X Components & the X Archetype

Workshop

Overlay vs X Components

Mantainability

Overlay was created as a stop-gap and with little time for planning how to scale the application.

Scalability

X Components had more time to plan and design the architecture, so we were able to have scalability in mind this time around.

Modular

We moved from config files to a more developer-friendly approach of modular components.

Lightweight

The main friction point with Overlay and previous frontend solutions was the bundle size and the initial loading times of the pages. Thanks to being modular, X Components allows customization of which components are used and added to the bundle.

Tree Shaking

Better tools allow X Components to remove from the bundle code that is not used in the setups, saving more loading time.

Faster development

With a new Design System, better testing and a more developer-friendly project, the setups for the clients are done much quicker.

Tech Stack

Development

- Typescript
- Vue 3
- Tailwind

Bundling

- Rollup *
- Vite
- Webpack

Testing

- Jest
- Cypress

The X Monorepo

The X Components live inside the X monorepo. All the packages interact with each other, so it's important to understand how they are structured.

Relevant Packages

Adapter	Library of utils to ease the communication with any API.
Platform Adapter	Pre-made adapter to communicate with the Empathy.co API.
Bus	Library that provides an event bus for event orchestration.
X Components	Library of components aimed to create search experiences.
X Tailwind	Tailwind plugin for of the X Design System.

* There are other packages that provide utilities, typings, etc., for example the **x-translations** or the **x-svg-converter**.

* Other relevant package is the **X Archetype** but it has its own repository outside of the monorepo.

The X Components

X Components structure

Components

Modular base components, intended to be used across all the app.

X Modules

Main features of the search experience, each with its own store module, event handling and components.

Directives

Custom Vue directive for infinite scroll.

Plugins

Functionalities that register the app, the store modules, the events bus, the components and the adapter to perform requests.

Services

State machine and directional focus navigation.

Store

A main X Store definition with sub-modules and utilities.

Composables

Code that uses reactivity, intended to be used in any component.

Demo views

Pages to explore different functionalities of the X Components.

Components

The general purpose components that don't fit in the x-modules directory.

Animations

Usually intended to be passed as a prop.

Base

Single base components to use along anothers.

Icons

Follow a specific structure to work with the Design System.

Modals

Modals-related components allow you to configure modals.

Results

Result-related components allow you to configure how product results are presented.

X Modules

Empathize

Extra params

Facets

History queries

Next queries

Popular searches

Query suggestions

Related tags

Scroll

Search

Search box

Tagging

Url

Recommendations

Identifier results

Queries preview

Semantic queries

Experience controls

Components
· Animations
· Base
· Icons
· Modals
· Results

Animations

Animations are components that just render their slot wrapped by a Vue `transition` and apply the animation styles to it. Some components accept Animation components as props.

```
<transition v-on="$listeners" name="x-animation-" v-bind="$attrs">
  <!-- @slot (Required) Transition content -->
  <slot />
</transition>
```

Usage with a dynamic component:

```
<component :is="animation">
  <div
    v-show="isOpen && hasContent">
  // ...
  </div>
</component>
```

Components
· Animations
· Base
· Icons
· Modals
· Results

Base

Minimal components aimed to be used as building blocks for more complex ones, although they can be imported and implemented by themselves too. They are used across all the modules and are not coupled to any of them.

`base-event-button.vue`

```
<button v-on="$listeners" @click="emitEvents">
  <!-- @slot (Required) Button content with a text, an icon or both -->
  <slot />
</button>
```

Example of usage:

`clear-filters.vue`

```
<BaseEventButton
  v-if="isVisible"
  class="x-clear-filters x-button"
  data-test="clear-filters"
  :disabled="!hasSelectedFilters"
  :events="events"
  :class="cssClasses"
>
  <slot :selectedFilters="selectedFilters">Clear Filters ({{ selectedFilters.length }})</slot>
</BaseEventButton>
```

Icons

Are set up as Vue functional components and follow a specific structure to work with the X Tailwind Design System. The package `x-svg-converter` has a utility to convert from SVG to a X Components-ready Vue component.

```
<template functional>
  <svg
    :class="['x-icon'].concat(data.staticClass, data.class)"
    viewBox="0 0 8 8"
    fill="none"
    xmlns="http://www.w3.org/2000/svg"
  >
    <path
      d="M1.5 4H6.24683"
      stroke="currentColor"
      stroke-width="0.4"
      stroke-linecap="round"
      stroke-linejoin="round"
    />
    <path
      d="M4 6.5L6.5 4L4 1.5"
      stroke="currentColor"
      stroke-width="0.4"
      stroke-linecap="round"
      stroke-linejoin="round"
    />
  </svg>
</template>
```

Components
· Animations
· Base
· Icons
· Modals
· Results

Modals

Modals are panels with overlays that will overlap the page when opened to show its content. They can listen to events for opening and closing from anywhere within the app.

`base-events-modal.vue`

```
<BaseModal
  @click:overlay="emitBodyClickEvent"
  @focusin:body="emitBodyClickEvent"
  :animation="animation"
  :open="isOpen"
  v-bind="$attrs"
  ref="baseModalEl"
>
  <slot />
</BaseModal>
```

Props for events to open and close:

```
eventsToOpenModal: {
  type: Array as PropType<XEvent[]>,
  default: (): XEvent[] => ['UserClickedOpenEven
},
eventsToCloseModal: {
  type: Array as PropType<XEvent[]>,
  default: (): XEvent[] => ['UserClickedCloseEve
}
```

Listening events to open:

```
props.eventsToOpenModal?.forEach(event =>
  $x.on(event, true).subscribe(({ metadata }) => {
    if (!isOpen.value) {
      openerElement.value = metadata.target;
      isOpen.value = true;
    }
  })
);
```

Components
- Animations
- Base
- Icons
- Modals
- Results

Results

There is no a pre-made result component in X-Components, but a couple of results-related base components that can be combined to configure what would be a result card. This allows more modularity and flexibility when approaching a setup.

Example of what a minimal result component could be:

```
<MainScrollItem>
  <BaseResultLink>
    <BaseResultImage />
  </BaseResultLink>
  <BaseResultLink>
    { result.description }
    <BaseResultCurrentPrice />
    <BaseResultPreviousPrice />
  </BaseResultLink>
</MainScrollItem>
```

X Modules

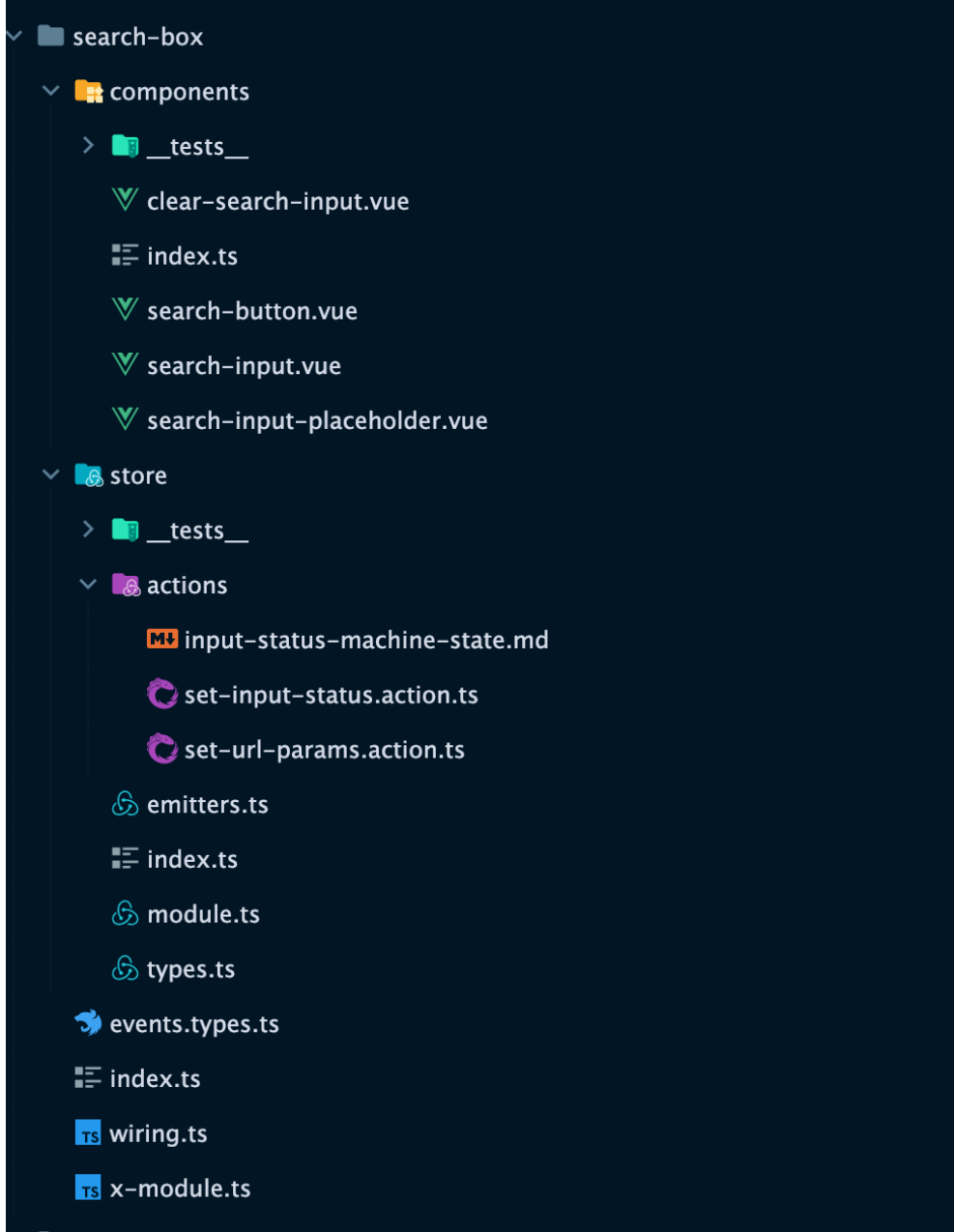
The X Modules are the core functionalities of the project. They are composed by:

Store
Components
Wiring

* The **wiring** is the event system that the X Modules use to communicate between each other & the app.

* They also communicate with the **endpoints** by calling an **endpoint adapter service**, and set the **store** values commiting data.

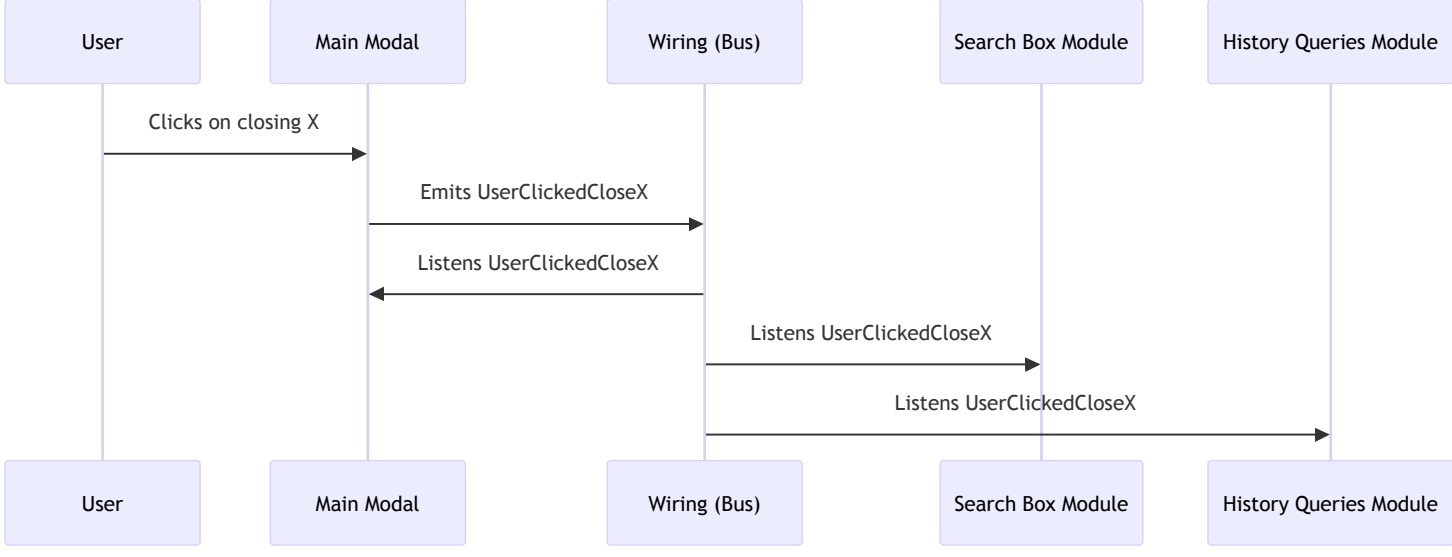
* An **X Module** is **registered** when it's going to be used, usually when the components that render its data are mounted. But this behavior can be customized if needed, as it is coupled to the application **init options**.



Wiring

· Each module has its own wiring to **listen to events** emitted across all the app and **trigger actions** from the module itself.
· Multiple modules can listen to the **same event** and react to it.
· The wiring of the modules is fully **customizable** from within a client setup, so you can add, remove or expand listeners and actions of any module.

Wiring flow example

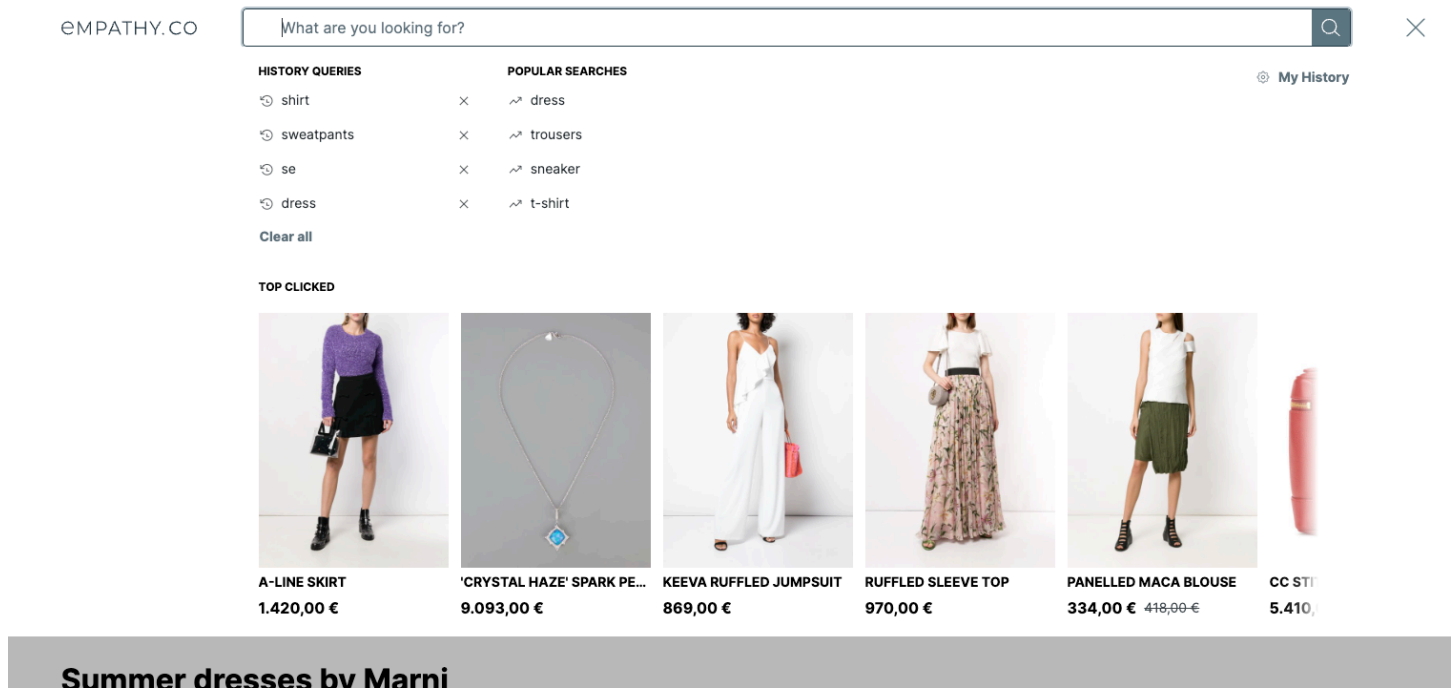


Modules
· Empathize
· Extra Params
· Facets
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Empathize

The Empathize is a modal-like container with an open/close state. Its content relies on other modules / components that are rendered in.

The Empathize Module is a core part of the search experience, tied to the action of searching. It gets opened when the **search box** has been **focused**. Provides an easy access to products and suggested queries while the user is **typing** or **about to search**.



Modules
· Empathize
· Extra Params
· Facets
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

The Empathize is coupled to the events that update its module's state:

`empathize.vue`

```
eventsToOpenEmpathize: {
  type: Array as PropType<XEvent[]>,
  default: () => ['UserFocusedSearchBox', 'UserIsTypingAQuery', 'UserClickedSearchBox']
},
eventsToCloseEmpathize: {
  type: Array as PropType<XEvent[]>,
  default: () => ['UserClosedEmpathize', 'UserSelectedASuggestion', 'UserPressedEnterKey', 'UserBlu
},

// Triggered on open or close
const changeOpen = useDebounce((newOpen: boolean) => {
  /**
   * Changes the state of Empathize `isOpen` and emits the `EmpathizeOpened` or
   * `EmpathizeClosed` events if the state really changes.
  */
}, 0);
```

Modules
· Empathize
· Extra Params
· Facets
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Empathize Usage Example:

```
<Empathize :animation="fade" :eventsToOpenEmpathize="['CustomOpenEmpathize']">
  <template>
    <h1>Suggestions</h1>
    <CustomSuggestions />
  </template>
</Empathize>
<BaseEventButton :events="['CustomOpenEmpathize']">
  Open Empathize
</BaseEventButton>
```

2. The content displayed inside the predictive layer.

3. The animation is imported from the x-component's library and then defined a constant to be passed as a prop.

Modules
~~Empathize~~
· Extra Params
· Facets
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Extra Params

This module contains a collection of params that will be sent in all the requests. They can be set via snippet config or programatically.

Sometimes requests to the endpoints require extra data that doesn't fit in the regular **requests parameters**. This module allows this data to be provided: the components in this module propagate the extra params to the rest of the application and stores.

A renderless component that sets the values of the Extra Params is the main component of the module:

`extra-params.vue`

```
values: {
  type: Object as PropType<Dictionary<unknown>>,
  required: true
}
// Events to update the store
$x.emit('ExtraParamsInitialized', { ...props.values });
$x.emit('ExtraParamsProvided', { ...props.values, ...params.value });
```

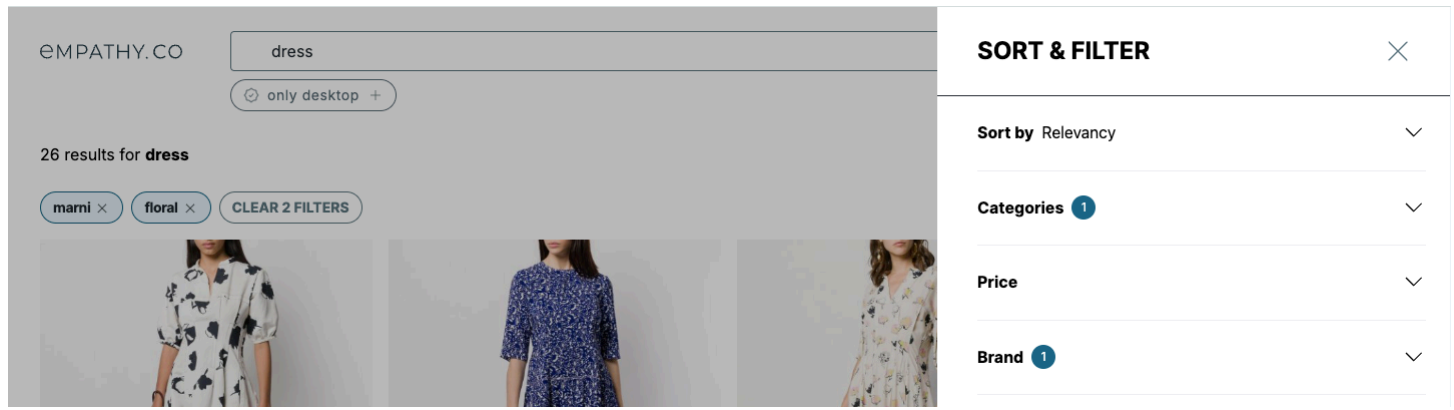
Modules
- ~~Empathize~~
- ~~Extra Params~~
- Facets
- History Queries
- Next Queries
- Popular Searches
- Query Suggestions
- Related Tags
- Scroll
- Search
- Search Box
- Tagging
- Url
- Recommendations
- Identifier Results
- Queries Preview
- Semantic Queries
- Experience Controls

Facets

Stores and controls everything related to filtering in the requests.

The wiring of the module handles events that change the facets's state directly or requires it to change (to render **filtering options**, manage **selected filters** state...).

There are different **types of filters**: simple, hierarchical, range... To help with their rendering specificities, there are pre-made list components (for sorted lists, for adding a search input, a collapse...).



Modules
~~Empathize~~
~~Extra Params~~
· Facets
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Modify filters behavior:

· Configure a filter to behave as a single selector:

By default, filters are expected to be multiselectable. This can be changed by implementing the `SingleSelectModifier` class before initializing the setup, this will create the filter instance with its coupled select & deselect logic.

The filter can be registered by `facetId` :

```
FilterEntityFactory.instance.registerModifierByFacetId('age_facet', SingleSelectModifier);
```

Or registered by `modelName` , thus then this would apply to all facets of this type:

```
FilterEntityFactory.instance.registerModifierByFilterModelName('HierarchicalFilter', SingleSelectMo
```

· Configure a filter to keep selected:

This can be done by implementing the `StickyModifier` class. This will enrich the filter with the `keepSticky` flag, if set to true, this will keep the filter selected.

```
FilterEntityFactory.instance.registerModifierByFacetId('brand_facet', StickyModifier);
```

Modules
~~Empathize~~
~~Extra Params~~
Facets
History Queries
Next Queries
Popular Searches
Query Suggestions
Related Tags
Scroll
Search
Search Box
Tagging
Url
Recommendations
Identifier Results
Queries Preview
Semantic Queries
Experience Controls

Facets Usage Example:

```
    facetsInStore = [{
    id: 'identifiableFacet',
    label: 'identifiableFacet',
    modelName: 'SimpleFacet',
    filters: [...]
  },
  {
    id: 'hierarchicalFacet',
    label: 'Hierarchical Facet',
    modelName: 'HierarchicalFacet',
    filters: [...]
  }];
```

```
<Facets>
    <template #identifiableFacet="{ facet, selectedFilters }">
      <FiltersList v-slot="{filter}">
        {{ filter.label }}
      </FiltersList>
    </template>
    <template #hierarchical-facet="{ facet, selectedFilters }">
      <FiltersList v-slot="{filter}">
        <HierarchicalFilter :filter="filter" />
      </FiltersList>
    </template>
</Facets>
```

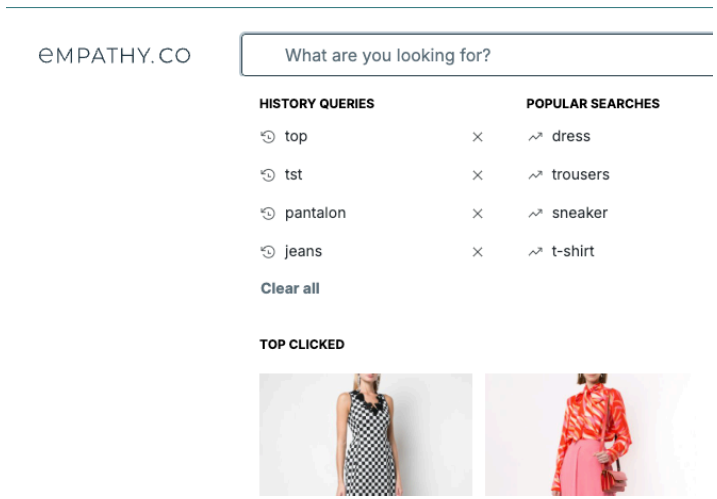
1. All the facets use a dynamic slot
2. The filters of the facet of type `HierarchicalFacet` uses the filter component of `hierarchical-filter`
3. The filters of the facet of id `identifiableFacet` show just the name of the filter

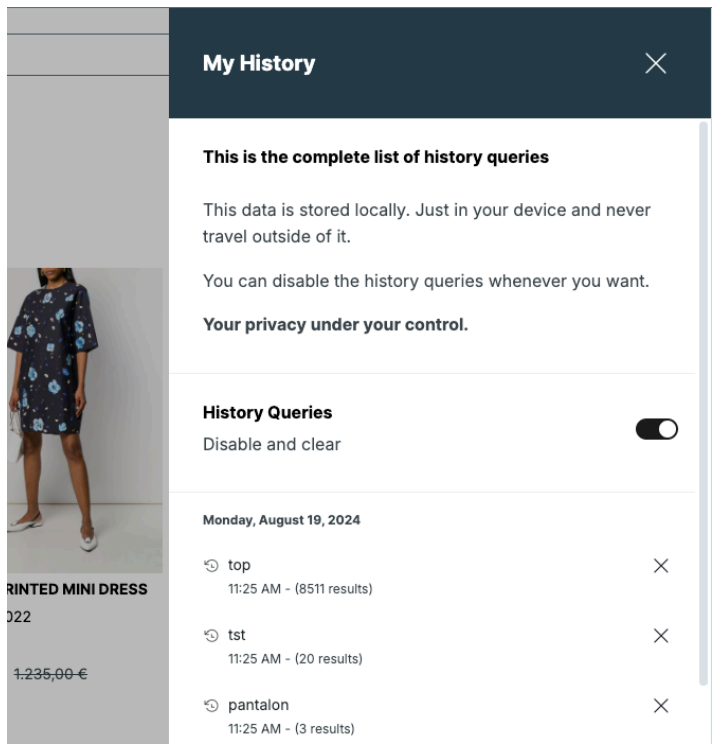
Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
History Queries
Next Queries
Popular Searches
Query Suggestions
Related Tags
Scroll
Search
Search Box
Tagging
Url
Recommendations
Identifier Results
Queries Preview
Semantic Queries
Experience Controls

History Queries

Module containing the queries already searched by the user.

Used to handle the logic for suggesting past queries, the `LocalStorage`, what queries are stored ot not, logic to remove them... Usually located inside the **Empathize** and into a dedicated **aside modal** that would enable users to handle the full list and even opt-out of storing their history queries.





Modules
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

The basic approach for the module is a list of the suggestions comming from the history queries store.

`history-queries.vue`

```
<BaseSuggestions v-bind="$attrs" :suggestions="historyQueries">
  <template #default="baseScope">
    <slot name="suggestion" v-bind="{ ...baseScope }">
      <HistoryQuery :suggestion="baseScope.suggestion">
        <template #default="historyQueryScope">
          <slot name="suggestion-content" v-bind="{ ...baseScope, ...historyQueryScope }" />
        </template>
        <template #remove-button-content="removeHistoryQueryScope">
          <slot name="suggestion-remove-content"
                v-bind="{ ...baseScope, ...removeHistoryQueryScope }"
          />
        </template>
      </HistoryQuery>
    </slot>
  </template>
</BaseSuggestions>
```

2. All the content can be customized. The slots have the suggestion info exposed.
3. A `HistoryQuery` is just a `base-suggestion` and a remove button to clear it from the store.
4. Slot for the history query (normally the text of the query being suggested).
5. Slot for the remove button.

Modules
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
· History Queries
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

History Queries Usage Example:

```
<HistoryQueries>
  <HistoryQuery
    class="x-suggestion-group-md"
    :suggestion="suggestion"
    suggestionClass="x-suggestion x-suggestion-md"
  >
    <template #default="{ query }">
      <Highlight :text="suggestion.query" :highlight="query" />
    </template>
    <template #remove-button-content>
      <Minus class="x-icon-md" />
    </template>
  </HistoryQuery>
</HistoryQueries>
```

2. The remove button slot is customized with a minus icon

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

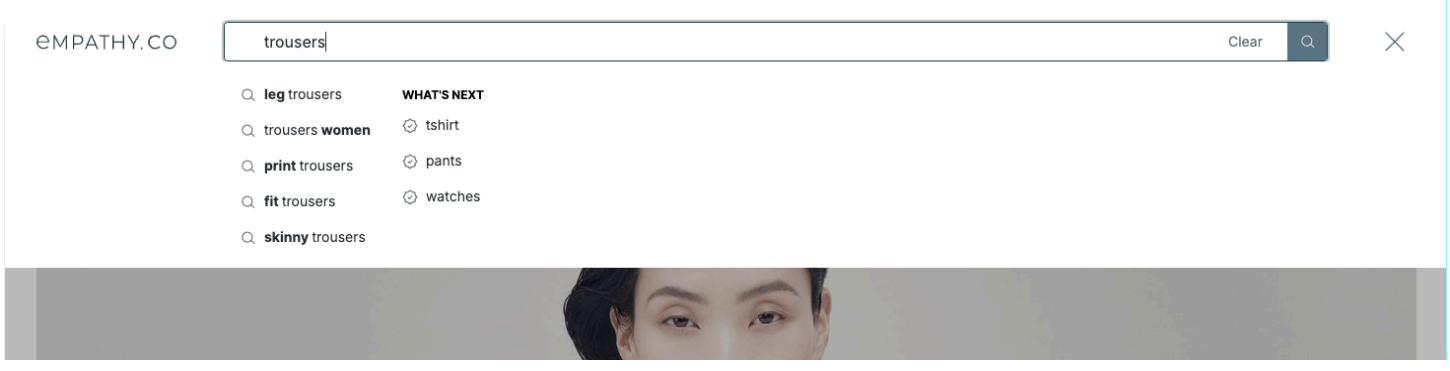
Next Queries

Module to handle terms usually searched by the users after the current query.

The next queries are requested to their own endpoint through the adapter, and the module stores the results. The `next-queries` and `next-query` components follow a similar approach of using `base-suggestions` for the listing and `base-suggestion` for the individual elements that we saw previously.

They can be shown as:

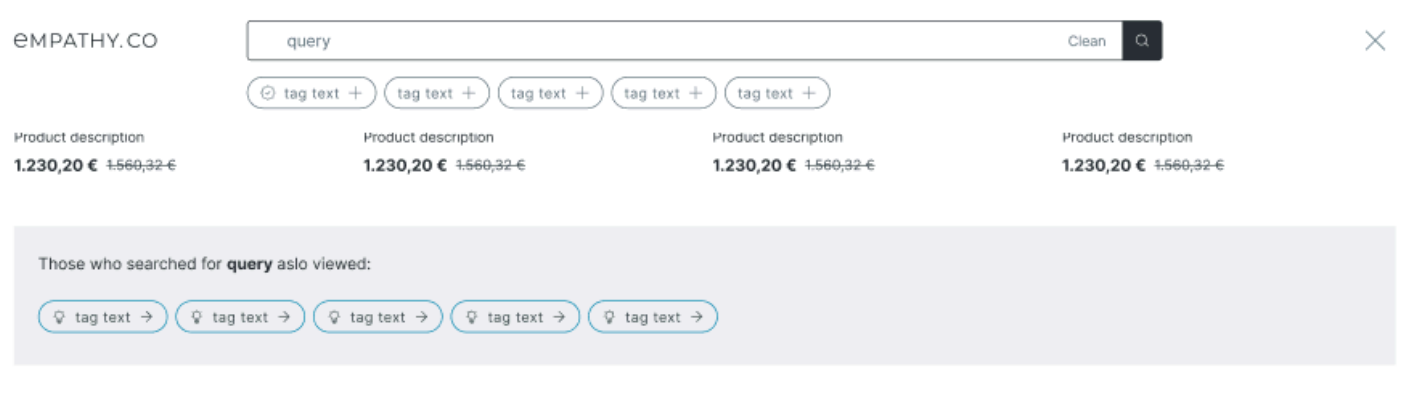
A list of queries, inside the predictive layer:



Modules

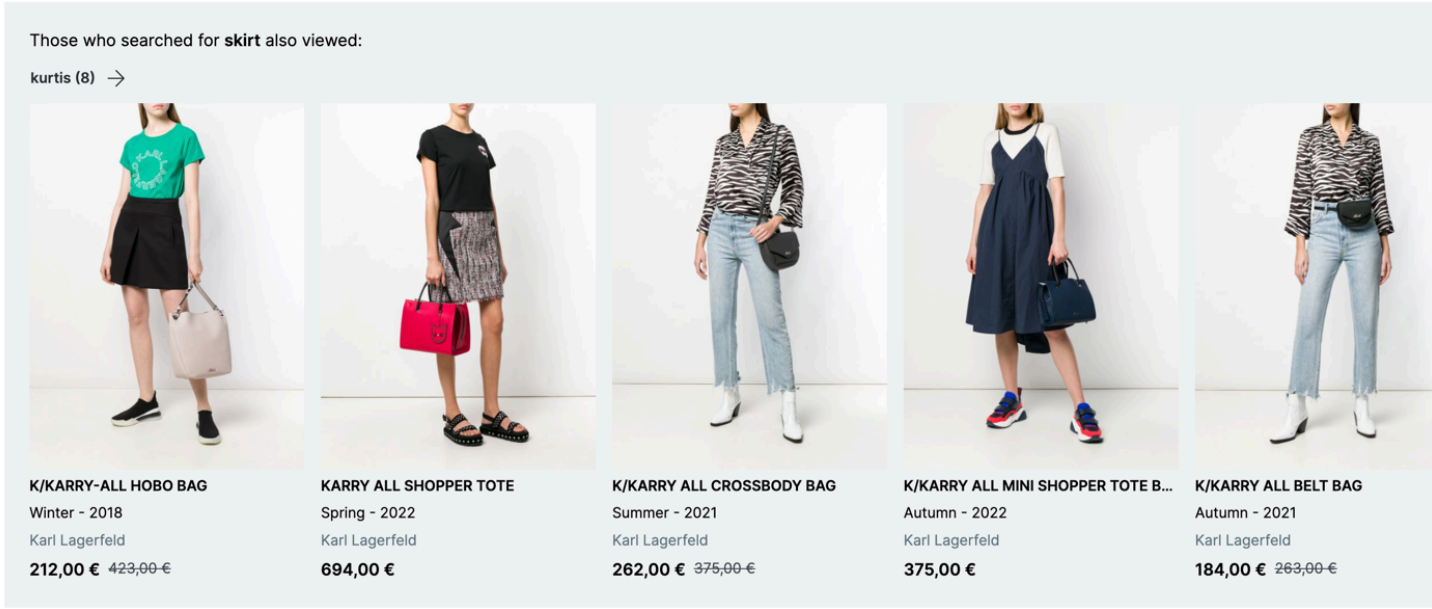
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
~~· History Queries~~
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

As a list of queries, scattered in the results grid:



Modules
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
~~· History Queries~~
· Next Queries
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Or even the module can also request and store a preview of the results of requesting that term, and display them as a sliding panel of results:



*There is already a dedicated component called `next-query-preview` in the module for this.

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

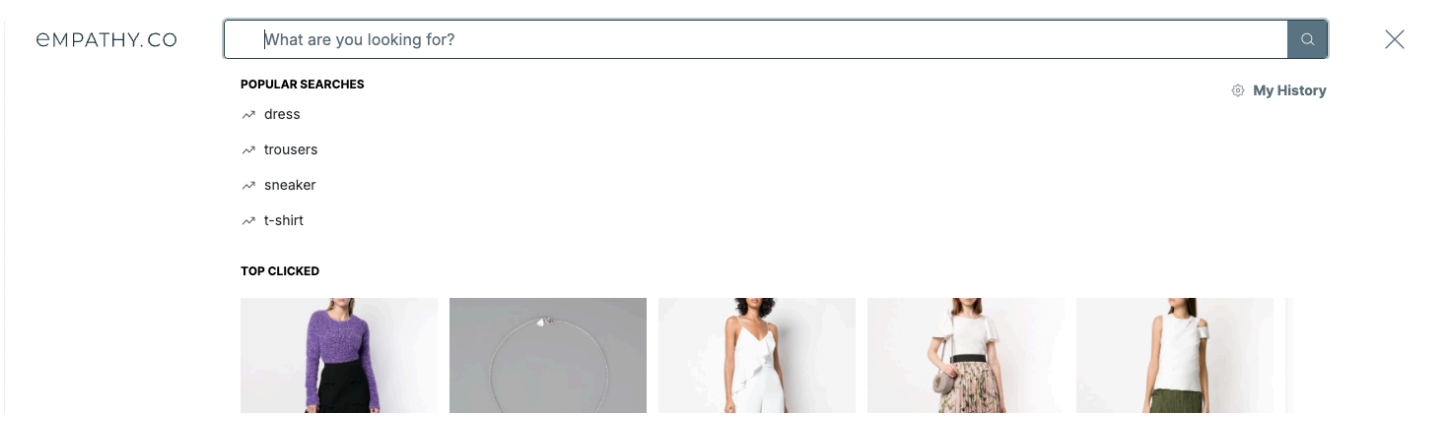
Popular Searches

Popular searches are the most searched terms, likely to be clicked by the user.

Popular searches are requested to their own endpoint, this can be done at any time as they don't depend on the current query. The wiring handles when to request them and the module stores them.

Once again, a pair of components, `popular-search` and `popular-searches`, based on `base-suggestions` and `base-suggestion`, are enough to render the information in this module.

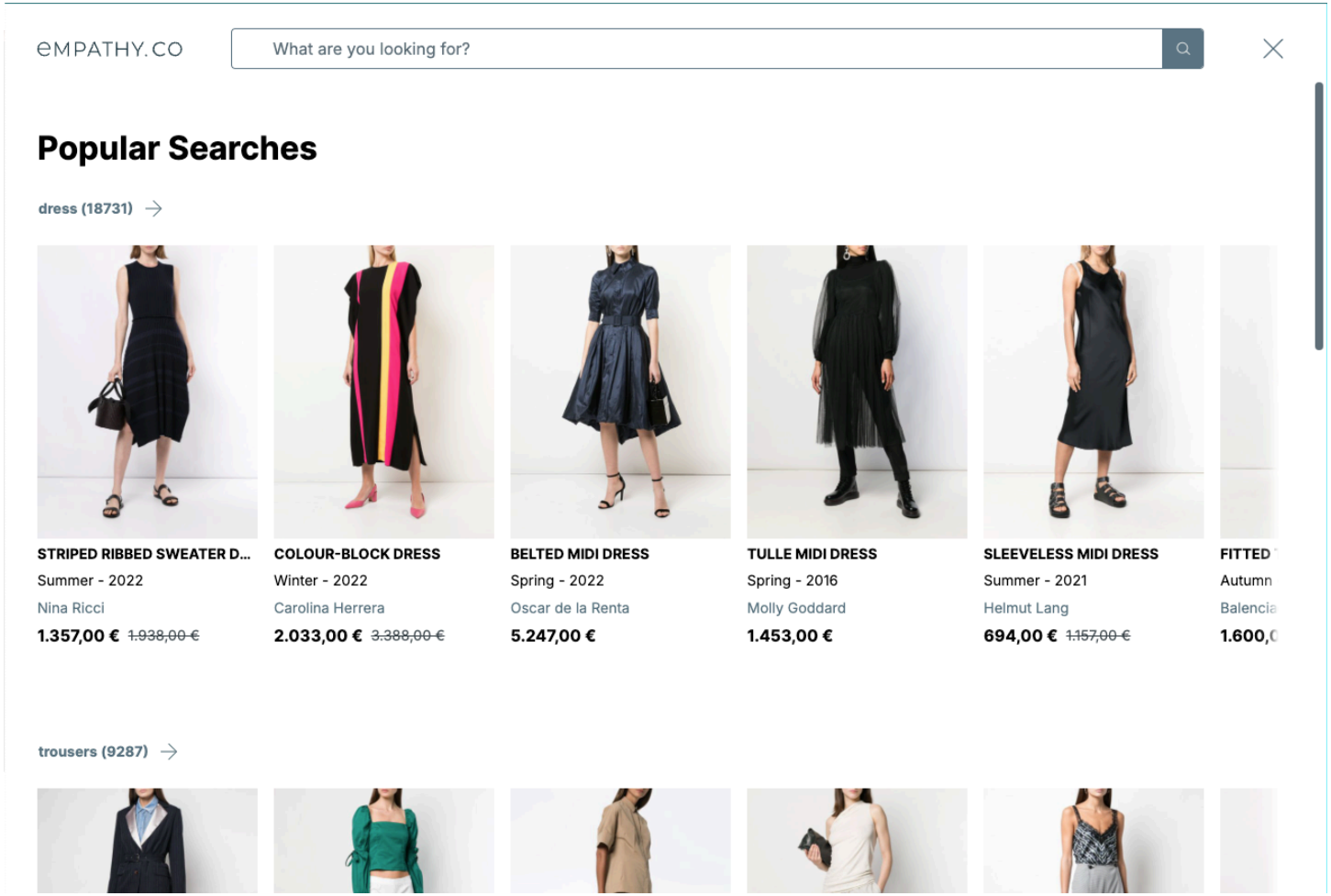
In the X-Archetype they are used to be rendered as a list of queries inside the predictive layer (empathize) when the search input is focused without a query, but they also can be displayed in the main grid zone as a pre-search strategy.



Modules
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
~~· History Queries~~
~~· Next Queries~~
· Popular Searches
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Popular searches shown as main page content before user starts searching.

Each popular searched query is used to be rendered with preview of its results inside a sliding panel.



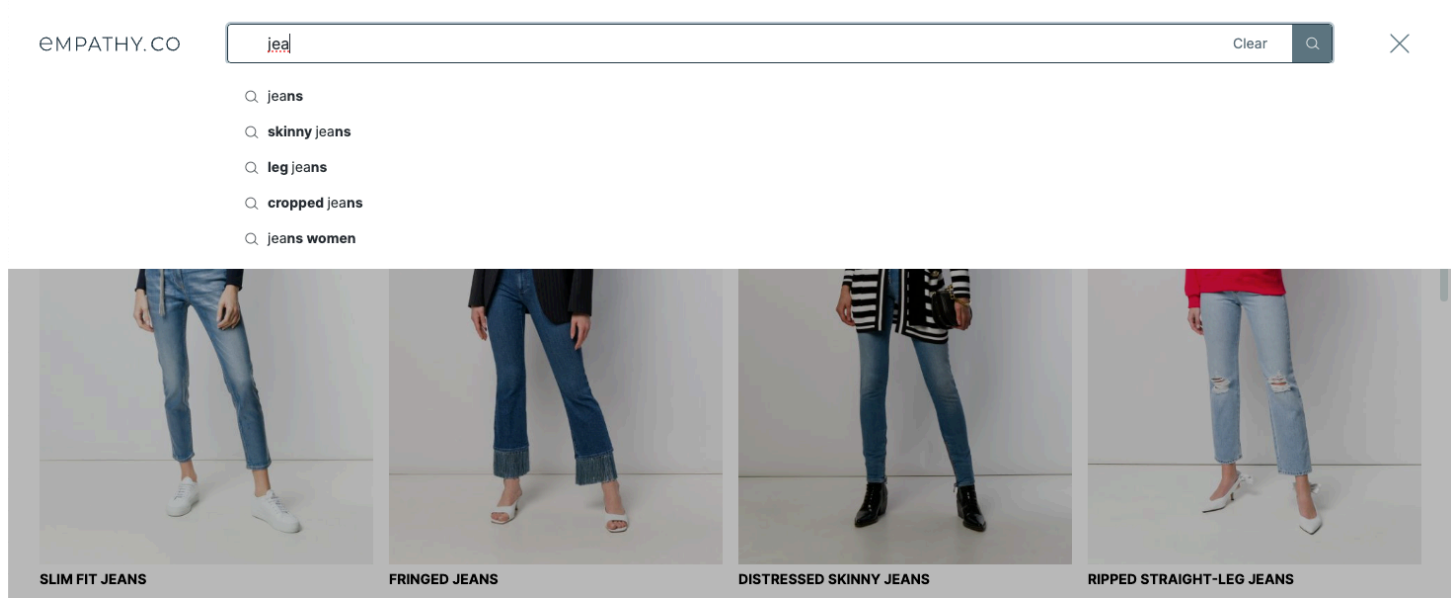
Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
· Query Suggestions
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Query Suggestions

It handles all the suggestions given while the user is typing in the search box.

Autocomplete is part of any worthwhile search experience. The module's wiring checks when the user is typing a query to request for suggestions and store them in the module. A dedicated endpoint receives the query and returns these query suggestions.

Two components using `base-suggestions` and `base-suggestion` handle the rendering, `query-suggestions` and `query-suggestion`.



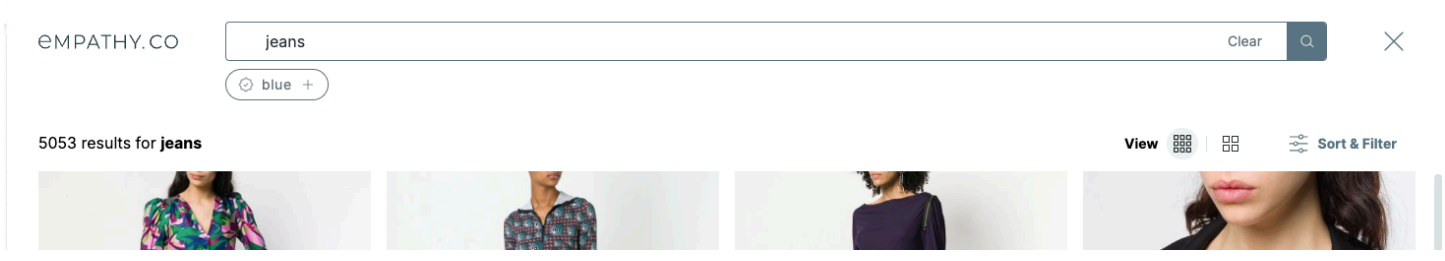
Modules
- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- Related Tags
- Scroll
- Search
- Search Box
- Tagging
- Url
- Recommendations
- Identifier Results
- Queries Preview
- Semantic Queries
- Experience Controls

Related Tags

Additional keywords to complement the current search.

The Related Tags Module depends entirely of the query, so when a query is accepted the request for the tags will be made and the response stored. The module needs to store the related tags, the selected suggestions and make request calls through its specific adapter.

Related Tags are treated as buttons instead of suggestions, so the components for rendering are a regular list of buttons, but the customization options with slots are very similar.



Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
· Related Tags
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Related Tags Usage Example:

```
<SlidingPanel v-if="$x.relatedTags.length">
  <RelatedTags :highlightCurated="true">
    <template #related-tag-content="{ relatedTag, isSelected, shouldHighlightCurated }">
      <CuratedCheckIcon v-if="shouldHighlightCurated" />
      {{ relatedTag.tag }}
      <CrossTinyIcon v-if="isSelected" />
      <PlusIcon v-else />
    </template>
  </RelatedTags>
</SlidingPanel>
```

2. Slot for customizing related tag item's content.
3. If the related tag is curated it will display a special icon on the left.
4. Selected/Unselected status also can be rendered differently.

>

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Scroll

Handles the state of the scroll, so stores all related data accordingly.

The Scroll Module tracks the id of the **current element** in the scroll's view, and so it can rely the information about its **position** relative to the start or the end of the **scrolling space**, its **direction**, and **restore** the position if necessary.

The `scroll` component is the backbone of the module. It uses the `base-scroll` component and it's the one which communicates with the module through **events**. To use it is enough to **wrap** the components that you want to apply the scroll to.

`scroll.vue`

```
<BaseScroll
  @scroll="emitScroll"
  @scroll:direction-change="emitScrollDirectionChange"
  @scroll:at-start="emitScrollAtStart"
  @scroll:almost-at-end="emitScrollAlmostAtEnd"
  @scroll:at-end="emitScrollAtEnd"
  v-on="$listeners"
  :id="id"
  v-bind="$attrs">
  <slot />
</BaseScroll>
```

2. Emits every scroll event so the module picks up the information and the `id` for the dictionary in the module slot.

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
· Scroll
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Aside from some utility components in the module, the other important ones are the `main-scroll` and the `main-scroll-item`. The `main-scroll` is an extended `scroll` component, it **tracks the current component** in view, so the scroll position can be restored if the page is loaded from the URL. The `main-scroll-item` adds the **ids and the tracking communication** to the components they wrap.

`main-scroll-item.vue`

```
item: { type: Object as PropType<Identifiable>, required: true },
tag: { type: [String, Object] as PropType<string | typeof Vue>, d

const firstVisibleItemObserver = inject<Ref<ScrollVisibilityObser
  ScrollObserverKey as string,
  null
);
// Mounted does not guarantee that child components are mounted t
onMounted(() => {
  nextTick(() => {
    if (firstVisibleItemObserver) {
      watch(firstVisibleItemObserver, observeItem, { immediate: t
    }
  });
});
```

2. The observable coming from an scroll element. (`main-scroll`).

3. Sets the observable that is tracking as the visible element in the scroll.

Usage:

```
<MainScroll>
  <Scroll id="main-scroll">
    <ResultsList>
      <MainScrollItem
        :item="result"
        tag="article"
      >
        <Result :item="result"
      </MainScrollItem>
    </ResultsList>
  </Scroll>
</MainScroll>
```

Modules
~~· Empathize~~
~~· Extra Params~~
~~· Facets~~
~~· History Queries~~
~~· Next Queries~~
~~· Popular Searches~~
~~· Query Suggestions~~
~~· Related Tags~~
~~· Scroll~~
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Search

The search module contains the results for the current query, as well as several components used to show the results, modify the sort, render the spellchecked query, etc.

The wiring makes the search requests through the search endpoint adapter and stores the response in the module:

· The different result object types: **regular results, promoted, banners, partials**, etc.
· The **number of results** for a query.
· The available **facets** related to that searched query.
· And also keeps track of the **pagination**.

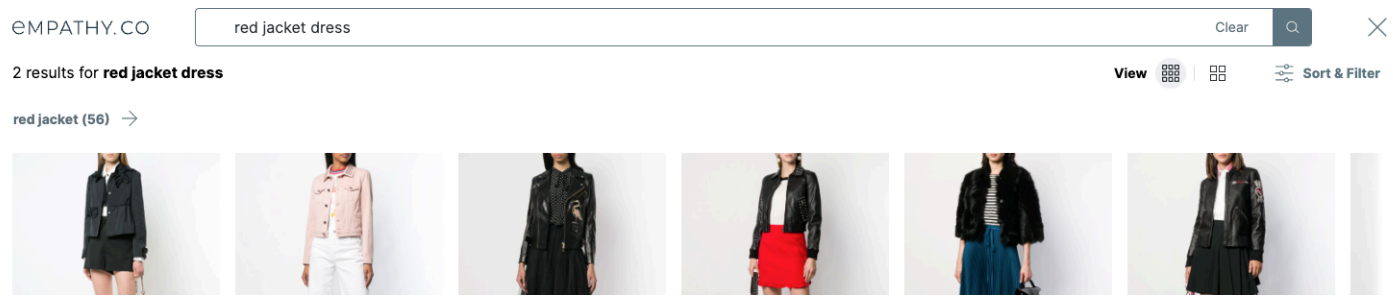
*Some of this data will be consumed in other parts of the setup and other will be handled by the search module itself.

· **Result**. A product with all the info.

· **Banner**. A full grid width ad.

· **Promoted**. Inside the grid, as a sponsored item within the normal results and in a pre-configured position.

· **Partial**. When the query has few results and can be divided in other more relevant queries. Example: with the query `red jacket dress`, we obtain partials as `red jacket`.



Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
· Search
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

There are lists and/or individual components for almost each type of result, and they are included in a single grid that contains all these elements but each with its own style and rendering strategy.

To achieve this, the `result-list` can wrap any other list in the module. It will render the results plus the elements of that list, and the next list that gets wrapped will do the same.

Chained lists work because they are injected with the elements to be rendered from the previous list.

Lastly, the way each type of element will render is determined by the named slots in the last chained list.

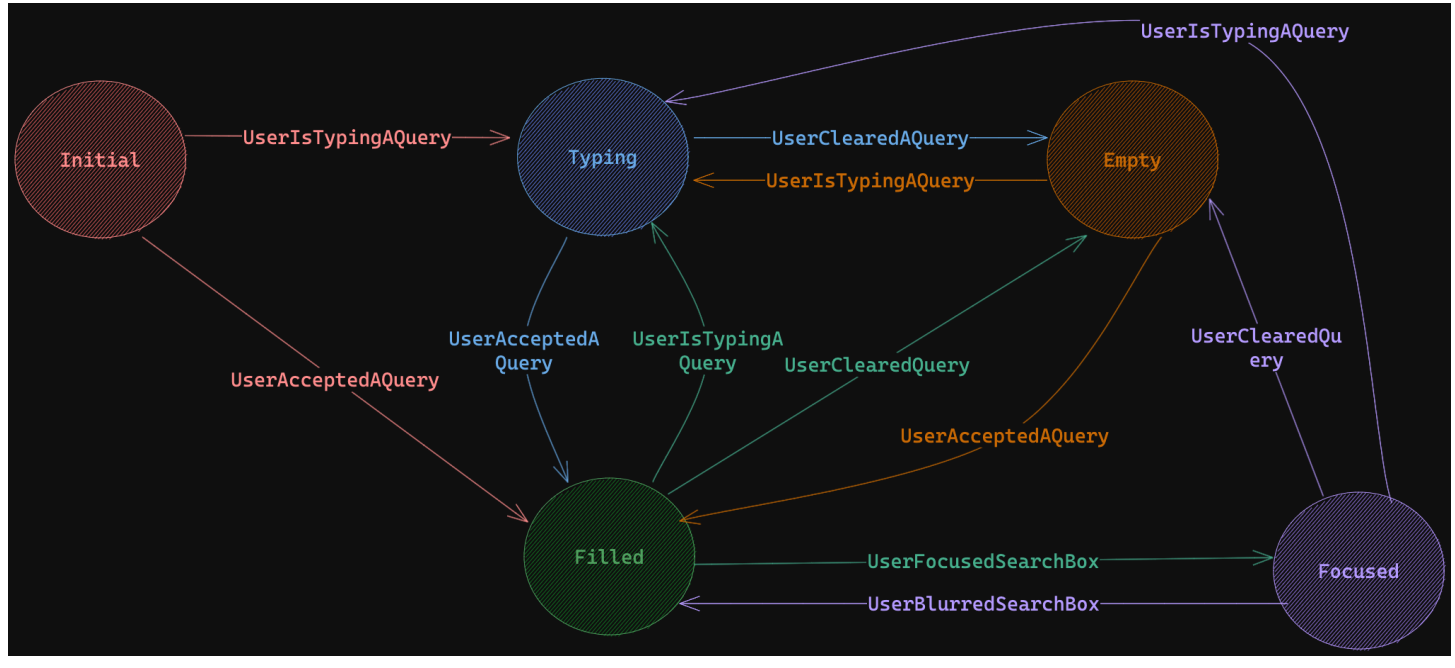
```
<ResultsList>
  <PromotedsList>
    <BannersList>
      <template #result="{ item }">
        <Result :item="item" />
      </template>
      <template #banner="{ item }">
        <Banner :item="item" />
      </template>
      <template #promoted="{ item }">
        <Promoted :item="item" />
      </template>
    </BannersList>
  </PromotedsList>
</ResultsList>
```

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Search Box

This module contains all the logic related to the search input. It is the source of truth for the `query` and hence most of the other modules.

· Mirrors the content of the search input to the store and emits that the query has changed.
· Emits events for most of the interactions that can be done with an input.
· Keeps track of the actions made by the user with a **state machine**.



Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
· Search Box
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

`search-box.vue`

```
<input
  ref="input"
  @mouseenter="emitUserHoveredInSearchBox"
  @mouseleave="emitUserHoveredOutSearchBox"
  @blur="emitUserBlurredSearchBox"
  @click="emitUserClickedSearchBox"
  @focus="emitUserFocusedSearchBox"
  @input="emitUserIsTypingAQueryEvents"
  @keydown.enter="emitUserPressedEnterKey"
  @keydown.up.down.prevent="emitUserPressedArrowKey"
  @beforeinput="preventSpecialKey"
  v-on="$listeners"
  :maxlength="maxLength"
  :value="query"
  autocomplete="off"
  enterkeyhint="search"
  inputmode="search"
  type="search"
/>
```

2. Focus related events (normally linked to showing the empathize).
3. Typing related events (showing suggestions).
4. Passed on events.
5. The value mirrored in the store.
6. Input configurations.

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
~~Search Box~~
· Tagging
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Tagging

This module is in charge of sending the requests to track some user actions, but also makes sure that no tracking is done before the user actually accepts it.

A **renderless component** enables the tracking.

Specific actions then make a tracking request, for example, when a result is clicked. The tagging request sends properties like **location** and **feature**, which can vary depending on where the result is located and why it is displayed.

Typically, this information is **provided** and **injected** to pass data to the events that the module wiring uses to make the requests.

Usage Example:

```
<Tagging :consent="true" />
<LocationProvider location="predictive_layer">
  <Result
    :item="result"
  />
</LocationProvider>
```

* The `constent` prop is set to true to enable tracking. This is usually set in the app init object.

* The result will send a tagging request with a location equal to `'predictive_layer'`.

Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
~~Search Box~~
~~Tagging~~
· Url
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Url

Manages the browser URL parameters to preserve them through reloads and history navigations.

The module is based on a **renderless component** called `url-handler` that parses the URL parameters and emits the events to update the stores of the modules. The component handles various logics and edge cases, like navigating with the browser back button or coming from Product Page.

· Stores url parameters that are relevant in case of a url reload (sorting, facets, query, etc.).
· Listens to events that impact on that relevant parameters for the url and updates the URL if needed.
· Restores the X Components store status when loading from URL.

* There's no special usage for this module, just placing the `url-handler` component in the root component of the setup is enough.

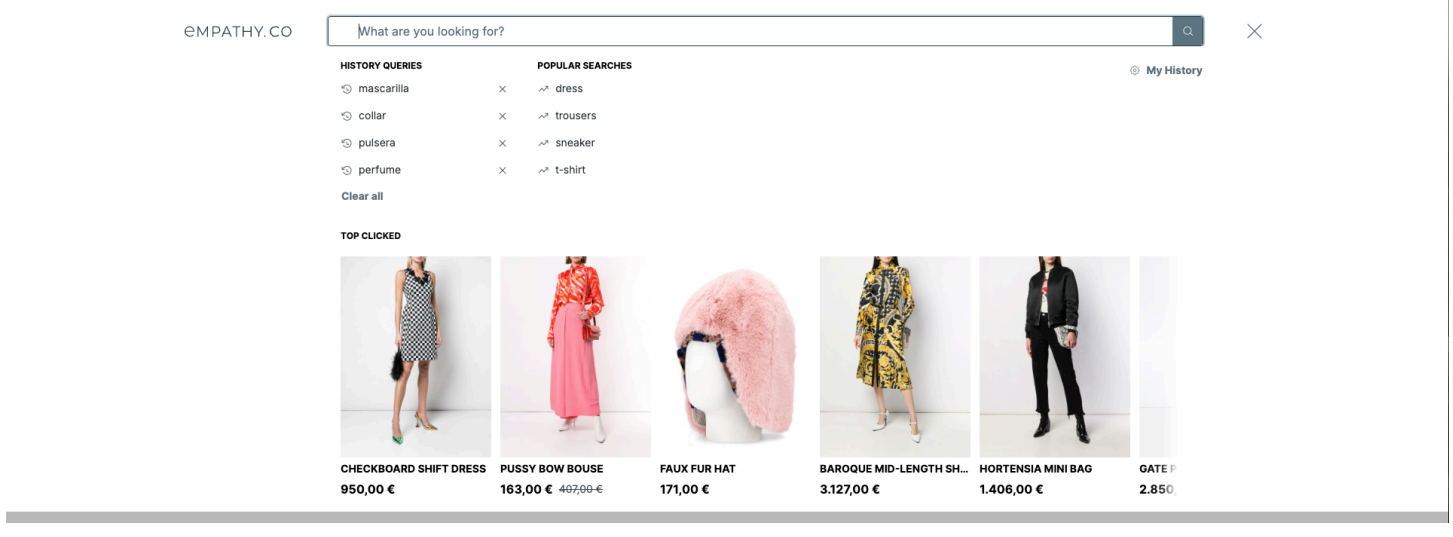
Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
~~Search Box~~
~~Tagging~~
~~Url~~
· Recommendations
· Identifier Results
· Queries Preview
· Semantic Queries
· Experience Controls

Recommendations

The Top Clicked products, likely to be clicked by the user and not bound to a specific part of the website.

The dedicated endpoint returns the recommendations and result objects. The idea is requesting the recommendation right away because the request is not expected to change. The module has a highly customizable component to render the list of recommendations as results.

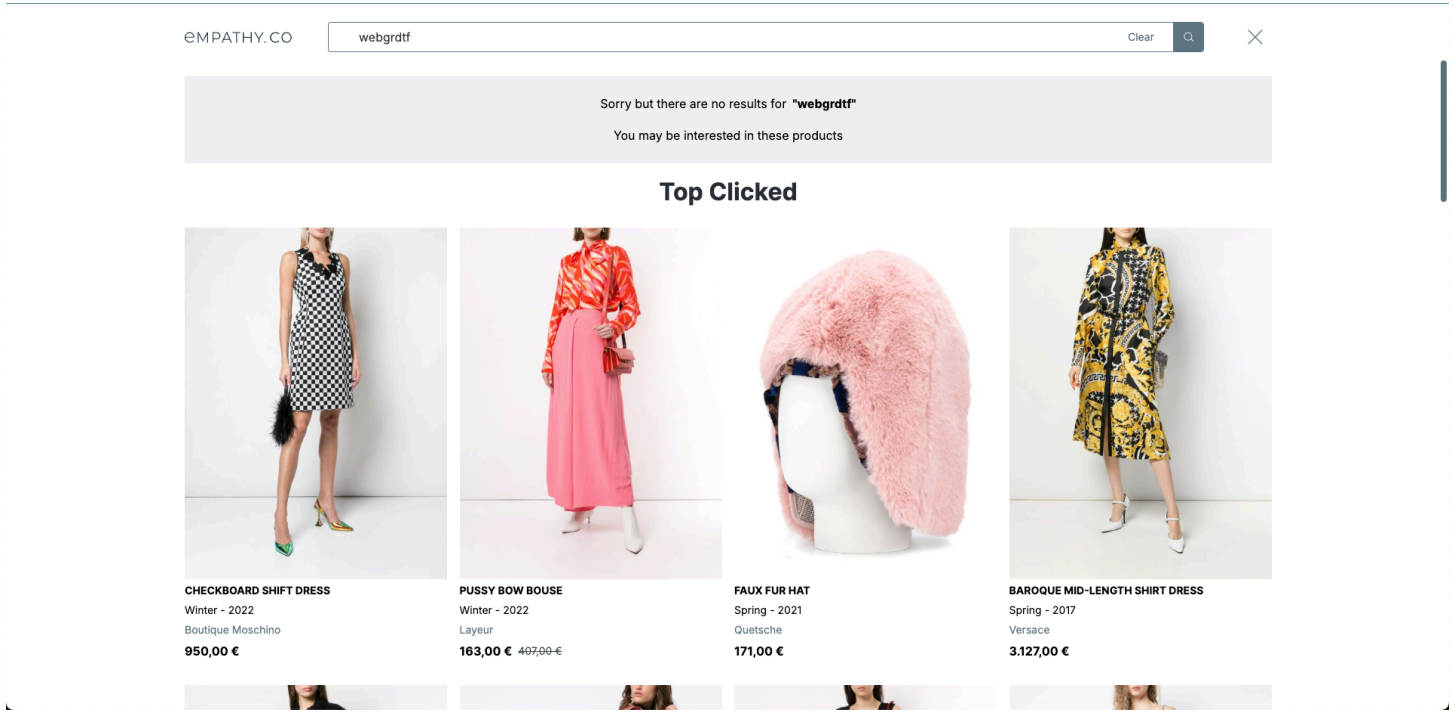
In the X-Archetype they are used to be rendered inside the predictive layer (empathize) when the search input is focused without a query, or when no-results are found, as fallback content.



Modules

- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- ~~Related Tags~~
- ~~Scroll~~
- ~~Search~~
- ~~Search Box~~
- ~~Tagging~~
- ~~Url~~
- Recommendations
- Identifier Results
- Queries Preview
- Semantic Queries
- Experience Controls

Top Clicked recommendations in a no-results scenario.



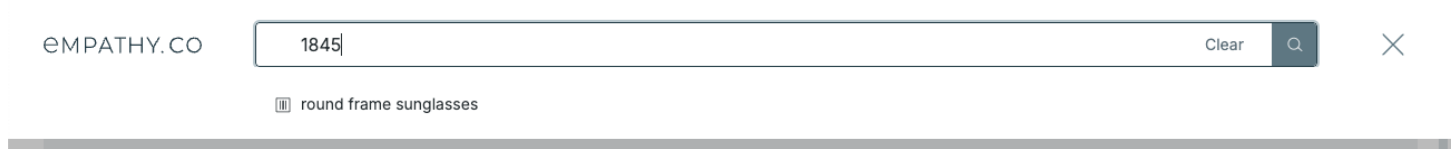
Modules
- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- ~~Related Tags~~
- ~~Scroll~~
- ~~Search~~
- ~~Search Box~~
- ~~Tagging~~
- ~~Url~~
- ~~Recommendations~~
- Identifier Results
- Queries Preview
- Semantic Queries
- Experience Controls

Identifier Results

Module containing the results of a search by product id.

Beside other common module configurations (e.g `maxItemsToRequest`), it requires the **regex expression** that provides the characters' match and activates the request to the **identifier search endpoint** through the adapter.

The response of that request is mapped and saved in the identifier results state module as `Results[]`. The components in the module are: the `IdentifierResult` item, with the identifier query and its matching part; and the `IdentifierResults` list of items that are rendered inside the predictive layer.

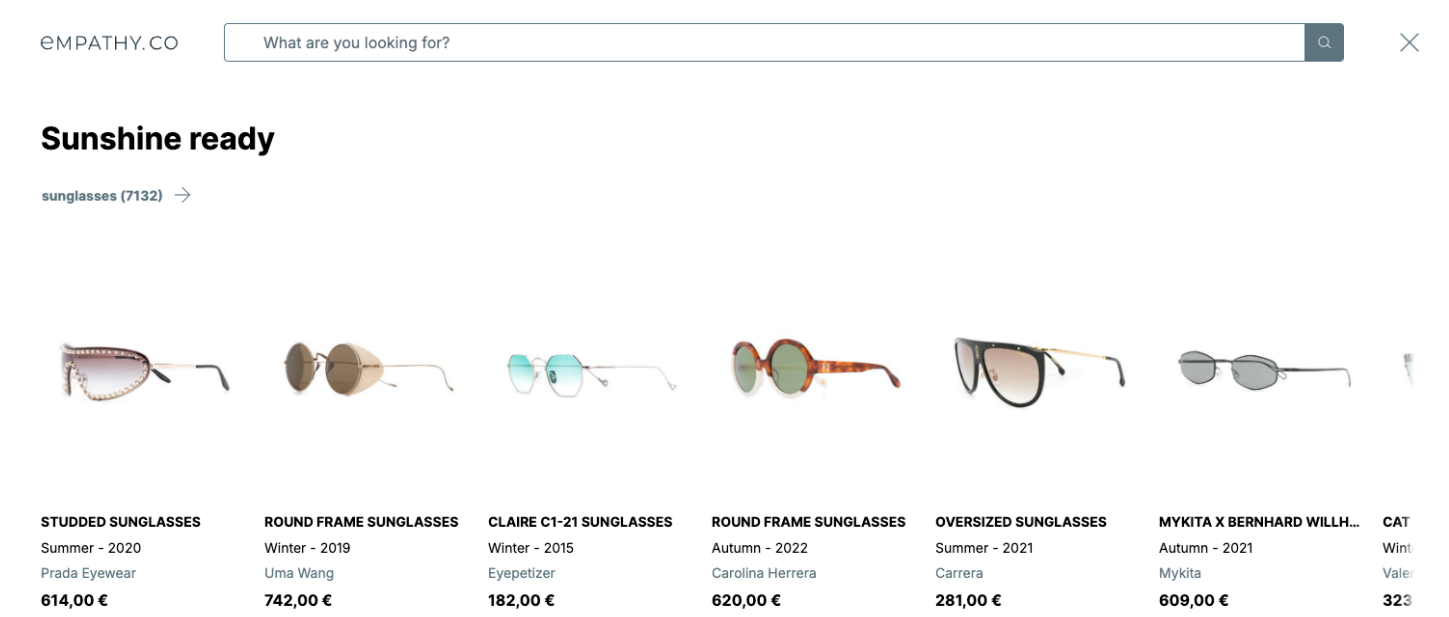


Modules
- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- ~~Related Tags~~
- ~~Scroll~~
- ~~Search~~
- ~~Search Box~~
- ~~Tagging~~
- ~~Url~~
- ~~Recommendations~~
- ~~Identifier Results~~
- Queries Preview
- Semantic Queries
- Experience Controls

Queries Preview

Retrieves a preview of the results of a given query. It is a pre-search strategy to display content before the user searches.

The queries to be previewed are defined via **Snippet Config** by passing an array of objects containing a **query**. The module requests those queries from the **search endpoint** in sequential order and catches them to avoid unnecessary requests if they have already been saved. This also allows us to save multiple queries and reuse components without colliding.



Modules
~~Empathize~~
~~Extra Params~~
~~Facets~~
~~History Queries~~
~~Next Queries~~
~~Popular Searches~~
~~Query Suggestions~~
~~Related Tags~~
~~Scroll~~
~~Search~~
~~Search Box~~
~~Tagging~~
~~Url~~
~~Recommendations~~
~~Identifier Results~~
· Queries Preview
· Semantic Queries
· Experience Controls

Queries Preview Usage Example:

```
<QueryPreviewList
  v-if="queriesPreviewInfo"
  :debounceTimeMs="250"
  :queriesPreviewInfo="queriesPreviewInfo"
  :persistInCache="true"
  #default="{ queryPreviewInfo, totalResults, results }"
>
  <QueryPreviewButton
    :queryPreviewInfo="queryPreviewInfo"
  >
    {{ queryPreviewInfo.query }}
    ({{ totalResults }})
    <ArrowRightIcon class="x-icon-lg" />
  </QueryPreviewButton>
  <SlidingPanel>
    <Result
      v-for="result in results"
      :key="result.id"
      :result="result"
    />
  </SlidingPanel>
</QueryPreviewList>
```

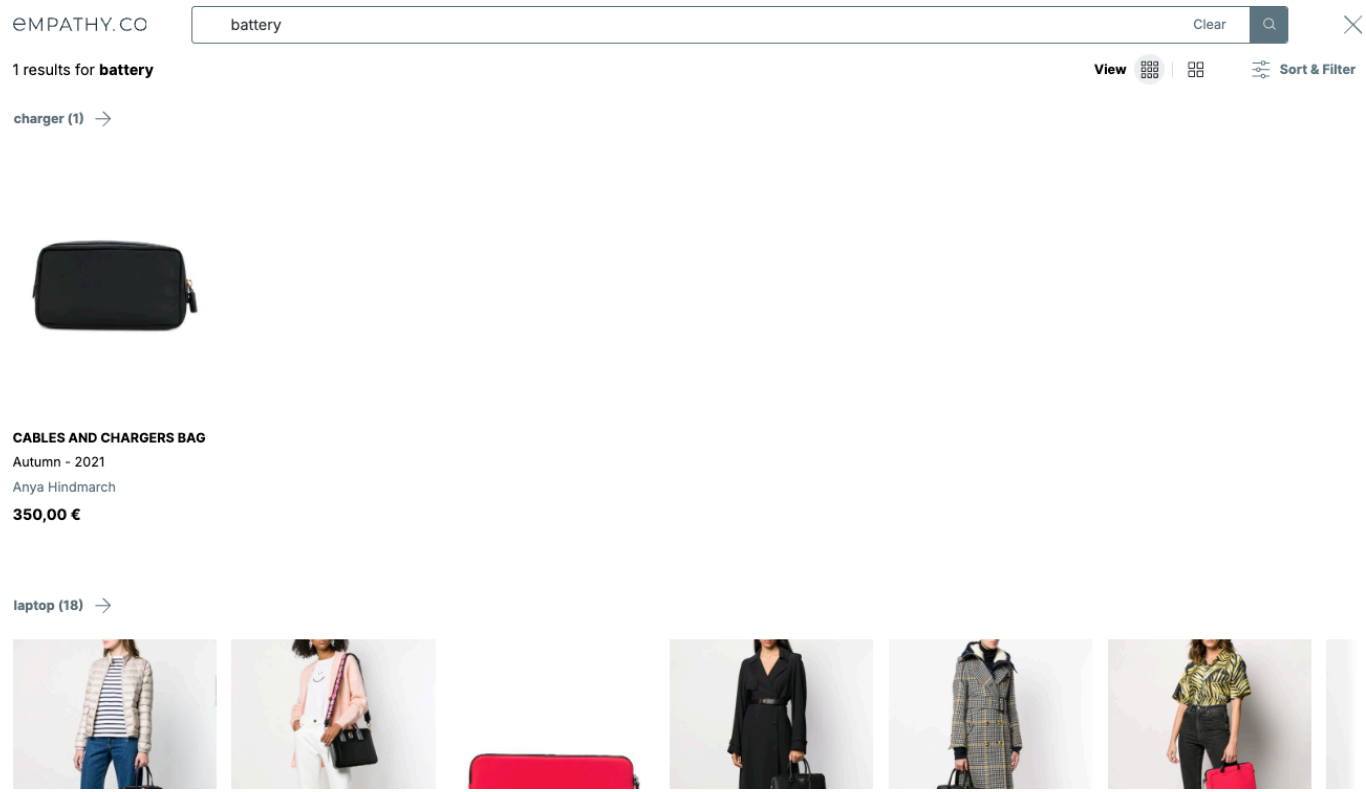
2. If the `QueryPreviewButton` is clicked, it will emit the event `UserAcceptedAQueryPreview` and a search request will be performed.

Modules
- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- ~~Related Tags~~
- ~~Scroll~~
- ~~Search~~
- ~~Search Box~~
- ~~Tagging~~
- ~~Url~~
- ~~Recommendations~~
- ~~Identifier Results~~
- ~~Queries Preview~~
- Semantic Queries
- Experience Controls

Semantic Queries

This module is in charge of requesting and rendering meaningful search results.

Usually rendered when there are few or no results, they are another kind of suggestion with its endpoint adapter, which has a special configuration option, the `threshold`, to filter out terms that are not relevant enough.



Modules
- ~~Empathize~~
- ~~Extra Params~~
- ~~Facets~~
- ~~History Queries~~
- ~~Next Queries~~
- ~~Popular Searches~~
- ~~Query Suggestions~~
- ~~Related Tags~~
- ~~Scroll~~
- ~~Search~~
- ~~Search Box~~
- ~~Tagging~~
- ~~Url~~
- ~~Recommendations~~
- ~~Identifier Results~~
- ~~Queries Preview~~
- ~~Semantic Queries~~
- Experience Controls

Experience Controls

This module aims to allow the implementation of some configuration settings outside the application.

The Experience Controls Module is coupled to the Empathy's Instance Management Console. IMC enables users to manage Vectors/Semantics search and connect it to X-Components to change their behavior. These settings can be activated or deactivated to suit temporary business needs without implementing this logic in the code itself.

The module has a dedicated component that receives a list of events and emits them.

The X Adapter & The X Platform Adapter

X Adapter

The adapter is the middleman between the backend endpoints and the X Components.

The X Components modules call a specific **adapter** to request the information and map the response. The modules will request to the endpoint with a format and expect the response with a specific structure, the adapter has the task of convert the request to something that the endpoint understands and to map the response into something the module expects.

Roughly speaking, each adapter consists of 3 parts that would need to be configured. The **endpoint** that will be called, the **request mapper** and the **response mapper**

The endpoint is very straightforward, it's just the URL that will be called.

The request and response mappers are functions called before the request and after the response respectively. They are the ones that will adapt the formats to the needs of each part of the requests, and they do so with **Schemas**.

Schemas

A Schema is a dictionary that indicates what values from an object should be mapped to what field from another. They are used by the mappers to know how to map requests and responses.

Imagine you have a search requests where the endpoint expects the query value to be `keyword` but the X Components are sending it as `query`. You can create a schema for the request mapper that maps the requests as the endpoint expects it:

```
const searchRequestSchema = {
  keyword: 'query',
}
```

In the same way you can use a schema to map the response from the endpoint to the format that the X Components expect:

```
const searchResponseSchema = {
  images: ({imageStrings}) => imageStrings.split(','),
}
```

X Platform Adapter

The X Platform Adapter is a pre-built X Adapter with all the adapter endpoints configured to work with Empathy's backend.

```
export const platformAdapter: PlatformAdapter = {
  search: searchEndpointAdapter,
  popularSearches: popularSearchesEndpointAdapter,
  recommendations: recommendationsEndpointAdapter,
  nextQueries: nextQueriesEndpointAdapter,
  querySuggestions: querySuggestionsEndpointAdapter,
  relatedTags: relatedTagsEndpointAdapter,
  identifierResults: identifierResultsEndpointAdapter,
  tagging: taggingEndpointAdapter
};
```

The X Adapter allows overriding of the mappers and schemas, so the idea of the X Platform Adapter is to have something working with an existing endpoint out of the box and override the configurations as needed.

The alternative to using the X Platform Adapter is to create a new X Adapter form scratch.

Customizing a Setup

X-Archetype

A pre-built setup using X Components, X Tailwindcss, and X Platform Adapter to work as a starting point.

Init options

- Snippet config
- XPluginOptions

Design System

- Changing theme brand and component styles
- Changing config options or adding plugins

Translations

- Setting a default locale
- Adding locales
- Adding more messages

Adapter

- Adding endpoints
- Extending platform adapter endpoints
- Overriding schemas

Modules

- Adding a new module
- Extending an existing module
- Initializing modules

Event handling

- Listening and reacting to X events
- Callbacks

- Creating a new X Event

Customizing
· Init Options
· Design System
· Translations
· Adapter
· Modules
· Event Handling

Snippet Config

It is a global config object for the X Components setup.

This JSON object is passed in the `initX` object of the `window` and by default it takes the values from the URL (without using the URL module). These configurations will affect the whole app.

```
window.initX = {
  instance: popFromURLParameters('instance') || 'empathy',
  env: getEnv(),
  scope: popFromURLParameters('scope') || 'desktop',
  lang: popFromURLParameters('lang') || 'en',
  device: popFromURLParameters('device') || 'mobile',
  uiLang: popFromURLParameters('uiLang') || lang,
  currency: popFromURLParameters('currency') || 'EUR',
  consent: popFromURLParameters('consent') !== 'false'
  [...]
}
```

*The whole list of parameters to receive through the `SnippetConfig` are defined in the X Components library's section x-installer.

Adding new parameters

All params defined in the `SnippetConfig` object are treated as `ExtraParams`, and, **unless it is not specified**, they will be attached to all the requests by default. If we want to avoid this, we just have to pass the **whole** list of the excluded extra parameters to the `SnippetConfigExtraParams` component when we use it in our `App.vue` file.

Finally, the new param should be added to the `SnippetConfig` interface using the `shims-x-components.d.ts` file.

```
<SnippetConfigExtraParams :excludedExtraParams="excludedExtraParams" />

const excludedExtraParams = [
  'callbacks',
  'uiLang',
  'consent',
  'documentDirection',
  'currency',
  'filters',
  'isolate',
  'queriesPreview',
  'cart',
  'wishlist'
];

// Extend type
declare module '@empathyco/x-components' {
  export interface SnippetConfig {
    wishlist: Result['id'][];
  }
}
```

XPluginOptions

Initializing the `XPlugin` and apply it to the Vue instance.

The `XPlugin` is used to add the basics the X Component need to work, like the adapter and the store, but can be used to override things like the modules stores and the wiring.

On top of these options, there are install options, specified when installing the X in a Vue app and, aside of the plugin options, adds things like installing extra plugins with access to the snippet config for the initialization.

```
async installExtraPlugins({ app, snippet }) {
  const i18n = await I18n.create({
    locale: snippet.uiLang,
    device: (snippet.device as string) ?? device.deviceName.value,
    fallbackLocale: 'en',
    messages
  });

  app.use(i18n);
  app.config.globalProperties.$setLocale = i18n.setLocale.bind(i18n);
  app.config.globalProperties.$setLocaleDevice = i18n.setDevice.bind(i18n);

  return {
    i18n: i18n.vueI18n
  };
}
```

X Design System

Based in the Tailwind CSS framework and exported as a plugin, it provides:

The XDS Theme

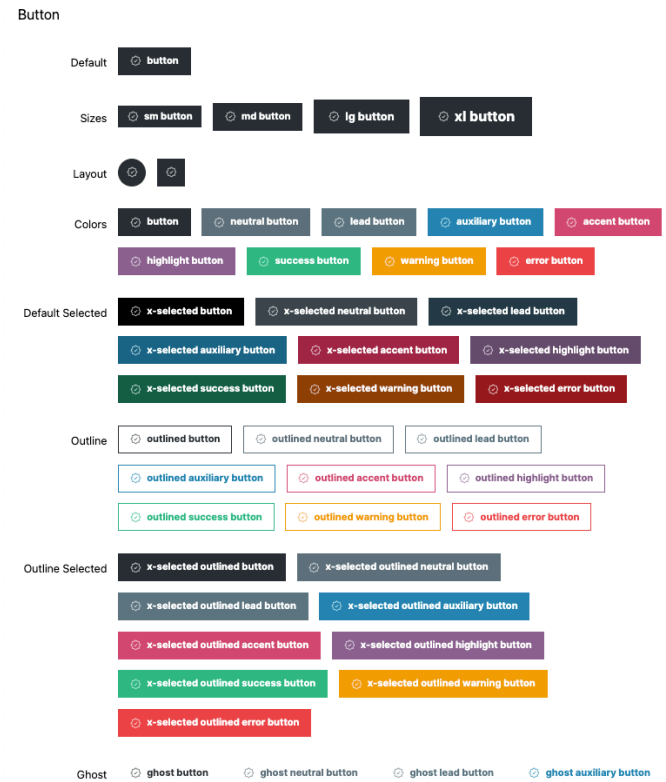
A set of colors, fonts, spacings, and other properties that the base utilities of Tailwind use, as well as the components.

Custom components' styles

The custom components are bundles of styles that modify the look and feel of the X Components when applied to those elements. Things like `x-button`, `x-suggestion` and `x-icon` are used throughout the customer's setup. There are also different variants designed to be combined: variants such as color, size, style (outlined, ghosted...), and focused on user experience (disabled, selected, checked...).

XDS Showcase

It can be run locally to see the whole design system components and variants, thus click-copy each item's classes to implement them in a setup quickly.



Changing theme & component's styles

The theme is defined in the plugin-options.js file. The XDS automatically generates classes to implement theme colors or other properties that can be extended.

```
module.exports = {
  theme: {
    colors: {
      newColor: {
        50: '#FFCD4C'
      }
    },
    extend: {
      // The `x-shadow-empathize` class will be
      boxShadow: {
        'empathize': '0 0 10px 0 rgba(0, 0, 0,
      }
    }
  },
  components: ({ theme }) => ({
    // Classes applied in components can be ove
    '.picture': {
      aspectRatio: '1/1'
    },
  })
};
```

Changing config options or adding plugins

`tailwind.config.js` file can be found in the root of the project. There you can modify some configuration options, such as the prefix used to apply in CSS generated classes, add custom plugins or more features that are still not available in the core of framework.

```
1   const { xTailwindPlugin } = require('@empathyco/x-tailwindcss');
2   const options = require('./src/tailwind/plugin-options');
3
4   /** @type {import('tailwindcss').Config} */
5   module.exports = {
6     content: [
7       './public/index.html',
8       './src/**/*.vue',
9       'node_modules/@empathyco/x-components/**/components/**/*.js'
10    ],
11    prefix: 'x-',
12    important: true,
13    plugins: [xTailwindPlugin(options)]
14  };
```

Customizing
· Init Options
· Design System
· Translations
· Adapter
· Modules
· Event Handling

Translations

To handle the internalization for the customer's setups, X Archetype is integrated with the vue-i18n package. It is imported as a plugin and initialized as an extra plugin.

Setting a default locale

The default locale, is taken from the `uiLang` value defined in the `SnippetConfig`.

```
42        const i18n = await I18n.create({
43          locale: snippet.uiLang,
44          device: (snippet.device as string) ?? device.deviceName.value,
45          fallbackLocale: 'en',
46          messages
47        });
```

Adding locales

Each locale has its own language file, so create one for the new language you want to add, and also add it to the `index.ts` file to get it lazy loaded.

```
18    export const es = () => import('./es.messages.json');
```

Adding more messages

Language files are written in `JSON` format, and each entry is typed under a `Messages` interface to avoid forgetting them. So, to add a new message, add it to the `Messages` interface and then to all the `JSON` files.

The language files are **device-featured** so that we can specify different content for desktop or mobile devices. This works by adding a mobile object with the keys intended to be overwritten.

```
    "mobile": {
      "nextQueriesGroup": {
        "message": "Los que buscaron {query} también han visto:"
      },
      "nextQueryPreview": {
        "viewResults": "Ver todos"
      },
      "totalResults": {
        "message": "{totalResults} resultados"
      },
      "queryPreview": {
        "viewResults": "Ver todo"
      }
    }
  }
```

* The X Translations package aids to the creation of language files by converting from `.csv` files to the `JSON` format that the package uses and vice-versa.

Adapter

Adding endpoints

A new endpoint adapter should be created:

```
export const dummyEndpointAdapter = endpointAdap
  endpoint: 'https://dummyjson.com/users/search'
  defaultRequestOptions: {
    id: 'dummy-search'
    // Rest of options
  },
  requestMapper: schemaMapperFactory<OwnAppReque
    q: 'query'
  }),
  responseMapper: schemaMapperFactory<ApiRespons
    people: ({ users }) =>
    users.map(user => {
      return {
        id: user.id.toString(),
        name: user.firstName
      };
    })
    // Rest of mapping
});
```

And added in our adapter to be included:

```
import { platformAdapter } from '@empathyco/x-ad

export const adapter = platformAdapter;
adapter.dummySearch = dummyEndpointAdapter;
```

* Additionally it should also be declared as part of the `XComponentsAdapter` interface as follows:

```
import { EndpointAdapter } from '@empathyco/x-ad

declare module '@empathyco/x-types' {
  export interface XComponentsAdapter {
    dummy: EndpointAdapter;
  }
}
```

Customizing
- ~~Init Options~~
- ~~Design System~~
- ~~Translations~~
- Adapter
- Modules
- Event Handling

Extending platform adapter endpoints

An adapter can be extended when we can reuse most of the stuff, but we need to change some other, for example, the endpoint and the response mapper.

```
platformAdapter.search = searchEndpointAdapter.extends({
  endpoint: 'https://dummyjson.com/users/search',
  responseMapper: searchResponseMapper
});
```

Overriding or extending schemas

X Platform Adapter's schemas are created using a `MutableSchema`, that means their source and targets can be modified (extended, overriden, replaced) to suit our needs.

```
// Faked mapping for the purpose
resultSchema.$override<DummyResult, Partial<Resu
  id: 'product_id',
  name: 'product_name',
  // ... Rest of the fields
});
```

* The override strategy enables us to modify some predefined platform API models to our own use case.

```
export interface DummyResult extends PlatformRes
  active_price: number;
  app_price: number;
// ... Rest of fields
}
```

* The `DummyResult` has been declared extending the Empathy's `PlatformResult`, so we can keep what is defined in the original model and evolve it to suit our needs.

Customizing
· Init Options
· Design System
· Translations
· Adapter
· Modules
· Event Handling

Modules

Adding a new module

As seen in the x-modules section, a module will have a store, components, and ist own wiring. It should be declared and registered, and also be added to the `XModulesTree` interface:

```
export const dummySearchXModule: DummySearchXModule = {
  name: 'dummySearch',
  storeModule: dummySearchXStoreModule,
  storeEmitters: dummySearchEmitters, // * Not all the modules will have emitters, they should be d
  wiring: dummySearchWiring
};

XPlugin.registerXModule(dummySearchXModule);
```

The store will include the module's state, getters, mutations and actions, and they should be typed accordingly.

```
export const dummySearchXStoreModule: DummySearchXStoreModule = {
  state: () => ({}),
  getters: {},
  mutations: {},
  actions: {}
};
```

Customizing
· ~~Init Options~~
· ~~Design System~~
· ~~Translations~~
· ~~Adapter~~
· Modules
· Event Handling

Initializing modules

As seen before, modules will be effectively registered as soon as the components that are using them are mounted, or, if necessary, this can be done as soon as the application is initialized.

```
/**
   * XModules that will be registered on init.
   *
   * - dummySearchXModule is loaded on init to have info available to render
   * stuff before any search is done.
   */
  initialXModules: [dummySearchXModule]
```

This should be done carefully, as will increment the loading app size on init. **So it is not recommended to do it with all the modules.**

Extending an existing module

Use case: adding wiring for the `UserModifiedAdd2cart` event.

This behavior is defined as a module option:

```
import { namespacedWireCommit, XModuleOptions } from '@empathyc
import { Result } from '@empathyco/x-types';

const updateResultOutOfStock = namespacedWireCommit('search')(
  'updateResult',
  ({ eventPayload: result }: { eventPayload: Result }) => {
    return {
      id: result.id,
      isOutOfStock: result.isOutOfStock
    };
  }
);

// Configuration options for installing the search XModule.
export const searchInstallOptions: XModuleOptions<'search'> = {
  wiring: {
    UserModifiedAdd2cart: {
      updateResultOutOfStock
    }
  }
};
```

And set in the xModule's install options:

```
import { searchInstallOptions

 xModules: {
   search: searchInstallOption
 }
```

Event Handling

Listening and reacting to X events

Depending on the purpose, this can be done by **extending some module's wiring**, as seen before; but also **accessing to the x events' bus**.

Extend a module's wiring

```
export const installXOptions: InstallXOptions = {
 //...
  xModules: {
    search: {
      wiring: {
        UserModifiedAdd2cart: { updateResultOutOfStock }
      }
    }
  }
};
```

Accessing to the x-bus to react to an event

Use the x-component's `useXBus()` composable to access to its `on` function.

```
const xBus = useXBus();
xBus.on('UserClickedAdd2Cart', false).subscribe(
```

Accessing to the x-bus to emit an event

Use the `useXBus()` composable to access to its `emit` function.

```
if (hasStock === null) {
  await xBus.emit('UserModifiedAdd2cart', {
    id: (payload.payload as Result).id,
    isOutOfStock: true
  });
}
```

Callbacks

Callbacks are functions defined in the client side that will be called when the specified events happen. Defined in the `SnippetConfig` under the `callbacks` key, they are used to react to events in the X Components from outside.

The example below shows a common use case: listening to the add to cart related events to update the cart and keep the search layer and customer's website synced.

```
    callbacks: {
      UserClickedResultAddToCart: function (result) {
        cart = cart + 1;
        InterfaceX.setSnippetConfig({
          cart
        });
      },
      UserClickedResultRemoveFromCart: function (result) {
        cart = cart > 0 ? cart - 1 : 0;
        InterfaceX.setSnippetConfig({
          cart
        });
      },
      UserClickedAddToCartIcon: function () {
        console.log('Client Cart icon pressed');
      }
    }
  };
```

Callbacks are integrated in the X-Archetype's `App.vue` file by importing its dedicated component.

`snippet-callbacks-vue`

```
const eventListeners = computed<XEventListeners>(() => {
  const callbacks = snippetConfig?.callbacks;
  return callbacks
    ? map(callbacks, (eventName, callback) => {
        return (payload: unknown, metadata: WireMetadata) => {
          const callbackReturn = callback(payload as never, metadata);
          xBus.emit('SnippetCallbackExecuted', {
            event: eventName,
            callbackReturn,
            payload: payload as never,
            metadata
          });
        };
      })
    : ({} as XEventListeners);
  });
  return { eventListeners };
}
```

2. Iterating and setting up the callback.

Creating a new X Event

When a new event is needed, it can be created by extending the `XEventsTypes` interface to have it available in the `XBus`. This way we can keep consistency and take advantage of the whole library ecosystem.

Steps to follow:

· Use the `BaseEventButton` component. This component is ready to receive an `XEvent` by prop and will use the `xBus` to emit it.

· Extend the `XEventsTypes` interface using the `shims-x-components.d.ts` file.

```
<BaseEventButton
  class="x-button x-button-tight x-h-24 x-w-24 x-text-lead-50 hover:x-button-lead"
  :class="{ 'x-button-lead': isWishListed }"
  :events="wishlistEvents"
>
  <HeartIcon class="x-icon-lg" />
</BaseEventButton>

// XEvent and payload
const wishlistEvents = { UserClickedResultWishlist: props.result }

// Extend type
declare module '@empathyco/x-components' {
  interface XEventsTypes {
    UserClickedResultWishlist: Result;
  }
}
```