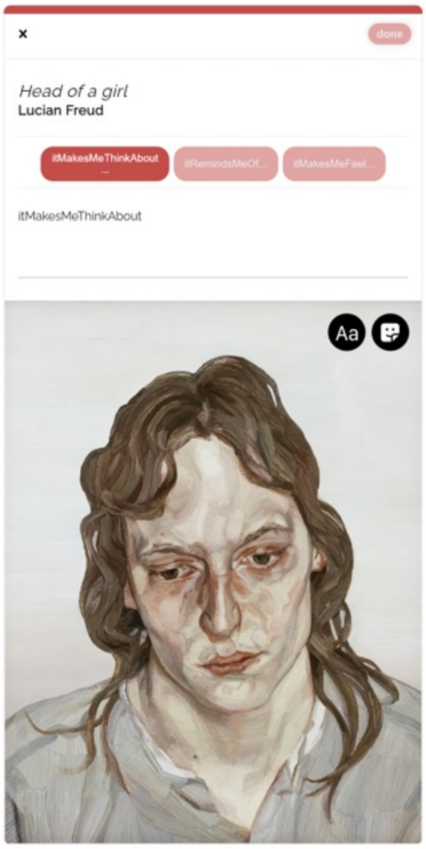


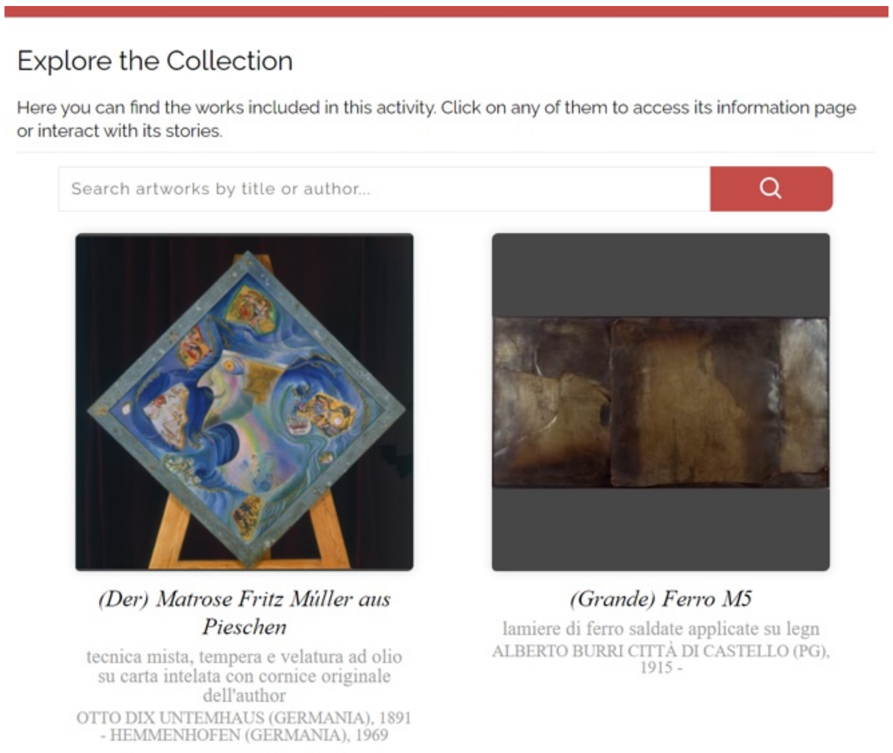
Figure 86. Create a story part interface.

Props needed for component:

```
interface CreateStoryPartProps {
  /** Artwork over which the current part will talk about */
  artwork: ArtworkData;
  /** Callback to parent that will be triggered when the user clicks on
   *  “done” with a valid part definition (passed as part param) */
  onSubmitPart?: (part: GamGameStoryPart) => void;
  /** Callback to parent specifying that the user wishes to give up on
   *  this story */
  onQuit?: () => void;
};
```

5.1.5. Stories List

This component renders a list of stories from different users and different artworks. This allows to share how each viewer perceives the artworks available.



Explore the Collection

Here you can find the works included in this activity. Click on any of them to access its information page or interact with its stories.

(Der) Matrose Fritz Müller aus Pieschen

tecnica mista, tempera e velatura ad olio su carta intelata con cornice originale dell'author

OTTO DIX UNTEMHAUS (GERMANIA), 1891 - HEMMENHOFEN (GERMANIA), 1969

(Grande) Ferro M5

lamiere di ferro saldate applicate su legn

ALBERTO BURRI CITTÀ DI CASTELLO (PG), 1915 -

Figure 87. List of stories to browse through.

Props needed for component:

```
export interface StoriesListProps {
  /** Stories to be rendered within this component */
  stories: GamGameStoryDefinitionData[];
  /** Artworks used within these stories */
  artworks: ArtworkData[];
  /** Callback to parent specifying that a given story with storyId
   *  has been selected */
  onStorySelected?: (storyId: string) => void;
};
```

5.1.6. Story Part View

Component which renders a story created by a user with the specify template prompt, the text added, emojis and tags. If the story is associated with other ones "Next" button can be interacted with to see them.

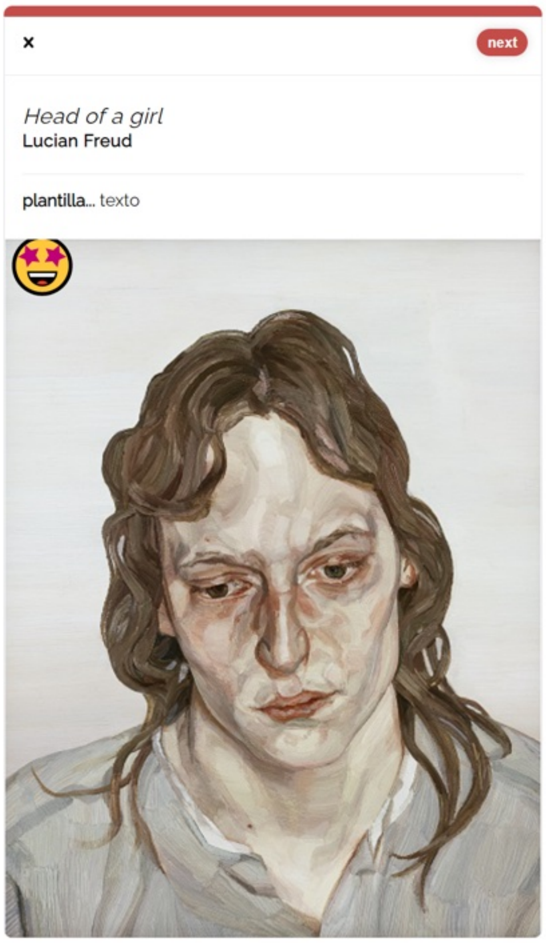


Figure 88. Visualization of a story.

Props needed for component:

```
export interface StoryPartViewProps {
  /** Story part to be rendered */
  storyPart: GamGameStoryPart;
  /** Artwork Data for the artwork included in storyPart as artworkId */
  artworkData: ArtworkData;
  /** Callback to parent component specifying that the next button has
   *  been pressed */
  onNextClicked?: () => void;
  /** Callback to parent specifying that the user is closing the story */
  onQuit?: () => void;
};
```

5.2. Viewpoints

5.2.1. Accordion Item

Item which will be part of a list of similar items. This Accordion Items can be opened to get more information about the artwork they content.

Figure 89. Drop-down item.

Props needed for component:

```
export interface AccordionItemProps {
  /** Artwork that will be shown when the item is opened. */
  artwork: Artwork;
  /** List of questions and responses about the artwork. */
  responses: Response[];
  /** If the item is open or closed, */
  defaultToggled?: boolean;
};
```

5.2.2. Artworks Component

This panel is used on several places to provide a selecting option between artworks. The artworks are being shown in an M×N grid view (in this case 3×2) from a list of artworks. To see the rest of the artworks in the list it is possible to click on next/previous buttons. A brief detail about the artworks are available on hover over them. Based on where this component is being used a different function can be called for handling a click.

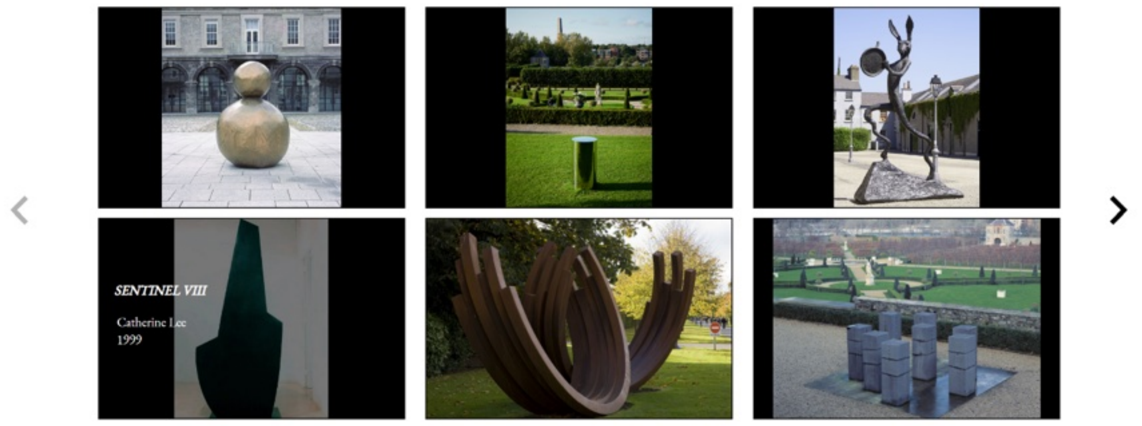


Figure 90. Grid view of artworks.

Props needed for component:

```
interface ArtworksComponentProps {
  /** List of artworks, each contain its id, name, artist, description, date,
   *  imageLoc, image, audio, notes and URL. */
  artworks: Artwork[];
  /** Callback that will be used whenever an artwork is selected in the list
   *  of available artworks, with the id of the selected artwork as a
   *  parameter. */
  onArtworkClicked?: (id: string) => void;
};
```

5.3. Natural Science Catalogue

5.3.1. Artifact Card

Component which renders a card with the image and a small piece of information about an artifact of the museum. The cards can be interacted with, being able to open, edit and delete them.

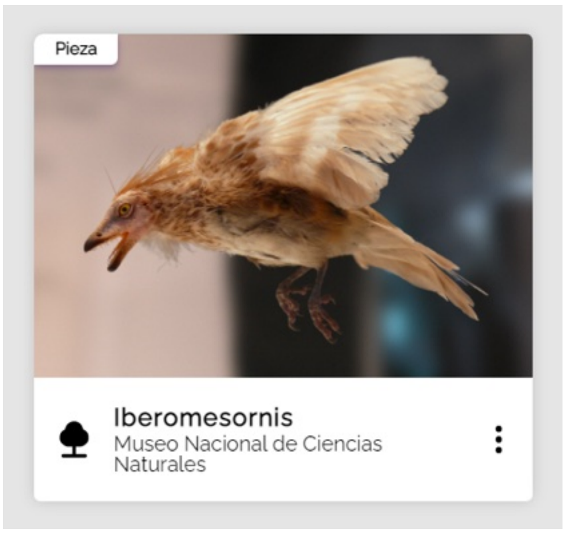


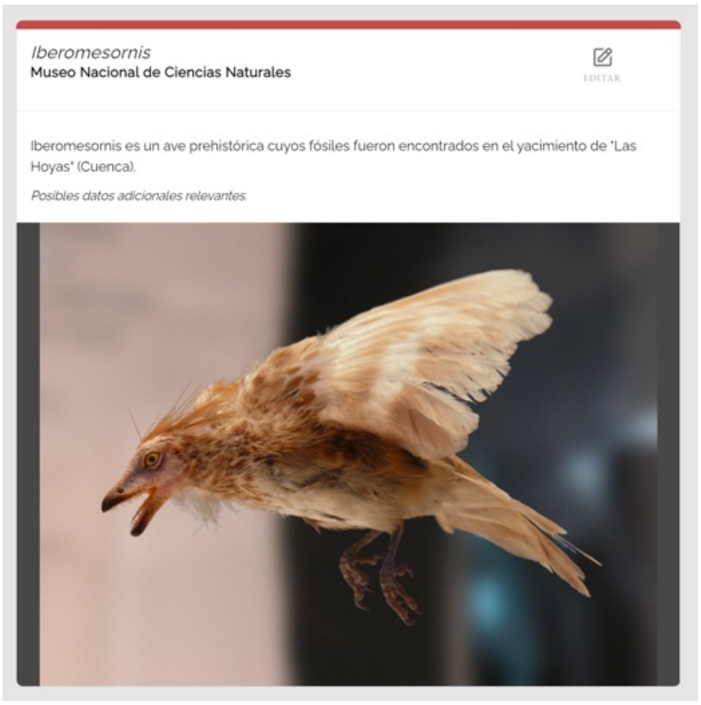
Figure 91. Visualization of the cards.

Props needed for component:

```
export interface ArtifactCardProps {
  /** Data about the artifact showing. */
  artifactData: MncnArtifact;
  /** Callback to the parent component to notify that the user has clicked
   *  over the card. */
  onOpenClicked?: () => void;
  /** Callback to the parent component to notify that the user is going to
   *  edit the card. */
  onEditClicked?: () => void;
  /** Callback to the parent component to notify that the user has deleted
   *  the card. */
  onDeleteClicked?: () => void;
}
```

5.3.2. Artifact Detail

Panel rendered with all the information about an artifact and its image. The content of the panel can be edited if the “Edit” button is enabled.


Iberomesornis
Museo Nacional de Ciencias Naturales

Iberomesornis es un ave prehistórica cuyos fósiles fueron encontrados en el yacimiento de "Las Hoyas" (Cuenca).

Posibles datos adicionales relevantes.

Figure 92. Detailed information about an artifact.

Props needed for component:

```
export interface ArtifactDetailProps {
  /** Information and details about the artifact */
  artifactData: MncnArtifact;
};
```

5.3.3. MNCN Catalogue Browsing Screen View

Interface generated with a list of selectable artifact cards with a searching bar to be able to find a specific artifact based on its title or description. The search can be filtered based on the tags selected.

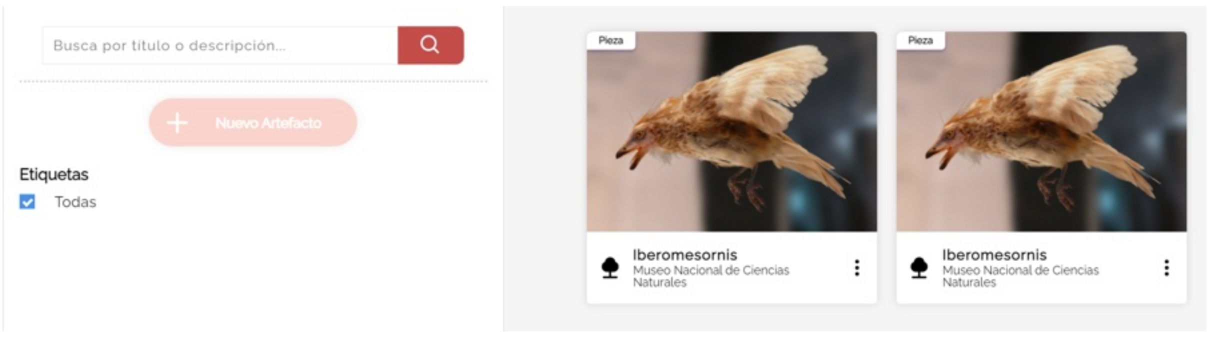

Busca por titulo o descripción...
Nuevo Artefacto
Etiquetas
Todas
Pieza
Iberomesornis
Museo Nacional de Ciencias Naturales
Pieza
Iberomesornis
Museo Nacional de Ciencias Naturales

Figure 93. Interface to browse through artifacts.

Props needed for component:

```
export interface MncnCatalogueBrowsingScreenViewProps {
  /** Artifacts list to browse through. */
  artifacts: MncnArtifact[];
}
```

5.4. Find Artwork (play treasure hunt)

5.4.1. Artwork Card

The artwork card is being used to show the details of a single artwork. This card later can be used in the grids or other components. It consists of a background image which is the artwork itself, some information about the artwork which is available on hover and a checkbox to select or unselect the item. Various callbacks can be added to the selected option based on where this card is going to be used. Since this card is mainly being use in the "Find artwork game" it provides a "status" as a prop which indicates if it is the correct artwork or not and how much point it has when finding it as the prize.

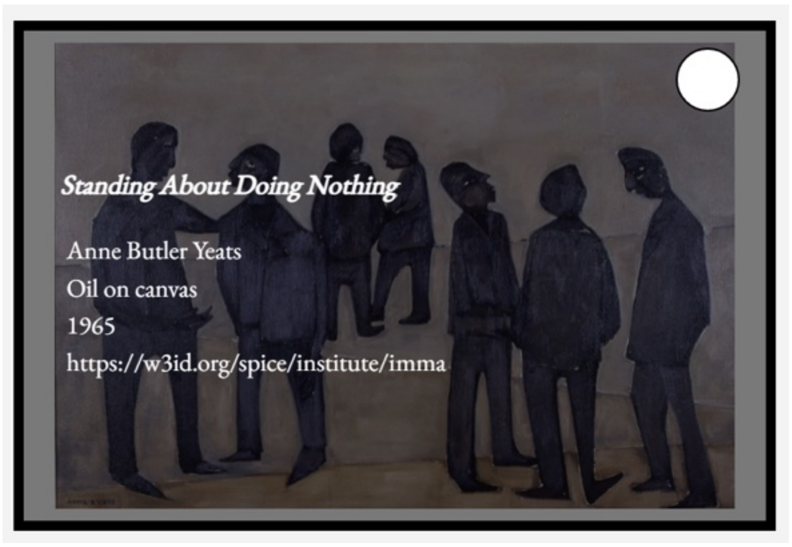

Standing About Doing Nothing
Anne Butler Yeats
Oil on canvas
1965
https://w3id.org/spice/institute/imma

Figure 94. Single artwork shown in a card.

Props needed for component:

```
export interface ArtworkCardProps {
  /** Piece of artwork to be guessed. Contains id, name, artist, description,
   *  date, imageLoc, image, audio, notes and URL.  */
  artworkData: ArtworkData;
  /** Indicates whether the card has been guessed correctly or not. */
  status: ArtworkCardStatus;
  /** When flipped it is shown if the anwser was correct or not. */
  flipped: boolean;
  /** Callback to the parent component to notify that the user has selected
   *  the card. */
  onCardSelected?: () => void;
};
```

5.4.2. Artwork Correct

This card is being use the find artwork game when the player successfully selects the correct artwork. It contains the background image and a title text to inform the player that it is the correct artwork. There is also a boolean that is being passed to acknowledge the fact that this card has been flipped.

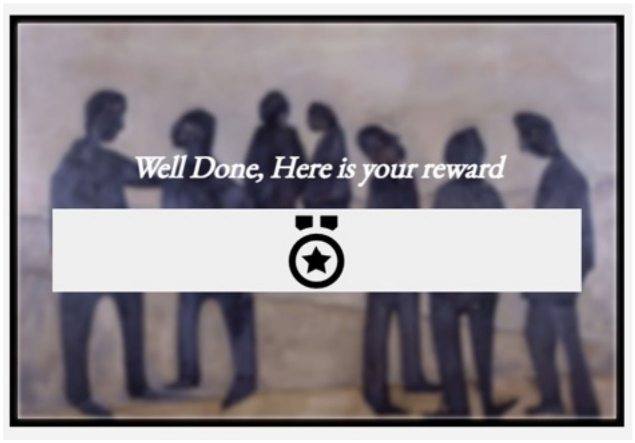


Figure 95. An artwork card when selected correct.

Props needed for component:

```
export interface ArtworkCorrectProps {
  /** If the image is flipped it's shown, otherwise is hidden. */
  flipped: boolean;
  /** Path to the source image of the artwork which is used as the
   *  background of the card. */
  image: string;
  /** Title of the artwork card. */
  title: string;
};
```

5.4.3. Artwork Failed

This card is being use the find artwork game when the player unsuccessfully selects the wrong artwork. It contains the background image and a title text to inform the player that it is the wrong artwork. There is also a boolean that is being passed to acknowledge the fact that this card has been flipped.

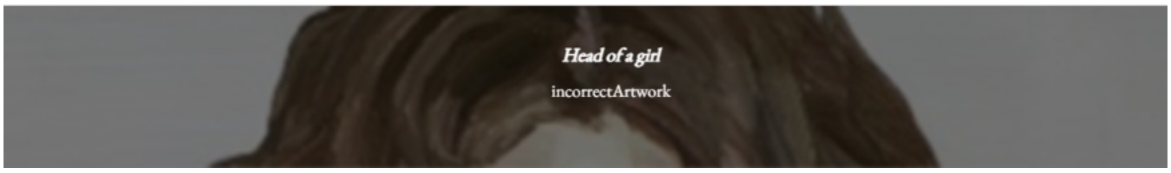


Figure 96. Visualizaton when the wrong artwork is selected.

Props needed for component:

```
export interface ArtworkFailedProps {
  /** If the image is flipped it's shown, otherwise is hidden. */
  flipped: boolean;
  /** Path to the source image of the artwork which is used as the
   *  background of the cards. */
  image: string;
  /** Title of the artwork card. */
  title: string;
};
```

5.4.4. Artwork Popup

After selection of the correct artwork this component will be shown as a popup. Complete information of the artwork is shown in this popup. Moreover, the prizes assigned for finding this specific artwork can be revealed.

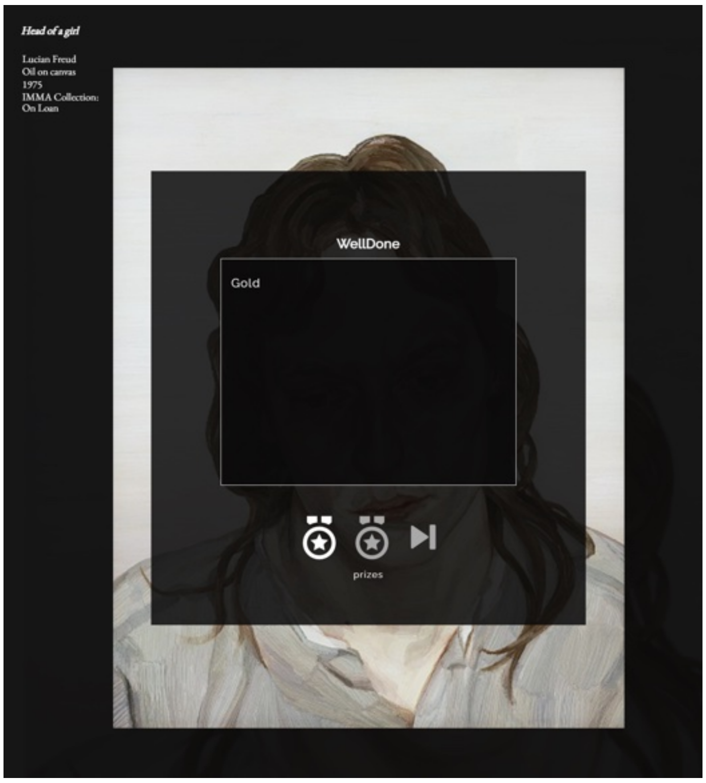


Figure 97. An artwork in popup mode.

Props needed for component:

```
export type Props = {
  /** If the image is flipped it's shown, otherwise is hidden. */
  flipped: boolean;
  /** List of prizes earned by the user. */
  prize: string[];
  /** Artwork piece correctly guessed to earn the prizes.  */
  artworkData: ArtworkData;
};
```

5.4.5. Clue Holder

This component shows the users the clues they might need to find an artwork in a game. If the user decides to use more clues some of their point will be reduced.

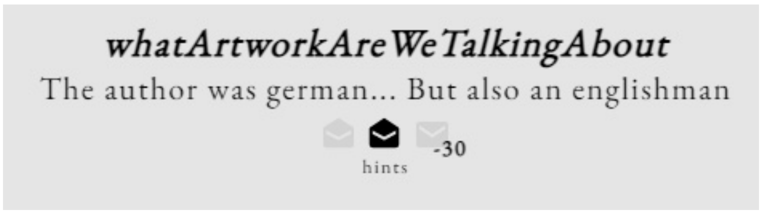

whatArtworkAreWeTalkingAbout
The author was german... But also an englishman
-30
hints

Figure 98. Available hints interface.

Props needed for component:

```
export interface ClueHolderProps {
  /** Clues to help the user guessing the artwork. */
  clues: string[];
  /** Callback to the parent component to notify if a clue was opened and
   *  which one. */
  onClueOpened?: (points: number) => void;
};
```

5.4.6. Find Artwork

This component is responsible for the complete find artwork game experience. It has all the stage information and the artworks to be displayed in the game and is checking the points of the game.

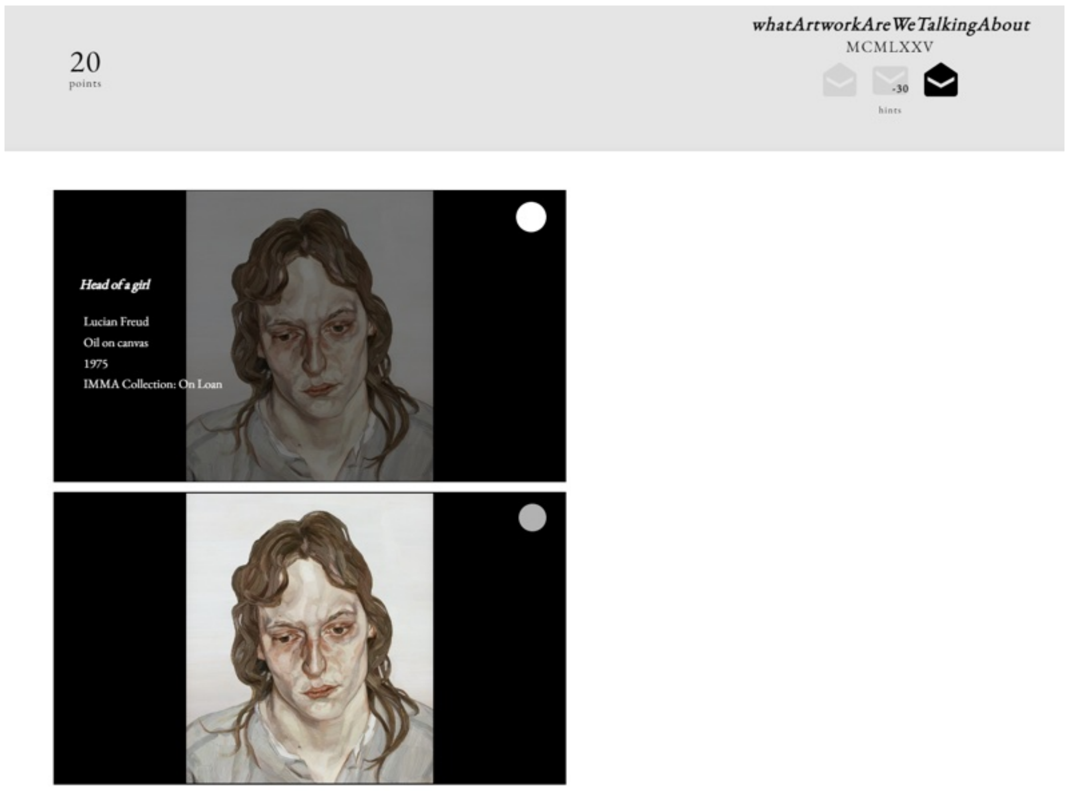


Figure 99. Game interface.

Props needed for component:

```
export interface FindArtworkProps {
  /** Groups several Artworks and clues for them to be identified. */
  stageData: StageData;
  /** List of artworks with their respective information. */
  imagesData: ArtworkData[];
  /** Callback to the parent to notify when all have been guessed. */
  onStageCompleted?: () => void;
  /** Callback to the parent to notify the points have changed. */
  onPointsUpdate?: (value: number) => void;
  /** The score achieved by the user. */
  score: number;
};
```

5.4.7. Points Panel

A simple box to show the user the total points they have.

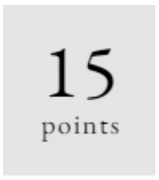


Figure 100. Representation of the score obtained.

Props needed for component:

```
export interface PointsPanelProps {
  /** Score of the user updated. */
  points: number
};
```

6. Accessibility

Web accessibility consists of making web pages usable by the maximum number of people, regardless of their knowledge or personal abilities and regardless of the technical characteristics of the equipment used to access the Web.

To reach that accessibility level there are different guidelines or guides have been developed that explain how web pages have to be created to be accessible.

These guidelines will be applied within each of the components and the guidelines will be applied according to the WCAG checklist https://www.wuhcag.com/wcag-checklist, which determines 3 levels of Web Accessibility.

The objective is to reach the Intermediate level (AA) which is the level required by websites of public organism.

According to the WCAG list, web accessibility should be focused on the following as follows:

WCAG Checklist Level A (Beginner)
1.1.1 – Non-text Content: Provide text alternatives for non-text content that serves the same purpose.
1.2.1 – Audio-only and Video-only (Pre-recorded): Provide an alternative to video-only and audio-only content.
1.2.2 – Captions (Pre-recorded): Provide captions for videos with audio.
1.2.3 – Audio Description or Media Alternative (Pre-recorded): Provide audio description or text transcript for videos with sound.
1.3.1 – Info and Relationships: Content, structure and relationships can be programmatically determined.
1.3.2 – Meaningful Sequence: Present content in a meaningful order.
1.3.3 – Sensory Characteristics: Instructions don't rely solely on sensory characteristics.
1.4.1 – Use of Colour: Don't use presentation that relies solely on colour.
1.4.2 – Audio Control: Don't play audio automatically.
2.1.1 – Keyboard: All functionality is accessible by keyboard with no specific timings.
2.1.2 – No Keyboard Trap: Users can navigate to and from all content using a keyboard.
2.1.4 – Character Key Shortcuts: Allow users to turn off or remap single-key character shortcuts.
2.2.1 – Timing Adjustable: Provide user controls to turn off, adjust or extend time limits.
2.2.2 – Pause, Stop, Hide: Provide user controls to pause, stop and hide moving and auto-updating content.
2.3.1 – Three Flashes or Below Threshold: No content flashes more than three times per second.
2.4.1 – Bypass Blocks: Provide a way for users to skip repeated blocks of content.
2.4.2 – Page Titled: Use helpful and clear page titles.
2.4.3 – Focus Order: Components receive focus in a logical sequence.
2.4.4 – Link Purpose (In Context): Every link's purpose is clear from its text or context.
2.4.7 – Focus Visible: Keyboard focus is visible when used.
2.4.13 – Page Break Navigation: Provide a way to navigate between page break locators.
2.5.1 – Pointer Gestures: Multi-point and path-based gestures can be operated with a single pointer.

2.5.2 – Pointer Cancellation: Functions don't complete on the down-click of a pointer.
2.5.3 – Label in Name: Where a component has a text label, the name of the component also contains the text displayed.
2.5.4 – Motion Actuation: Functions operated by motion can also be operated through an interface and responding to motion can be disabled.
3.1.1 – Language of Page: Each webpage has a default human language assigned.
3.2.1 – On Focus: Elements do not change when they receive focus.
3.2.2 – On Input: Elements do not change when they receive input.
3.2.6 – Consistent Help: Help options are presented in the same order.
3.3.1 – Error Identification: Clearly identify input errors
3.3.2 – Labels or Instructions: Label elements and give instructions
4.1.1 – Parsing: No major code errors
4.1.2 – Name, Role, Value: Build all elements for accessibility

WCAG Checklist Level AA (Intermediate)
1.2.4 – Captions (Live): Add captions to live videos.
1.2.5 – Audio Description (Pre-recorded): Provide audio descriptions for pre-recorded videos.
1.3.4 – Orientation: Your website adapts to portrait and landscape views.
1.3.5 – Identify Input Purpose: The purpose of input fields must be programmatically determinable.
1.4.3 – Contrast (Minimum): Contrast ratio between text and background is at least 4.5:1.
1.4.4 – Resize Text: Text can be resized to 200% without loss of content or function.
1.4.5 – Images of Text: Don't use images of text.
1.4.10 – Reflow: Content retains meaning and function without scrolling in two dimensions.
1.4.11 – Non-Text Contrast: The contrast between user interface components, graphics and adjacent colours is at least 3:1.
1.4.12 – Text Spacing: Content and function retain meaning when users change elements of text spacing.
1.4.13 – Content on Hover or Focus: When hover or focus triggers content to appear, it is dismissible, hoverable and persistent.
2.4.5 – Multiple Ways: Offer at least two ways to find pages on your website.
2.4.6 – Headings and Labels: Headings and labels describe topic or purpose.
2.4.11 – Focus Appearance (Minimum): Focus indicators are clearly distinguishable when active.
2.5.7 – Dragging Movements: Functionality that uses dragging movements can be achieved with a single pointer without dragging.
2.5.8 – Target Size (Minimum): The target size for pointer inputs is at least 24 by 24 CSS pixels.
3.1.2 – Language of Parts: Each part of a webpage has a default human language assigned.
3.2.3 – Consistent Navigation: Position menus and standard controls consistently.
3.2.4 – Consistent Identification: Identify components with the same function consistently.
3.3.3 – Error Suggestion: Suggest fixes when users make errors
3.3.4 – Error Prevention (Legal, Financial, Data): Reduce the risk of input errors for sensitive data

As the main propose is not the complete a full web environment, but rather a breakdown of components, the WCAG rules will be applied to each component, so that will take a 100% accessible list of them.

In the current version of inSPICE only those components relevant to the GAM Game use case have been implemented.

6.1. GAM Game Usability Issues

The design of the GAM Game interface has been based on a set of requirements elaborated in cooperation with the Turin Institute for the Deaf, which represents the target group of the pilot (see **1.3.3 – Sensory Characteristics**).

In particular, as already discussed in the previous Section, these requirements concern:

- Text content: text should be limited to the bare minimum, and be replaced by visual interface elements, such as icons, and accompanied by video clips containing Italian Sign Language translation (**2.4.6 – Headings and Labels**). In addition, the text readability should be improved to the maximums using appropriate fonts, colour scheme, text size, etc., in accordance with WACG.
- Contrast: to help focalization, salient elements, such as artworks, should emerge clearly as important in the interface, using colour contrast to improve the visibility of artworks and text; the use of a dark background should be considered for indoor environments (see **WACG 1.4.1**).
- Complexity: the number of menu items, available options and items on display should be limited in order to reduce the cognitive load.

After testing the web app with the target group, a few redesign needs emerged. In general, deaf testers complained about the use of text in the interface. Remember, in fact, that for signers, written language is a second languages, with different letter set, terms, and syntax, and so can be challenging also for skilled reader, especially when reading occurs as part of a complex task.

In particular, the following requests were put forth:

- Text buttons: replacing the text in buttons (e.g. “Next”) with an equivalent icon (e.g., an arrow) wherever possible, so that deaf users are not disadvantaged (**1.3.3 – Sensory Characteristics**).
- Text duplication: as the current interface contains duplicated text in Response template (“It makes me think of…”), where the text is displayed twice (once for template description, once in the text box), users asked to replace the first text chunk with an icon.
- Search function: the search function was deemed impractical by the users, since it relies on text, so, they requested an alternative search mode, e.g., through QR codes or, even better, image-based search (**2.4.5 – Multiple Ways)**.
- Layout: some users had troubles in visualising the selected artwork in the Annotation interface after selection, since the box containing Response templates is the first element and the artwork image is only partly visible on some displays, to they asked to put the image first, followed by the Response templates (placed below the image, see **1.3.2 – Meaningful Sequence**).

7. Discussion and next steps

Regarding future work and limitations of the framework in its current state, the following points should be mentioned:

- There are currently no integrated mechanisms for aggregation and visualization of metrics or other representations of the data collected by the different modules and activities of the framework (activity definitions, records of user interaction with works, etc), beyond the representations that each use case might be able to produce with such information from the Linked Data Hub with external tools specific to each use case. As a future plan, we intend to provide tools integrated into the inSPICE framework itself to offer native solutions to the most common use cases in terms of visualization and aggregation / calculation of statistics on the recorded data. We plan to integrate into inSPICE the community visualization tools developed in the context of WP3 (see D3.5) on user data related to different interpretation activities.
- Another important consideration in the context of museums that include certain groups among their target audience (e.g. blind or deaf people) is to ensure that the components displayed in the application meet sufficient accessibility standards. In recent months we have been conducting an exhaustive preliminary analysis of accessibility issues and requirements specification in this area, and in the following iterations of the project we will proceed to progressively incorporate all accessibility requirements detected during this first analysis.
- As of today, all the activities and modules of the framework are hosted in a single centralized server instance as a first approximation, but in a practical context, it is very likely that each of the museums for which we would like to provide support for these citizen curation activities would like or need to have their own server with an associated instance of the Framework, with a particular configuration that suits their technical and contextual needs. In view of this, during the next year we will work on facilitating the instantiation and hosting of these inSPICE instances with their own configurations in each museum that may require these services.
- Lastly, it is worth mentioning that the modules and activities currently present in the framework are based on versions of activities and prototypes of web applications developed independently by the different SPICE use cases, but these independent applications have been evolving and growing over the last few months, so that the activities included in inSPICE to support these use cases are not necessarily synchronized with the current versions of these resources. Over the next few months, it will be necessary to review the developments in the proprietary applications of the other use cases to ensure that all of our templates can continue to support the use cases they reference.