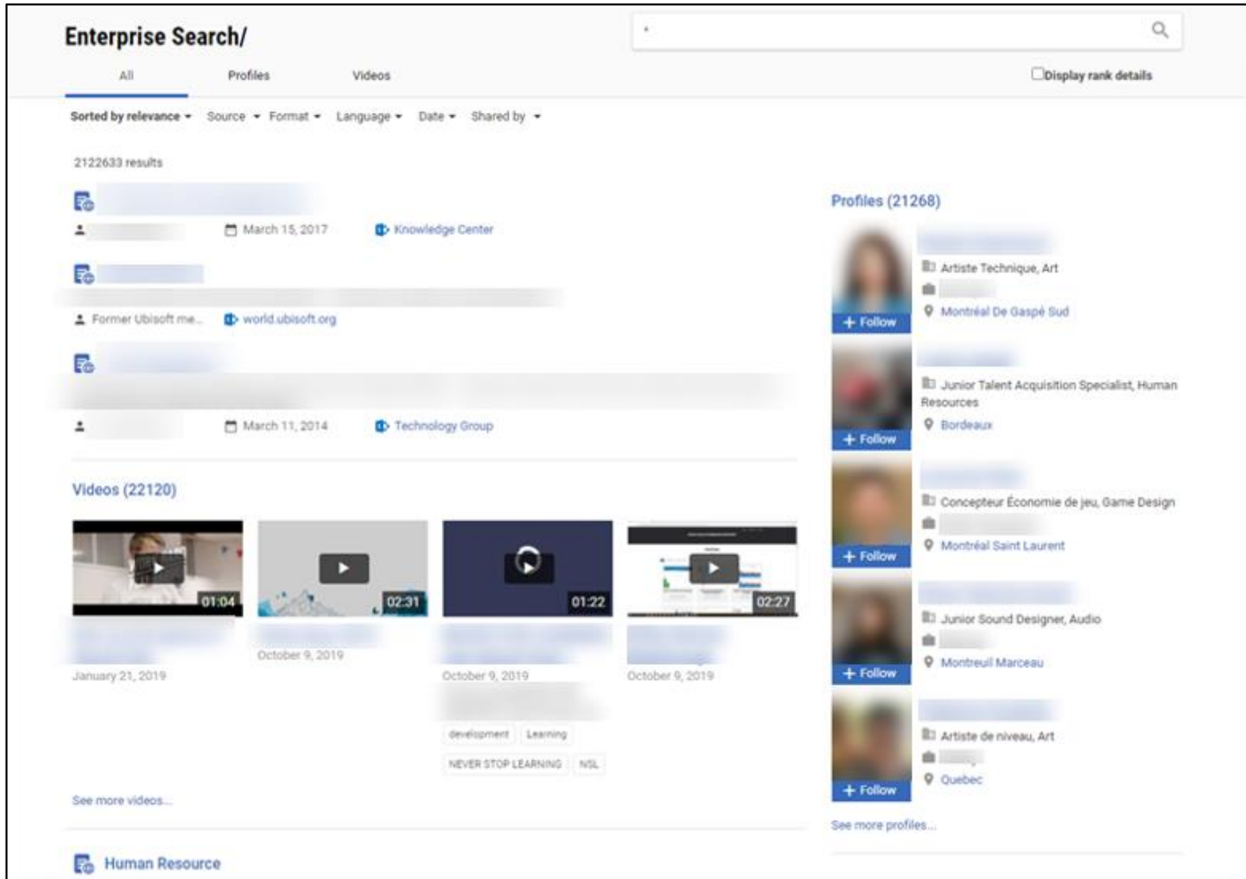
THE UBISOFT JOURNEY WITH MICROSOFT SEARCH

MICROSOFT SEARCH IMPLEMENTATION USE CASE

Ubisoft Entertainment | 2, avenue Saint-Mandé | 94160, France | T : +33 (1) 48 18 50 00
www.ubisoft.com

INTRODUCTION

In 2023, Ubisoft decided to update its old global search application relying on SharePoint 2013 on-premises and Business Connectivity Services connectors. SharePoint Server 2013 was about to be decommissioned (i.e. Microsoft end of support) and we needed to implement a suitable enterprise-wide solution as a replacement. Also, we needed something more modern to match our new requirements as well.



Ubisoft Search Center based on SharePoint 2013 and BCS connectors.

A WORD ABOUT UBISOFT

Ubisoft is a creator of worlds, committed to enriching players' lives with original and memorable entertainment experiences. Ubisoft's global teams create and develop a deep and diverse portfolio of games, featuring brands such as Assassin's Creed®, Brawlhalla®, For Honor®, Far Cry®, Tom Clancy's Ghost Recon®, Just Dance®, Rabbids®, Tom Clancy's Rainbow Six®, The Crew® and Tom Clancy's The Division®. Through Ubisoft Connect, players can enjoy an ecosystem of services to enhance their gaming experience, get rewards and connect with friends across platforms. With Ubisoft+, the subscription service, they can access a growing catalog of more than 100 Ubisoft games and DLC. For the 2022–23 fiscal year, Ubisoft generated net bookings of €1.74 billion. To learn more, please visit: www.ubisoftgroup.com.

ABOUT THE AUTHORS

This document has been written initially by (in alphabetical order) Franck Cornu, Microsoft 365 developer, Stephanie Daigle, IT Manager Microsoft 365 and Mihaela Nita, Product Owner as members of the Enterprise Search project.

ABOUT THE AUDIENCE

This document is mainly targeted to IT technical teams to help them to implement Microsoft Search in their own organization learning from a real-world implementation. Therefore, it requires a basic knowledge of Microsoft Search and its concepts. At Ubisoft, we don't pretend our approach is the only way to go and our implementation is an example among other possibilities. It all depends on your context 😊.

A SPECIAL THANK YOU

This document is also dedicated to all people who contributed to this project and made it a success (in alphabetical order):

Léo Bousquet, Content Strategist KM - Knowledge Management

Iulian Colesnicenco, QA Engineer - Quality Control

Laura-Simona Dobresenciuc, Technical Lead - IT Development

Jessica Fernandes, IT Associate Director - Communication & Collaboration

Andrei-Lucian Ghenea, Senior Web Designer

Rishi Nanda, IT Manager - Connect & Explore Product Team

Manuela Neaga-Budoiu, Manager Software Engineering - Collab & Comm

Cosmin-Alexandru Serban, FullStack Software Developer

Table of Contents

INTRODUCTION 2
A WORD ABOUT UBISOFT 3
HOW DID WE PROCEED? 6
- 1. Identify the project scope and vision 6
- 2. Source inventory 6
- 3. Functional/non-functional requirements 7
- 4. Identify enterprise search solutions 8
- 5. Implementation 8

WHY DID WE CHOOSE MICROSOFT SEARCH? 9
- The Microsoft Fastrack Program 9

OUR NEW SEARCH APPLICATION 10
HOW WE MANAGED OUR SEARCH INTERFACE? 11
- Custom built application versus the Microsoft Search default interface 11
- Our "Search as a service" approach 13
  - Reusable web components 13
    - The Microsoft Graph Toolkit 16
    - Styles management 16
      - EGG internal design system 16
      - Figma and TailwindCSS 17
  - Connectors 18
- How do we handle results display? 19
  - Styling adaptive cards 21
  - Manage experience between custom/OOTB UI 22
  - Video formats support 23
    - Authentication challenges with videos 23
    - Manage performances for videos. 24
  - Web components testing 25
- How do we handle analytics? 26
  - Track events in adaptive card 27
- Special case of SharePoint integration 28

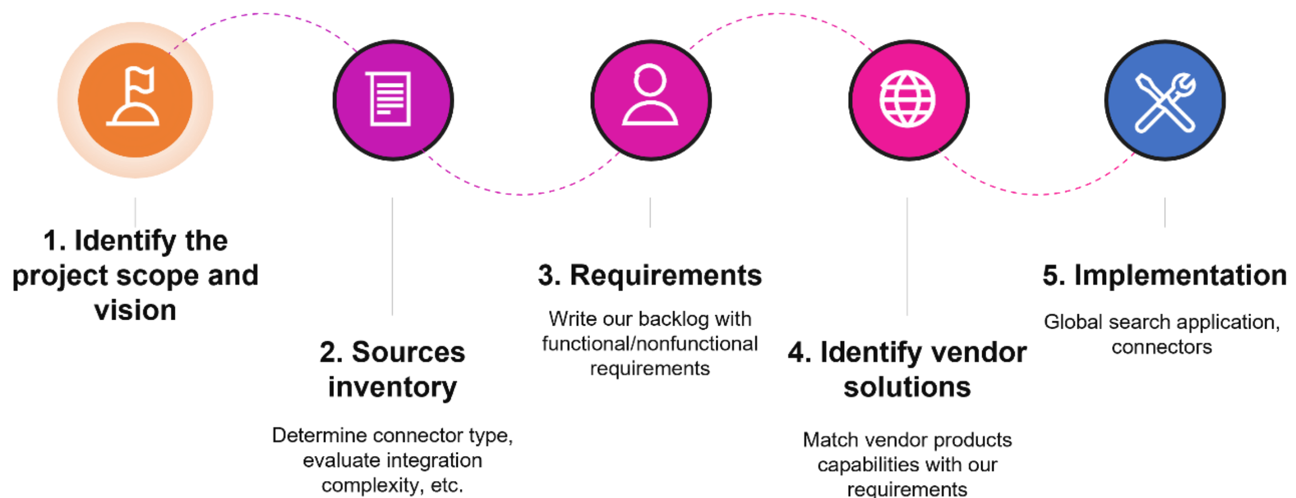
WHAT WAS OUR SEARCH DATA STRATEGY? 31
- Search schema definition 31
- Unified search schema 34
  - Custom connectors 34

Microsoft connectors ... 35
SharePoint and OneDrive ... 35
Augmenting source metadata ... 40
"Source" property ... 40
"Refinable" and "Searchable" for the same property ... 40
How did we build our custom data connectors? ... 42
Building a custom Graph connector, our experience ... 42
Full crawl algorithm ... 42
Incremental crawl algorithm ... 42
Managing errors during crawls ... 43
Configuring alerts ... 45
Key tips when building a custom connector ... 47
Data source permissions strategy ... 47
Manage source custom permissions ... 48
Managing connector settings ... 49
Credentials ... 49
Connector settings ... 49
Test and debug a connector ... 51
Test and debug locally ... 51
Debug a deployed connector ... 51
DevOps with custom connectors ... 53
Managing environments (tenants/sources) ... 54
Permissions and domain constraint ... 55
CONCLUSION AND CHALLENGES ... 57
Managing search quotas (i.e., licenses) between our two Microsoft tenants. ... 57
Relevancy in the interleaved results feed. ... 57
Connector debug process is slow and time consuming. ... 57
Find workarounds for some API limitations! ... 57
Confluence on-premises indexation ... 57
BONUS: DO THE SAME AS US WITH THE OPEN-SOURCE VERESION OF WEB COMPONENTS ... 59

HOW DID WE PROCEED?

Project Context

Business side	Technical side
• Around 19 000 employees. • 45+ studios spread all over the world. • 65 spoken languages. • 90 nationalities. • Multiple project teams involved.	• Microsoft 365 for 3 years. • Multiple heterogenous internal sources to index with millions of items to crawl. • Many on-premises systems. • Strict security context to comply with.



1. Identify the project scope and vision

The initial requirement was to replace the global enterprise search web application, the one used by all Ubisoft employees to search for their content across all sources. However, this change also implied to update the underlying global search engine, so we also wanted to cover the search usage elsewhere in the Ubisoft applications ecosystem.

2. Source inventory

At Ubisoft, in addition to SharePoint and OneDrive content, we have many custom applications representing multiple sources to index. Some of them were already indexed in our old SharePoint farm using BCS connectors and some of them were new to index. Among the sources we had to integrate:

- Our internal employee profiles database called "Profiles" aggregates data from multiple sources (Azure AD, our HR database, custom metadata repository for people etc.).
- Our local studio intranets are based on WordPress.

- Our business terms wiki called "Ubipedia".
- Our ServiceNow knowledge base.
- Confluence on-premises.
- Custom made internal social network site.
- etc.

We did the inventory of all these sources and gathered multiple information for each:

- Technology to access data (ex: REST API, database).
- Does it have user permissions or roles.
- Owner/contact of the source.
- Number of items to crawl.
- Business priority.
- Etc.

This step was important to determine if we already had enough **Microsoft Search quotas** in our tenant or if we needed to purchase additional ones (allowed quotas are based on number licenses you have). Also with this inventory, we clearly identified the needed connector type: a built-in Microsoft one (Confluence, ServiceNow) or a custom built one. In the latter, it also gave us some clues about expected integration complexity regarding the data source technology (SDK available, API, etc.).

3. Functional/non-functional requirements

Then we created a backlog based on these sources and agnostic from any technology, each story having a priority and an implementation effort score (*Small* to *Extra Large*).

We drafted the stories with the format *"Person - Action"*, like *"Ubisoft Employee - Filter Confluence content"* or *"Admin - Manage Relevancy"* with a quick description and minimum acceptance criteria with enough information to estimate the effort.

Not only functional requirements were considered during our decision, but we also considered the technological context at Ubisoft in addition to people skills and availabilities.

Also, we had a couple of non-functional requirements like:

⇨ **User interface**

- Localization support (labels, values, etc.). At least French and English.
- Analytics on user actions/monitoring.

⇨ **Data**

- Permissions support on original sources. Users will not see content they do not have access to.
- Get all items in the same results feed seamlessly.
- Manage and adjust relevancy settings (ex: boost specific results).

⇨ **Technological**

- Performances: ability to ingest millions of items in a reasonable amount of time for a full crawl (maximum 1 week).
- Ease of implementation for custom connectors.

- Convenient REST API to retrieve data.
- Integrate the search feature easily to other Ubisoft internal teams and have enough flexibility to fit **our** and **their** business requirements (source scopes, query configurations, etc.) (more on this part later).

4. Identify enterprise search solutions

We finished by identifying all search solutions on the market suited for our requirements based on the vendor technical sheets. We had an initiative a few years ago to replace SharePoint with the, Lucidworks Fusion Platform but it failed for several reasons. In the end, we pre-selected the two following products that could match our requirements:

- Elastic Enterprise Workplace search, that was already used internally by other products at Ubisoft.
- Microsoft Search, part of our existing Microsoft 365 tenant.

5. Implementation

Finally, the final step was to implement the strategy based on the chosen technology. The implementation covered:

- The global search application itself.
- The custom connectors to integrate our internal data sources.

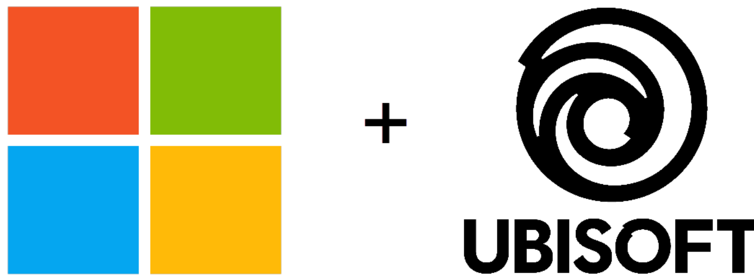
WHY DID WE CHOOSE MICROSOFT SEARCH?

Spoiler, we chose Microsoft Search as our enterprise-wide search engine. Among the main reasons:

- Well integrated in the Microsoft 365 suite and SharePoint/OneDrive which represent our biggest content source (mainly for documents).
- Built-in connectors we can use for our other major sources (Confluence, ServiceNow, SQL).
- Easy to implement custom connectors using the .Net SDK.
- Stable REST API available making it possible to build custom search experience on top of it (Microsoft Graph).
- We basically already paid for it through our Microsoft 365 licenses and had sufficient quotas from the start…

The Microsoft Fastrack Program

Another great reason we chose Microsoft Search is because Ubisoft had the opportunity to participate for the Microsoft fast track program.



Microsoft + Ubisoft: Fastrack Program

From Microsoft:

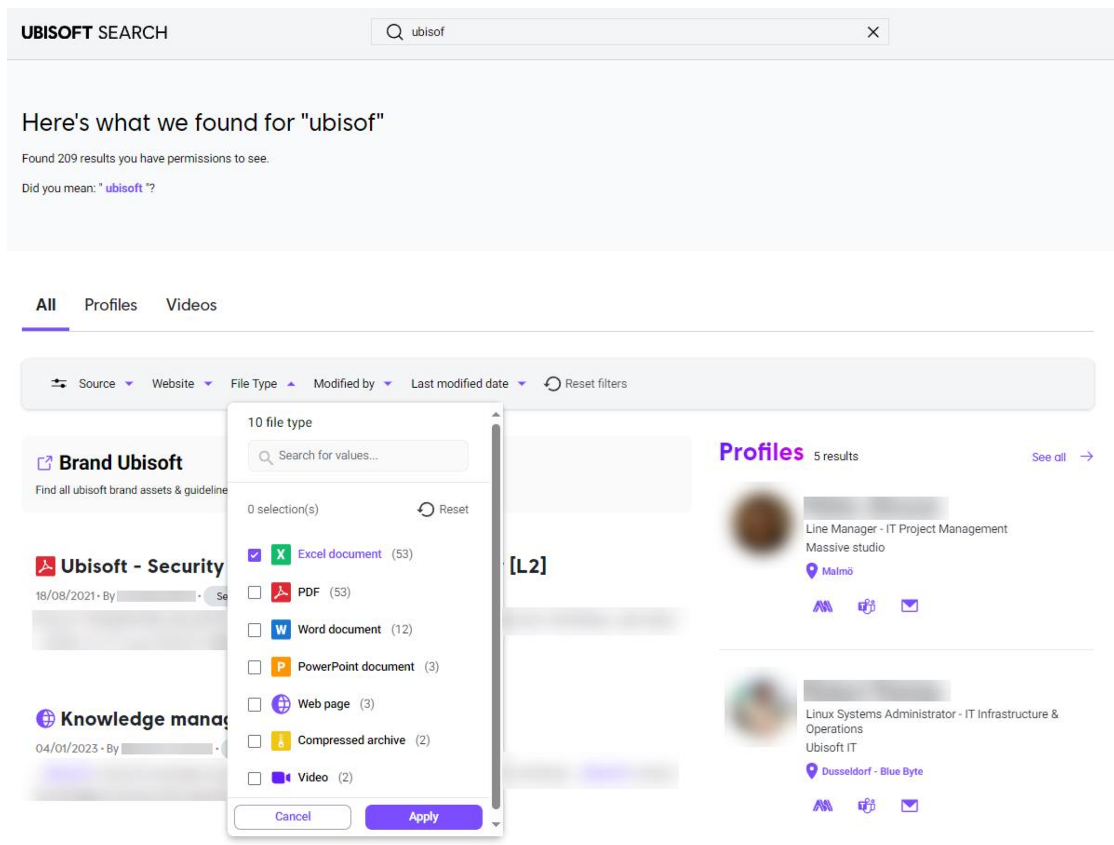
> *"The program is for enterprises that are committed to deploy Microsoft Search as their primary workplace search solution. Customers get to work closely with Microsoft to influence the direction of the product."*

The program duration was about 8/9 months in our case. During this time, we worked closely with the Microsoft Search product team on different aspects (connectors, API, data relevancy, etc.) and had early access to the latest features to share our blockers/feedback.

Overall, this was a great opportunity and experience for us and although the program is now ended, we still collaborate closely with Microsoft through diverse channels to improve the tool by testing features and sending feedback.

OUR NEW SEARCH APPLICATION

After a few months of development, we did our first soft launch on March 2023 (without analytics). The official launch was made in April 2023. From now on, the search application is continuously updated to integrate new features.



Ubisoft new search application

The interface is split into multiple search verticals ("All," "Profile" and "Videos"), the first one aggregating results from all sources (except profiles) in an interleaved feed.

In the next chapters, we detail how we proceed regarding our:

- UI strategy
- Data strategy
- Connectors strategy
- Search as a service approach

HOW WE MANAGED OUR SEARCH INTERFACE?

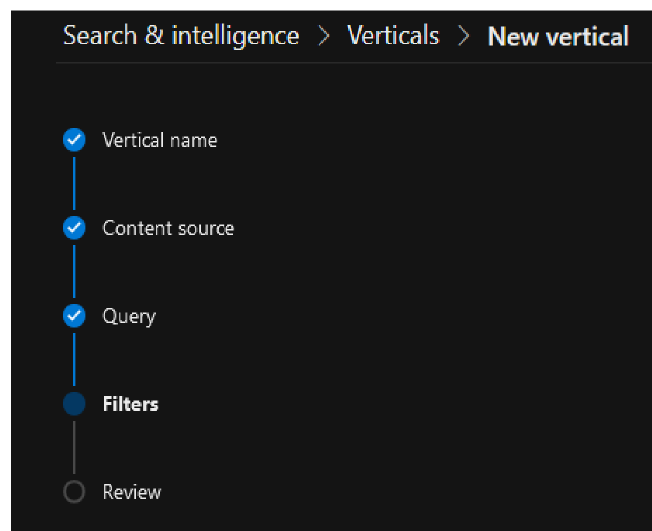
Custom built application versus the Microsoft Search default interface

Microsoft Search is the search engine for the Microsoft 365 suite. Therefore, it can be accessed from many locations (office.com, bing.com, sharepoint.com and even Windows) and because of this particularity, the customization options are limited. That is understandable as the goal is to guarantee a uniformized experience across all locations and Microsoft cannot afford to let developers customize their default experiences the way they want, that would be a huge burndown to support. Fair enough Microsoft.

However, it is still possible to lightly modify the UI. If we look at available interface customizations for Microsoft Search and their limitations, they can be summarized as follow:

⇨ Custom verticals with:

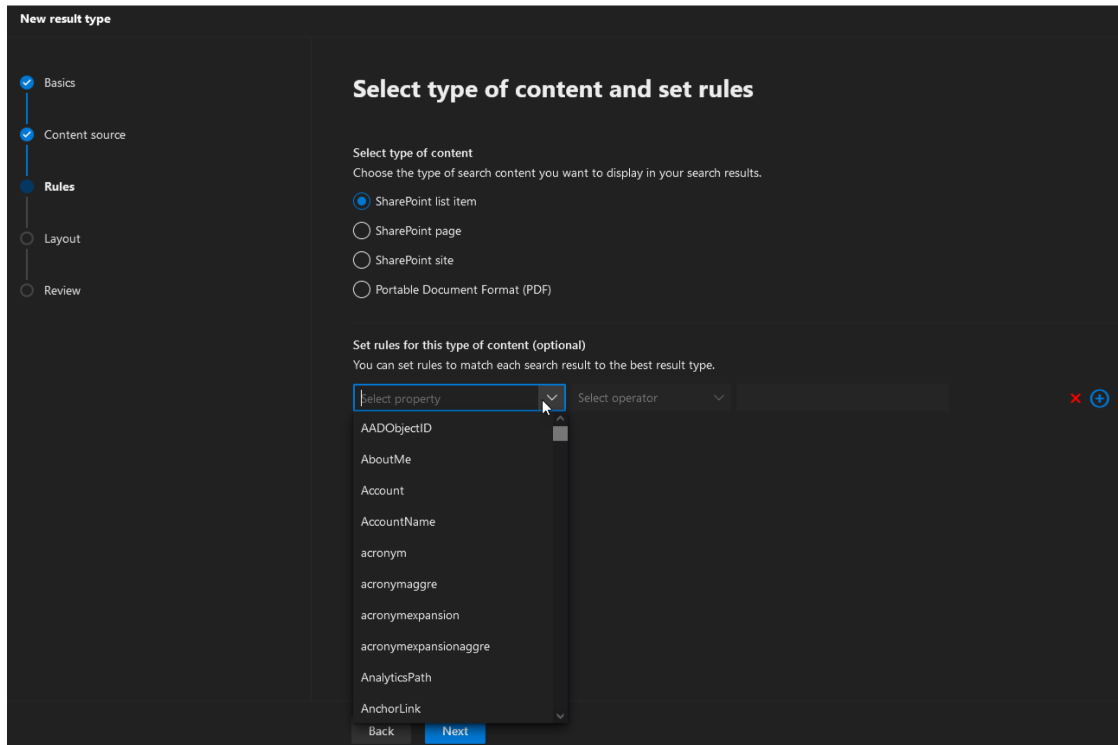
- Custom name (non-localizable).
- Custom default KQL search query with query variables support.
- Custom content sources, but either multiple connectors together (i.e., external sources) or SPO content but not at the same time.
- Custom filters:
    - Filter name (non-localizable)
    - Filter type (date time or text).
        - *"Date Time"*
            - Comes with predefined ranges that cannot be changed (path month, 3 months, etc.).
            - User manual value input.
        - *"Text"*
            - Single or multi values.
            - No filter icons or grouped filters.
- Enable or disable vertical but not able to deactivate some of the default ones, like "People". In our case for instance, we use our own "Profiles" source repository, so it comes as a duplicate and it misses some user information as not all properties are set in Azure Active Directory.
- Ability to create verticals at SharePoint site level, quite useful to build "mini search centers."
- All these additional limitations.



Custom vertical

⇨ **Result types**

- Create an adaptive card layout using conditions based on item properties.
- Only work for external connectors and few SharePoint item types (list item, page, site, PDF?!). For instance, using conditions on FileType with SharePoint list item type does not work.
- Limitation on adaptive card itself:
    - No HTML markup (for instance reusing web components from our other libraries) so no way to process user actions (ex: custom analytics, custom web components, etc.). If you add HTML markup in the card, it will be rendered as text in the UI by default.
    - No adaptive card actions support.
    - Cannot provide an host configuration to set common styles.



Custom result types

At Ubisoft, **we do LOVE custom things**. Considering our requirements, using the default Microsoft Search experience was not an option. Therefore, we compared two approaches for a custom solution:

- Build a custom search page directly in SharePoint using the well-known PnP Modern Search WebParts.
- Create a custom independent search application.

The first initiative was to revamp the global search application and one of our main requirements was to be able to integrate search in other internal applications as well (local intranet sites, etc.). In the end, the choice was made to go with a custom independent solution. This solution was a better option for us to be able to cover all our UI requirements.

⇨ *What happened to the default Microsoft Search experience?*

Users can still use default UI (as we cannot deny it) but it is not officially supported, and we left it "as is."

Our "Search as a service" approach

We already had at Ubisoft some internal libraries relying on the web components technology and we wanted something similar to be able to distribute this feature easily to other teams and have enough flexibility to fit **our** and **their** business requirements.

If every client-app would consume individually the Microsoft Graph API, they would duplicate the resources we already spent (time, people with necessary skills, security audit, legal audit and budget) as well as create very different user experiences on each application that might reduce ease of usability of the Ubisoft ecosystem.

Instead, we opted to position our search product as a service that can be integrated in other application with less effort and still offer customization without breaking the experience.

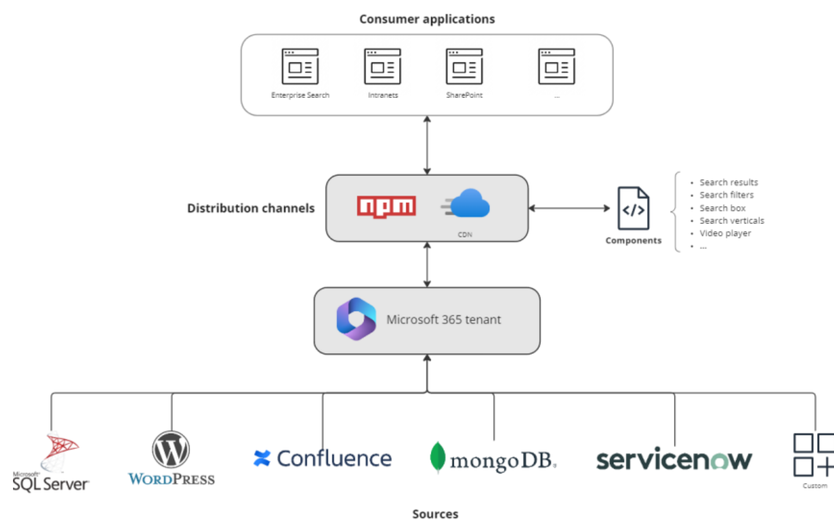
It means at Ubisoft, search is seen as a service internally developed and supported that can be used and customized by any internal team (reusable web components and connectors).

Reusable web components

We created an internal web components library that provides the following building blocks to be integrated in line of business applications:

Component	HTML tag	Purpose
Search results	`<ubisoft-search-results>`	Display results from Microsoft Search index according to configured parameters.
Search filters	`<ubisoft-search-filters>`	Display filters from connected search results components.
Search box	`<ubisoft-search-input>`	Allow users to enter free text search keywords.
Search verticals	`<ubisoft-search-verticals>`	Displays tabs to search by verticals.
Video player	`<ubisoft-video-player>`	Play video from SharePoint.

Components are distributed through a JavaScript bundle that can be included in any HTML page and through a **npm** package for developers.

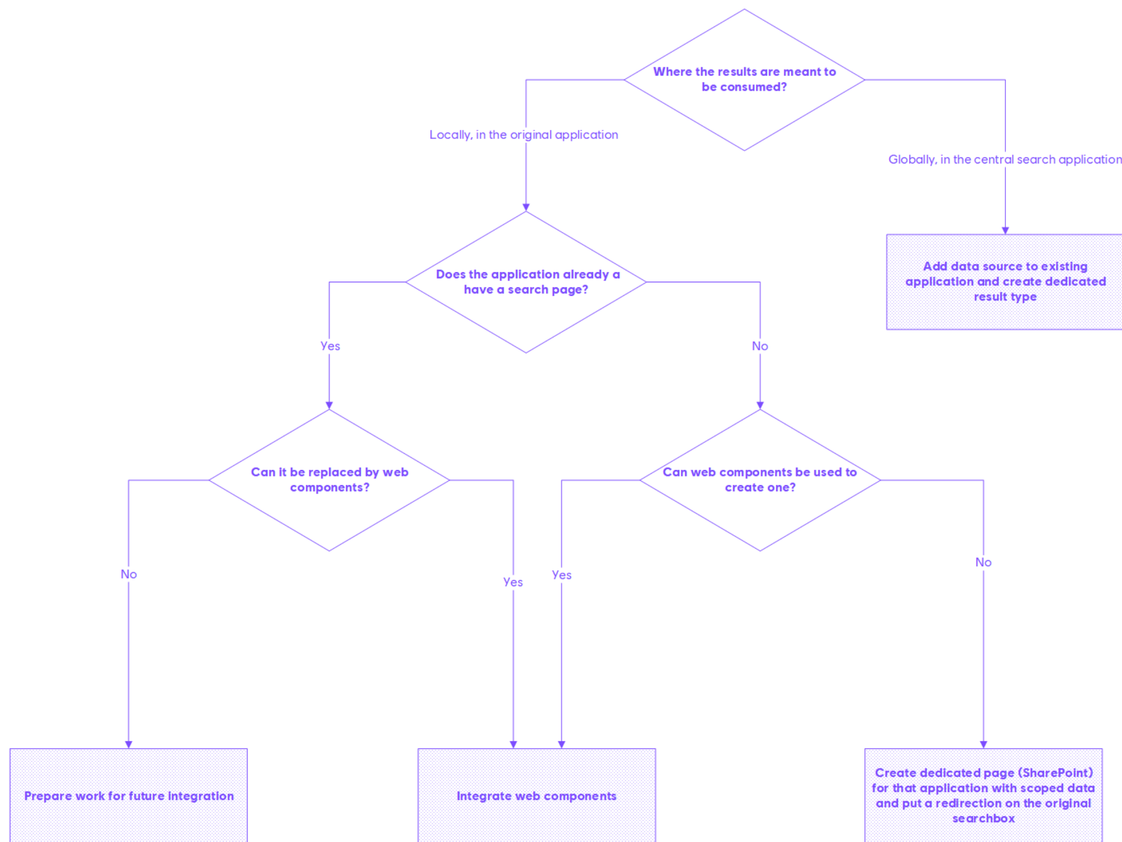


Search as a service architecture

To help promoting our components approach, we also build a playground (using Storybook) that acts as a live documentation allowing developer to quickly see how to integrate in their applications:

Our internal components "Storybook" documentation

This integration can occur at multiple levels in applications, and we distinguished multiple scenarios suing this decision tree:



Microsoft Search and search components integration for other applications decision tree

In the Ubisoft ecosystem, we can have certain applications where the global enterprise search application completely substitutes the search feature, for instance applications where web components cannot be used for whatever reasons (ex: our "Profiles" application where the internal search box is just a redirection to the global search application having a dedicated vertical).

For others, only their data are indexed and surfaced in the global search application to be more discoverable in the main results feed meanwhile the local search experience remains untouched. It may seem odd for end users to have two places to search sometimes but the reality is that not all applications are ready to be updated with the search web components. Sometimes the technical context does not meet our prerequisites (ex: no Azure AD authentication, etc.) or the value to integrate is not there yet. However, the long-term vision is clearly to normalize search experiences for all internal applications.

The Microsoft Graph Toolkit

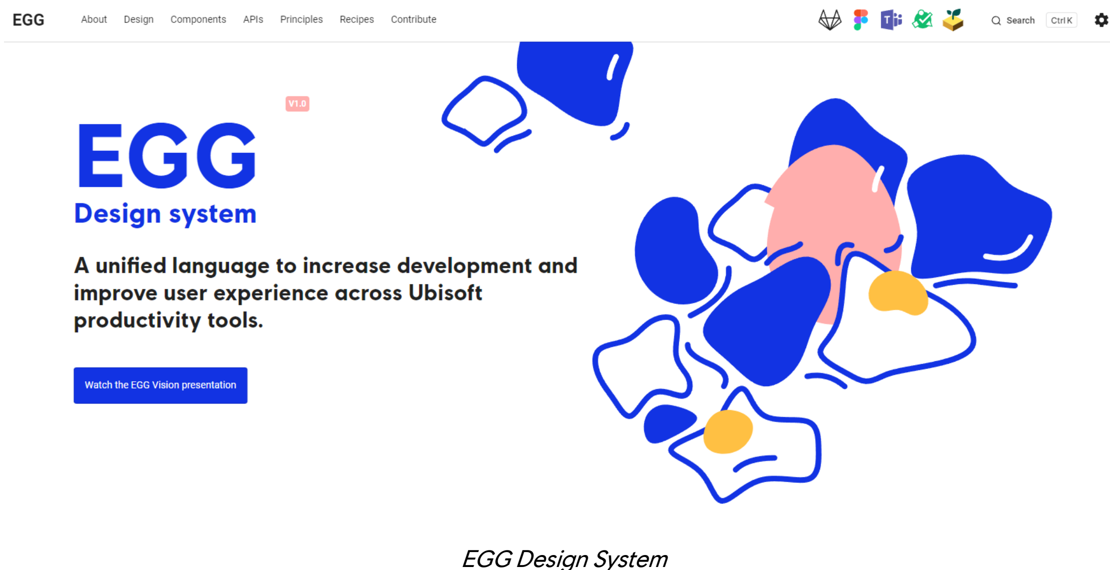
If you know the Microsoft Graph Toolkit already, you may know that it provides built-in ready to use components you can integrate and customize in your web applications. However, it also comes with an extensibility model allowing developers to create their own components on top of it. Instead of creating web components from scratch, we decided to use MGT as a base because of the following reasons:

- Comes with an authentication providers mechanism making it easy for components to consume Azure AD protected APIs, especially Graph API. For our specific use case, we were able to create our own provider to adapt to our environment and our seamless SSO authentication setup.
- Comes with localization support.
- It comes with full template support and a data binding syntax as a convenient wrapper over the native web components slot mechanism.
- Simple/flexible enough to be extended with the `@microsoft/mgt-element` base classes.
- Easy to distribute and test + well known underlying web stack (`TypeScript`, `Lit`).

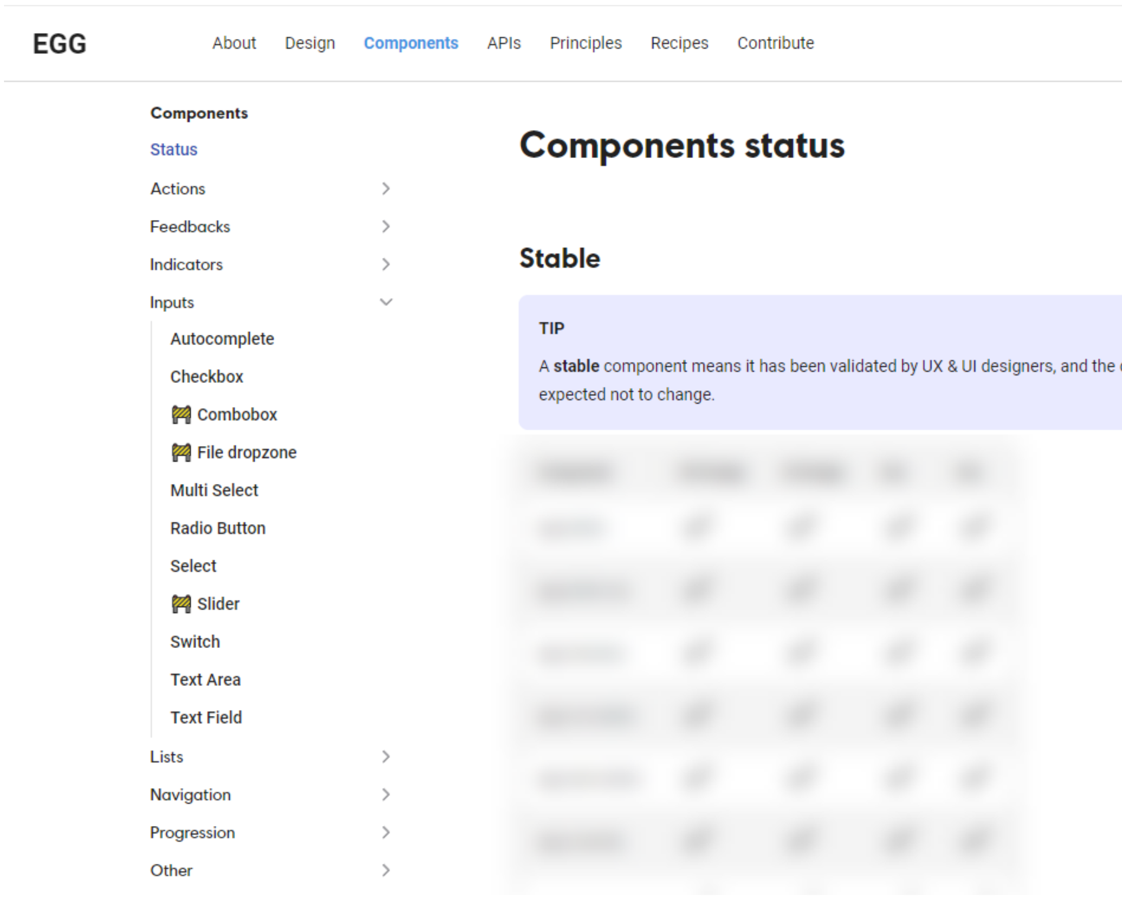
Styles management

EGG internal design system

At Ubisoft, we do have a components library called **"EGG"** providing a design system and basic building blocks to build applications, the same way Fluent UI does. These are `Lit` web components built over the open source "Lion" web components made by ING bank. They are used by most of our internal applications to harmonize the user experience and comply to organizational standards in terms of accessibility and design.



EGG Design System



EGG Components

Figma and TailwindCSS

In our search components, we do use EGG components in addition to another popular framework internally used at Ubisoft: TailwindCSS. As a developer, if you don't already know this framework, I strongly suggest you take a look. With TailwindCSS, you don't need to write CSS stylesheets anymore in your code (CSS, SASS, LESS, etc.). The styling is done with predefined/custom CSS classes. This is an example of what it looks like in a component:

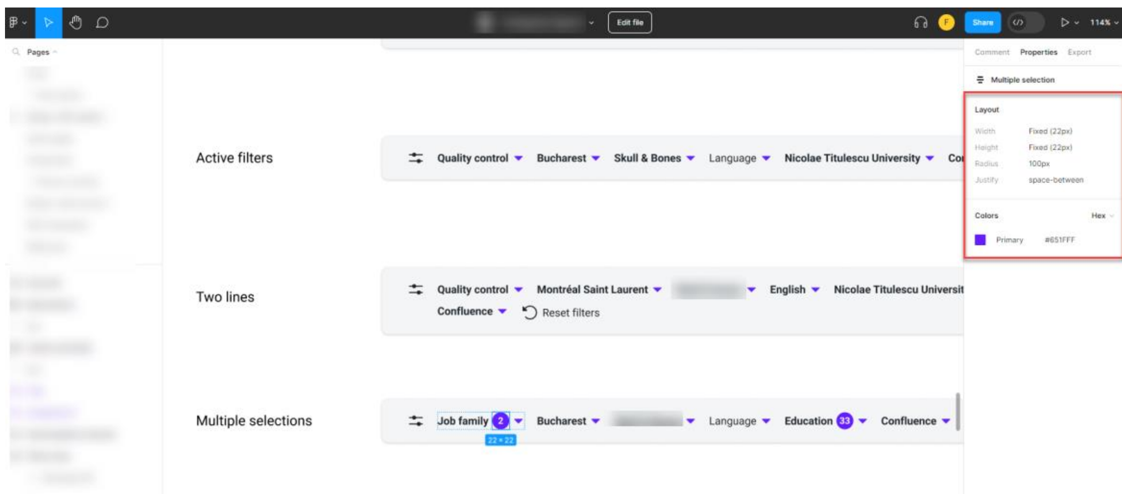
```
const renderSearchBox = html`
  <form @submit=${(e) => { e.preventDefault(); this.submitSearch();}} class=${this.theme}>
    <label for="searchbox" class="mb-2 text-sm font-medium text-gray-900 dark:text-textColorDark sr-only">Search</label>
    <div class="relative">
      <input id="searchbox"
        placeholder=${this.inputPlaceholder ? this.inputPlaceholder : this.strings.searchPlaceholder}
```

TailwindCSS example

It also has a great level of customization (ex: custom variables, values, etc.) and can accommodate any design. Another big advantage is you never ship unused CSS as all CSS code is generated at build time based on what you use in your code.

The writing may seem odd at first, but in our case, it saved us a ton of time because we only had one to two developers to implement the components and we all know developers are not good at writing good and efficient CSS :D.

As an input, we used Figma to define the UI made by our graphic designers. The main advantage of this tool is it is fairly simple to reproduce the same design as CSS values are already set correctly in mock-ups. For a developer, it just becomes a "translation" job using TailwindCSS. No need to overthink a stylesheets mechanism in your application.



Figma example

Connectors

For connectors, we do not provide "generic" ones. When a team needs their application data to be indexed by Microsoft Search, we work together to identify the needs and see if a native Microsoft connector can be used or a custom one needs to be developed using the C# SDK.

How do we handle results display?

Despite Microsoft Graph Toolkit provides its own way to build templates, we chose to only use adaptive cards for results display for the following reasons:

- Benefits from all prebuilt function to format display. It avoids creating specific helpers in components template context to cover edge cases and overall, it offers more possibilities than the Microsoft Graph Toolkit templating system.
- Standardized (almost) experiences across all search locations. Even if we use a custom search application, it does not prevent results from showing up in the default Microsoft Search experience and users (ex: office.com, bing.com,etc.). Although we do not expose the same verticals as our custom search application in the default Microsoft Search experiences, we keep this possibility, depending on how the tools evolves, without having to do a major refactoring in the future.
- Easy to update. Whether or not the card is used in a result type, we can quickly update it without the need to redeploy everything (ex: by updating the results type in search admin portal or uploading a new file version in the CDN).

We created a custom web component `<ubisoft-adaptive-card>` that allows us to display the JSON content of any adaptive card, either if it comes from a result type, or statically served from a CDN. This way, the customization technique remains the same for every item type. The strategy we use by sources is as follow:

Source type	Adaptive card strategy
External source	Microsoft Search result type
Everything that is not from external source (ex: office documents, videos, etc.)	Adaptive card .json file served from a CDN.

In the search application, the following HTML snippet allows us to cover all the scenarios with only a few lines. Exceptions are managed using Microsoft Graph Toolkit expressions, and results with result types simply processed dynamically based on the adaptive card payload returned from the API:

```
<!-- We target both SPO and external content as interleaved results -->
<ubisoft-search-results
    entity-types="externalItem,listItem"
    query-text="*"
    enable-result-types
    ...
    // Trimmed for brevity ...
>
    <template data-type="items">
        <div data-for='item in items'>
            <!--
              If the item has a result type associated, the content of this div is replaced by the generated HTML

              from the adaptive card payload coming from the HTTP response matching the 'hitId' item property
            -->
            <div id="{{item.hitId}}">

                <div data-if="item.filefamily === 'video'">
                    <ubisoft-adaptive-card
                        url="http://<cdn_url>/assets/cards/video_item.json"
                        context="{{item}}"
                    >
                    </ubisoft-adaptive-card>

                </div>

                <!-- "Default" item rendering, also used for documents -->
                <div data-else>
                    <ubisoft-adaptive-card
                        url="http://<cdn_url>/assets/cards/office_item.json"
```

```
                        context="{{item}}"
                    >
                    </ubisoft-adaptive-card>
                </div>
            </div>

        </div>
    </template>
</ubisoft-search-results>
```

For the default item rendering, the only difference is the icon displayed according to the item file extension. Instead of creating multiple conditions for each value, we group file types by families computing a dynamic `filefamily` property in the search component itself. This way the logic to display the correct icon becomes quite simple:

```
$schema": "http://adaptivecards.io/schemas/adaptive-card.json",
    "type": "AdaptiveCard",
    "version": "1.3",
    "body": [{
        "type": "Container",
        "items": [
            {
                "type": "ColumnSet",
                "id": "${hitId}",
                "columns": [
                    {
                        "type": "Column",
                        "width": "24px",
                        "verticalContentAlignment": "center",
                        "spacing": "none",
                        "bleed": true,
                        "items": [
                            {
                                "type": "Image",
                                "horizontalAlignment": "left",
                                "url": "https://<cdn_url>/assets/icons/${if(filefamily,filefamily,'generic')}.svg"
                            }
                        ]
                    },
                    // Trimmed for brevity ...
```

And the file families we defined:

File family	*File extensions*
word	”doc”,”docx”,”docm”,”dot”,”dotx”,”dotm”
excel	”xls”,”xlsx”,”csv”,”xlsm”,”xlsb”,”xlx”,”xml”,”csv”,”xltm”,”xlt”,”xltx”
powerpoint	”ppt”,”pptx”,”pptm”,”pps”,”ppsm”,”ppsx”,”potx”,”potm”,”pot”
onenote	”one”
text	”txt”,”rtf”
visio	”vsd”,”vsdx”,”vsdm”
webpage	”aspx”,”html”
pdf	”pdf”
archive	”zip”,”7z”,”rar”
video	”mp4”,”avi”,”mov”,”flv”,”wmv”,”webm”,”ogg”

Styling adaptive cards

If you already know and use adaptive cards, you know that you cannot really apply CSS on them as styles are managed by the host application. The common way to "style" an adaptive card is to provide an host configuration. For some UI requirements, we did not have the choice to add custom CSS rules on the `<ubisoft-adaptive-card>` based on element ids like this:

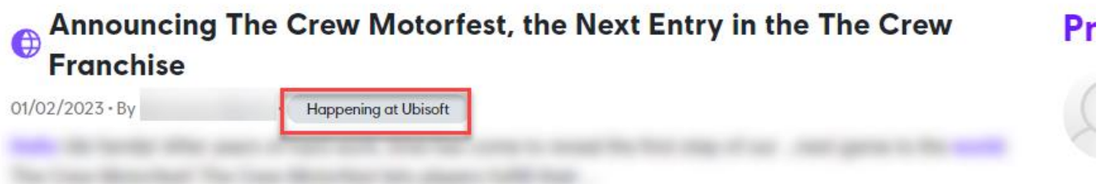
1. **Components embedded styles**

```
static override get styles() {
    return [
        css`

            ...

            .ac-container#ubiSearchSite > .ac-textBlock > p > a {
                padding-left: 16px!important;
                padding-right: 16px!important;
                padding-top: 4px!important;
                padding-bottom: 4px!important;
                border-radius: 25px;
                background-color: var(--topic-background-color) !important;
                text-decoration: none;
                color: var(--ubi365-internal-textColor) !important;
            }

            ...
        `,
        ...
    ];
```

2. **Styled Adaptive card.**

```
...
{
    "type": "Column",
    "verticalContentAlignment": "center",
    "items": [
        {
            "type": "Container",
            "id": "ubiSearchSite",
            "items": [
                {
                    ...
                }
            ]
        }
    ],
    "width": "auto",
    "bleed": false,
    "spacing": "none",
    "$when": "${not(empty(siteTitle))}"
}
...
```

3. **Results**



We do have a shared host configuration, but it only applies on components.

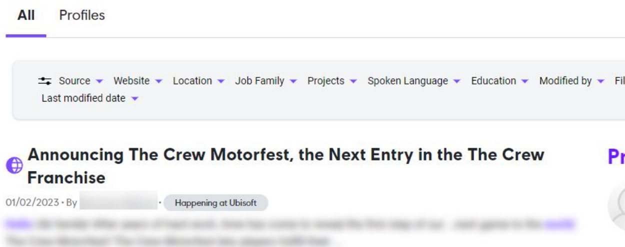
Manage experience between custom/OOTB UI

Because adaptive cards are meant to be displayed in any application the same way, including default Microsoft Search experiences (office.com, bing.com, sharepoint.com), in reusable search web components, we add a special field `iscustomapp` to each results item at runtime in the search results components. This way, in the adaptive card, we can 'detect' if the result type adaptive card is rendered from a custom or a native application. Using the `$when` directive, we can easily decide the rendering experience for both scenarios:

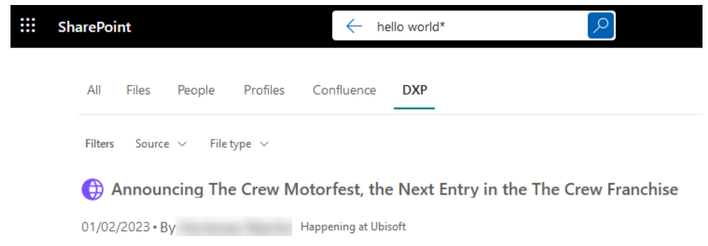
```
"items": [

    // Will be rendered in custom app
    {
        "type": "TextBlock",
        "weight": "bolder",
        "id": "trackEvent",
        "spacing": "small",
        "text": "<div id='eventData' category='SearchResultsEvents' action='SearchResultItemClicked' ...>[${MSSearchTitle}](<${MSSearchUrl}>)</div>",
        "size": "large",
        "$when": "${equals(iscustomapp,'true')}"
    },

    // Will be rendered only on OOTB UI
    {
        "type": "TextBlock",
        "weight": "bolder",
        "id": "trackEvent",
        "spacing": "small",
        "text": "[${MSSearchTitle}](<${MSSearchUrl}>)",
        "size": "large",
        "$when": "${empty(iscustomapp)}"
    }
]
```

Here is an example of the result for a WordPress item in custom/native experience based on these conditions:

Custom application



Default experience



In the native UI, the following features can't be replicated:

- Event tracking
- Fine-tuned styling
- Web components integration (EGG, MGT, etc.)

Video formats support

In the Ubisoft Microsoft 365 tenant, we have multiple videos with formats like *.avi*, *.flv* or *.wmv* making it impossible to read them directly through the default Adaptive Card "Media" element and using the raw video path. Only web suitable formats will work there like *mp4* or *webm*.

In this case, it means we need to rely on Microsoft Stream player. When you upload a video in SharePoint through Microsoft Stream, regardless of the input format, somehow Stream will re-encode the video to be able to watch it online through its built-in player. This player can then be embedded in an HTML page to play the video and therefore support all videos formats.

Authentication challenges with videos

Because videos are stored in SharePoint through Microsoft Stream but components are used outside of the SharePoint domain, we need an authenticated link to play the video. Using the item path directly into a player will result in an "*Access denied*" error if the user is not already authenticated in the sharepoint.com domain from where the components are displayed.

With Microsoft Graph and the `/preview` endpoint, we are able to get a preview link with embedded access token to play the video. Example:

```
https://tenant.sharepoint.com/sites/mysite/_layouts/15/embed.aspx?uniqueId=4ca086e8-59ae-4c97-9b60-1f
3c371c031f&access_token=eyJ0eXAiOiJKV1QiLC<…token>
```

By default, the preview URL information is not included in results and needs to be fetched subsequently once search results are retrieved for the first time. To request such information, we've added special properties `previewUrl`,`NormSiteID`,`NormListID`,`NormUniqueID` in the `fields` attributed to our `<ubisoft-search-results>` component. Behind the scenes, preview information is fetched in batch for all results in the page so they can be used in a video player component.

```
<ubisoft-search-results
    entity-types="listItem"
    fields="previewUrl,NormSiteID,NormListID,NormUniqueID,name,title,weburl,..."
    query-text="*"
    ...
```

Then, the field will be available in the adaptive card/MGT template context and can be passed to our video player component:

```
{
    "type": "Column",
    "items": [
        {
            "type": "TextBlock",
            "text": "<ubisoft-video-wrapper class='rounded-lg' thumbnail-url='${thumbnailUrl}' preview-
url='${previewUrl}'></ubisoft-video-wrapper>"

        }
    ]
}
```

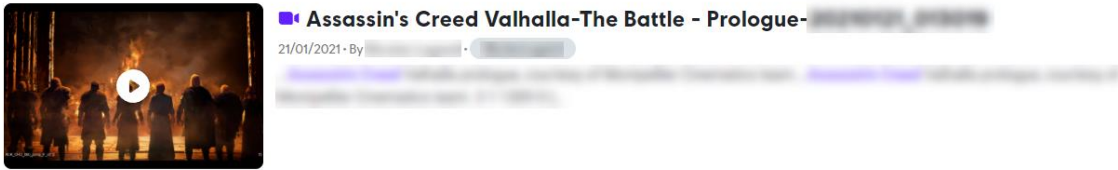
The access token included in the preview link expires in few minutes. In this case, the user will need to refresh the page or switch verticals to obtain a new link.

This is the same behavior with video thumbnails. In this case we are using the thumbnails endpoint to get the video thumbnail and the special fields `thumbnailUrl`,`NormSiteID`,`NormListID`,`NormUniqueID` in the `fields` attribute.

Manage performances for videos.

Because we need to use Microsoft Stream built-in HTML player to support all video formats, we need to integrate it as an *iframe* for every search item in the interface. However, loading an *iframe* for each item at the same time would decrease performance and increase load time. That is why we created a wrapper component `<ubisoft-video-wrapper>` that helps to manage performances using the following behavior:

- Display the video thumbnail through the already fetched thumbnailUrl property with a fake 'Play' button.
- When the play button is the clicked, added the *iframe* dynamically with *autoplay* mode.



Video item

This way, iframes are loaded on demand and do not impact performance at load.

The downside of this approach is the browser may block the video autoplay feature resulting in another play button, this one from the native Microsoft Stream. We needed to add a global organizational policy on browsers to allow autoplay for these specific applications.

Web components testing

The advantage of the web components approach is we can test them easily with Playwright and Open Web Components testing package. We run around 60 automated tests in our CI/CD process to ensure all components behave correctly. Here is an example of a test:

```
...
it("should render custom template for items if provided", async () => {

    const el: UbisoftSearchResultsComponent = await fixture(html`
        <ubisoft-search-results .defaultQueryText=${"*"}>
            <template data-type="items">
                <ul>
                    <li data-for='item in items' data-ref="item">
                        <a href="{{item.resource.fields.defaultEncodingURL}}">
                            <span class="font-bold">{{item.resource.fields.title}}</span>
                        </a>
                    </li>
                </ul>
            </template>
        </ubisoft-search-results>
    `);

    await stubSearchResults(el);

    assert.equal(getTemplateResultsItems(el).length, 5);
});
...
```

```
Chromium: |                              | 11/11 test files | 60 passed, 0 failed
Firefox:  |                              | 11/11 test files | 60 passed, 0 failed

Finished running tests in 150.5s, all tests passed! 🎉
```

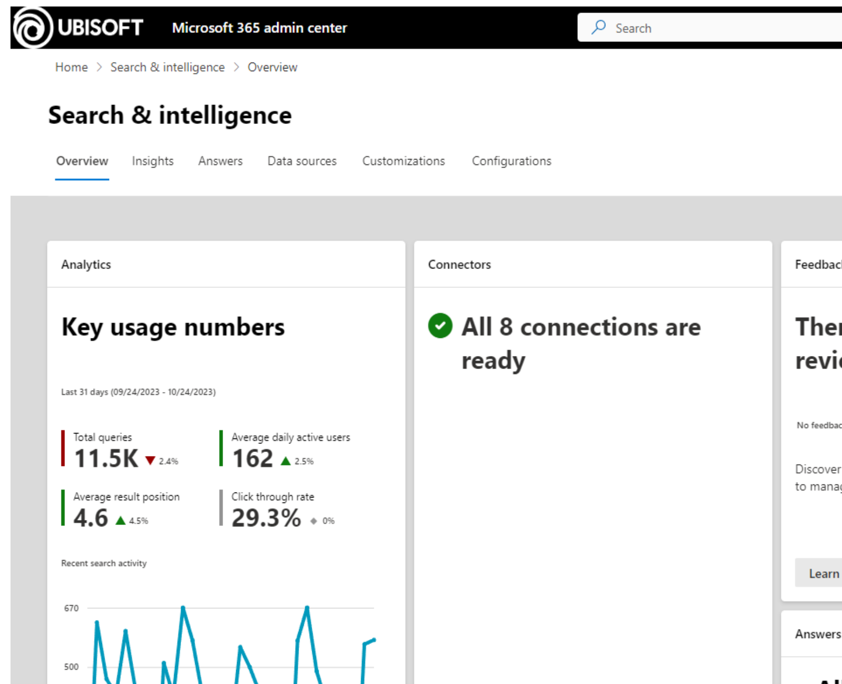
Web test runner

Once deployed, we also have a test team to perform manual tests focused on the overall search UI experience and data:

Search QA

How do we handle analytics?

A big downside of using a custom interface is you do not benefit from the default analytics and relevancy model improvements from the default Microsoft Search experiences. The reason is because analytics are not considered when using the Microsoft Graph API (same for SharePoint REST API). Metrics just won't show up in the search admin dashboard. That is an issue because you can completely miss all the search usage in your tenant if people use custom solutions, like the PnP Modern Search for instance. In the default search interface, some user events are registered by Microsoft that can improve relevancy over time. In our case, the search dashboard is quite empty, and we do not rely on it to measure adoption:



Default dashboard

Luckily, in our case, we wanted to integrate with our own analytics system we are using for our other applications: Matomo. With this application, we have total control over what we track and when. As an example, these are some events we track in the global search application:

Actions

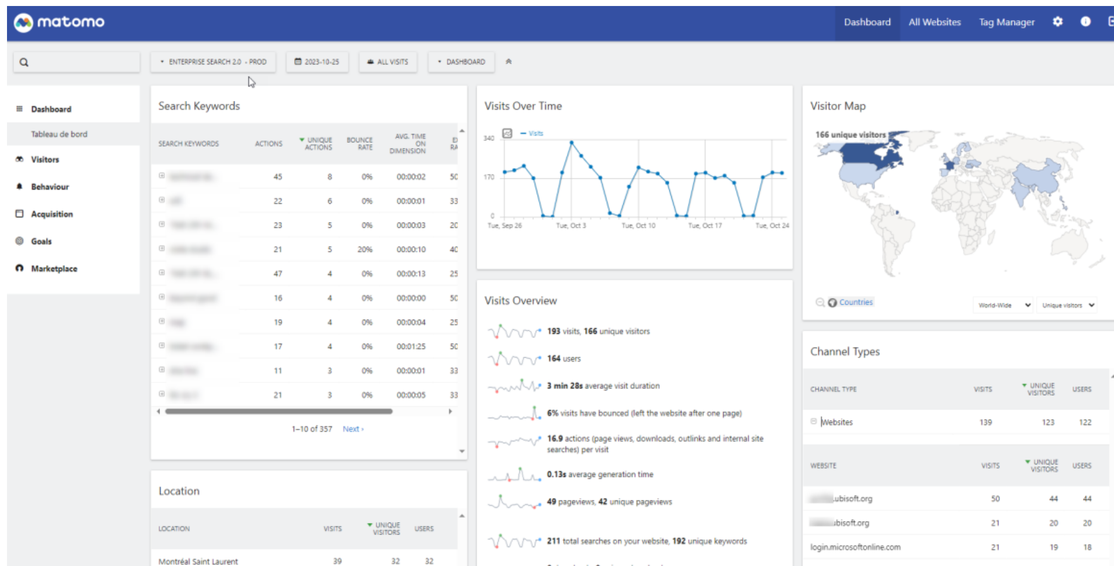
- When a result item is clicked
- When a page is browsed
- When a bookmark is clicked
- When a suggestion is clicked
- When a new keyword is submitted
- When a vertical is selected
- When a filter value is applied
- etc.

Context

- Browser used.
- Device used.

- User location, job family, department
- etc.

All the events are registered anonymously, and users are assigned a unique ID for data privacy purposes.



Matomo analytics

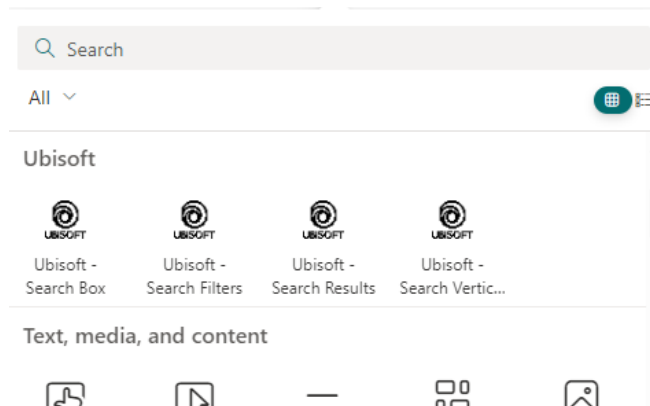
Track events in adaptive card

To track events in adaptive cards, we use the special **trackEvent** id in a TextBlock item. Then we use a div element with the id `eventData` and event attributes like this to pass context data:

```
{
    "type": "TextBlock",
    "weight": "bolder",
    "id": "trackEvent",
    "spacing": "small",
    "text": "<div id='eventData' category='SearchResultsEvents' action='SearchResultItemClicked' value='${rank}'
dimensions='${jsonStringify([{key:'ViewedResults',value:path},{key:'SearchDataSources',value:
if(contentSource,contentSource,'spo')},{key:'SearchItemRanks',value:rank},{key:'SearchContentType',value:
if(filefamily,filefamily,fileType)}])}'>YOUR TEXT</div>",
    "size": "large"
}
```

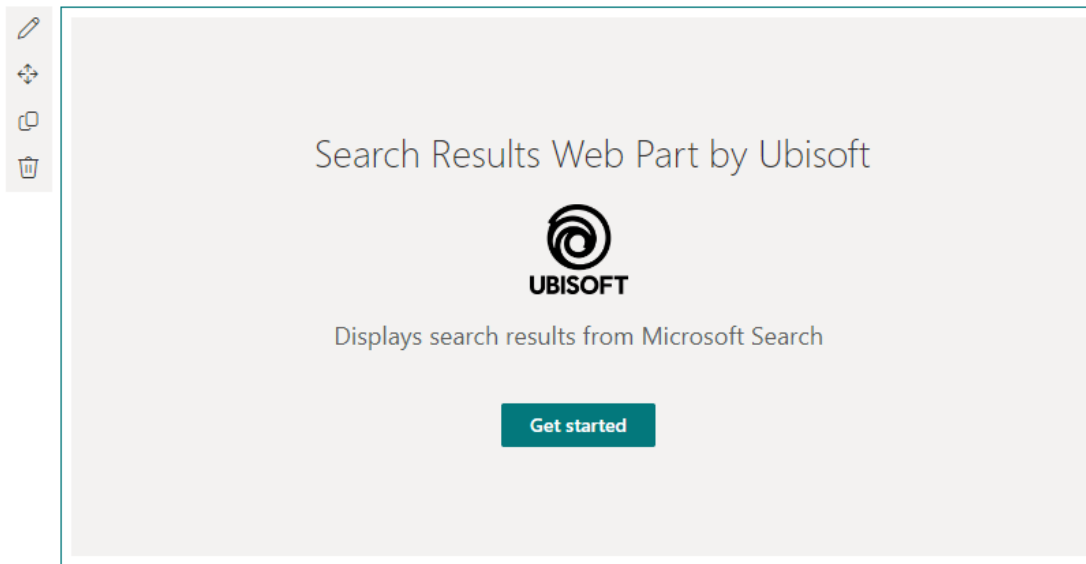
Special case of SharePoint integration

According to our "search as a service approach", SharePoint is considered as a consumer application like any other. However, the particularity is we cannot add web components directly on pages, we need to use the SharePoint Framework. Instead of doing a basic integration, we have taken the opportunity to build a complete end-user-oriented Web Parts set with all configurable options mapping the component capabilities.



Ubisoft Web Parts

This initiative was also to replace gradually the use of the PnP Modern Search in our Microsoft 365 tenant by providing the same Web Parts but tailored for Ubisoft needs:



Ubisoft Search Results Web Part

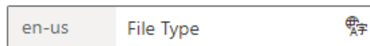
Add new filter

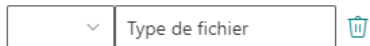
Basic Aggregations

Filter name *

FileType

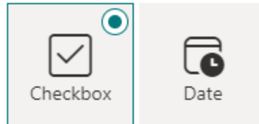
Display name *





Add new translation

Template



Show count

Operator

AND

OR

Is multi value

Sort by

By count

By name

Sort direction

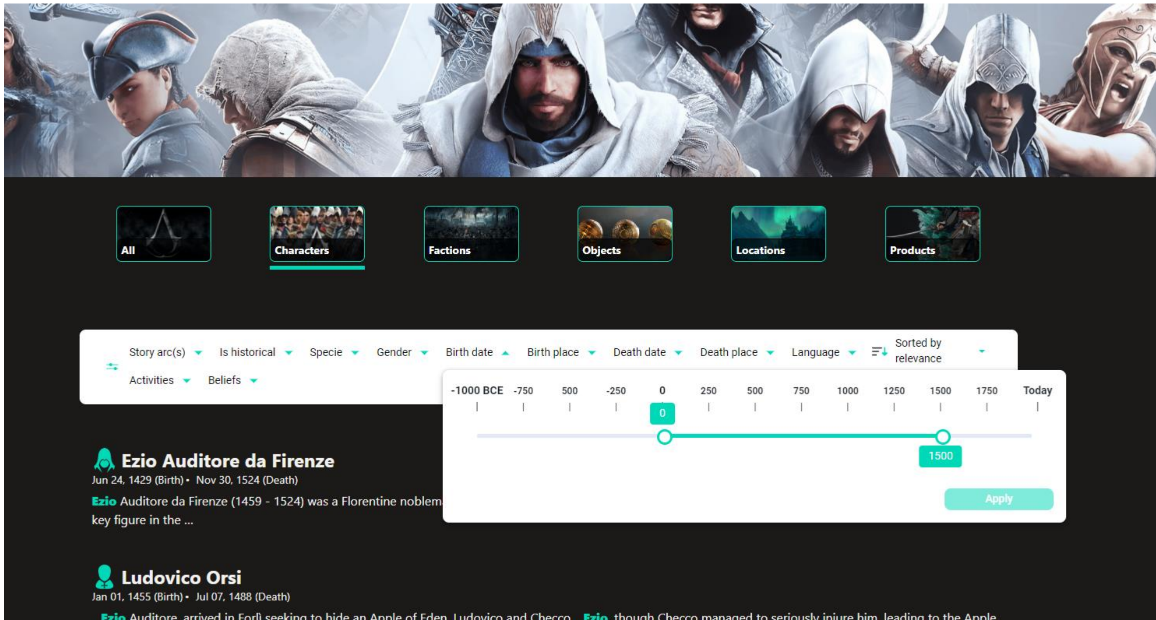
Ascending

Descending

Number of values

Ubisoft Search Filters Web Part

Here is an example the components integration for our internal Assassin's Creed lore documentation using these WebParts:



AC Bible integration

Behind the scenes, we use the same exact web components as any other applications.

WHAT WAS OUR SEARCH DATA STRATEGY?

Like mentioned earlier, we had many sources to index into Microsoft Search, many of them not covered by an "Out-of-the-box" Microsoft Search connectors:

- Local intranet sites (WordPress)
- Ubisoft's Wikipedia Site (MongoDB)
- Ubisoft People Profiles (SQL database) (for this one, we had a limitation on number of possible refiners at the time, so we chose to develop our own custom one)

Search schema definition

Before starting the implementation of any connectors, it is important to define how source properties will be used in the end-users search experience and define their attributes regarding the Microsoft Search schema (*Content*, *Queryable*, *Retrievable*, *Refinable*, *Searchable*, *Semantic Labels*). This step is important as it will save you precious time during implementation. For instance, for external connectors, search schema can't be changed once published so you will have to recreate the connection from scratch even for a small update!

⇨ **What properties will be matched against a "free text" search (aka Searchable attribute)?**

By default, you'd be tempted to put more properties as really needed because you want to be sure not to miss any item. However, the more searchable properties you define, the more noise you'll have in your results, especially in an interleaved results feed combining multiple sources. The goal here is to really stick with important properties at first. Good candidates for searchable properties are title, description, or "tags" fields. Remember searchable properties are here to generate the first set of results for the user when he enters initial free text keywords. Then, they can be complemented with refinable properties (filters) to narrow down the result on more precise values (like dates).

⇨ **What properties will be used as filters (aka Refinable attribute)?**

Within this category, likely fall all date properties (created date, modified date, etc.), multi-valued properties (choice, collection, etc.) or properties with values that can appear in many other items (like a tag) (there is no point to set a title as a refiner or a free text user input :D). From a technical point of view, there is a hard limit regarding the size of such properties during crawl: **4k elements** in a collection and **5k characters** in total, beyond this limit, the value we be truncated in the response and therefore in the UI.

Refinable properties are used in the `aggregations` field in the Microsoft search Graph API request. By default, the API only returns the 50 first refiner values. To get more, you need to configure the `size` parameter.

You **can't** have a property **Refinable** and **Searchable** at the same time in the schema.

⇨ **What properties will be displayed in results (aka Retrievable attribute)?**

This attribute is useful when displaying or using a property value as a condition in result templates (like an adaptive card). Regarding the API, those properties will likely appear in the `fields` property in the search request.

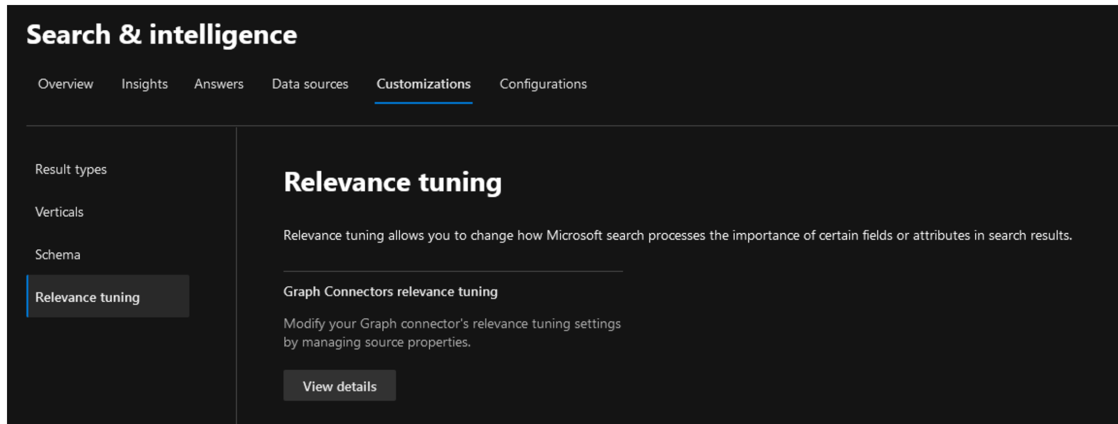
⇨ **What properties can be used in the query using KQL syntax (ex: MyProperty:"My value") (aka Queryable)?**

Usually, users don't use KQL syntax directly in a search box targeting a specific property. This attribute is more useful to define a query template behind the scenes in the results component configuration, for instance restricting on specific documents combined with the user keywords (ex: {searchTerms} FileType:docx FileType:pdf) and have prefiltered results.

⇨ **What properties should be mapped to semantic labels?**

Semantic labels are predefined properties set by Microsoft that influence relevancy. Each property is basically associated to a specific rank weight corresponding to its importance (which we don't know, but we could guess). For example, the title label will likely a 'high' importance by default, so items where keywords matching this property will be higher in the results feed. It is important to set up the semantic labels correctly as it will have a significant impact on the relevancy.

There is also a feature in the search admin portal called **"Relevancy tuning"** but it is only for properties that are NOT already mapped to a semantic label and more importantly, **this only works for OOTB connectors, not custom ones!** For instance, we don't use them in our data sources.



Default vertical strategy

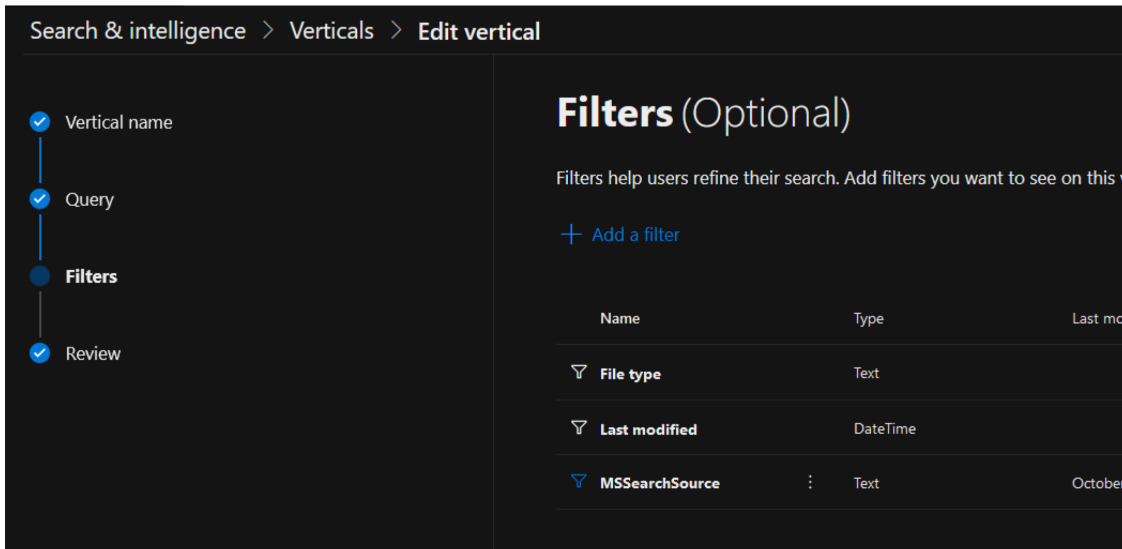
In our case, we didn't use it at all due to the last reason.

⇨ **What sources and filters should be displayed on search verticals?**

Search verticals feature is a convenient way for users to consume results by pre-defined silos, having their own dedicated filters, instead of having all types of content mixed in the same feed including all filters. In general, building a custom search application at organization level integrating with multiple sources will likely require you to plan for search verticals anyway, for at least two reasons:

1. Not all content types can be mixed in the same feed due to API limitation.
2. From a user experience perspective, having results from multiple sources all together will surcharge the user interface with too many filters and results in the feed resulting to a poor search experience (not accurate relevancy, page browsing required, etc.).

In the native Microsoft Search experience, filters and verticals concepts are tied together:



Default vertical strategy

However, with a custom UI implementation strategy, like we took, the approach is quite different as we do not rely on the Microsoft Search verticals configuration (there is no API to retrieve that configuration data anyway).

At this point, you might wonder what would be the relationship between search verticals and the search schema? ➔ **the search unified schema**.

Unified search schema

When we integrated multiple sources in an interleaved results feed for the *'All'* search verticals for the first time, we quickly saw how difficult it will be to manage multiple refiners containing the same information but coming from various sources. That is why we came up with the notion of a unified search schema to simply the configuration: **a schema that all sources need to comply with defining properties with the same meaning used in search.**

⇨ *"Unified search schema" is not a Microsoft concept, you won't find it anywhere in the official documentation.*

For instance, here is our unified schema for our global search:

Alias	Type	Description
MSSearchTitle	Text	Title of an item.
MSSearchUrl	Text	URL of the item.
MSSearchFileType	Text	Corresponds to the file extension of the item (ex: pdf, html, aspx, etc.). This value is used by search components to determine the file family and therefore the correct icon to display for the result.
MSSearchCreated	DateTime	The creation date of the item.
MSSearchModifiedBy	Text	The last users who modified the item.
MSSearchCreatedBy	Text	The user who created the item.
MSSearchLastModifiedTime	DateTime	The last modified date for the item.
MSSearchSiteName	Text	The site name from where the item comes from.
MSSearchSiteUrl	Text	The site URL from where the item comes from.
MSSearchSource	Text	The content source form where the item comes from (ex: 'SharePoint Content', 'WordPress').

Having a unified search schema has multiple benefits:

- We rely on only one filter component instance with a unique filters configuration.
- We can standardize our result item templates (in our case, adaptive cards) and avoid creating duplicates or having conditions inside templates just to support different property names depending on the source. For example, a "SourceTitle" property in a custom source and the default "Title" property for SharePoint content. In the end, they represent the same information.

Practically speaking, for a source to comply to the unified schema, the following actions need to be done:

Custom connectors

For a custom connector, you can simply declare properties according to this schema. We recommend keeping the original property name and set the unified property name as alias:

```
public override Task<GetDataSourceSchemaResponse> GetDataSourceSchema(GetDataSou
{

    Log.Information("Trying to fetch datasource schema");
    SourcePropertyDefinition[] properties =
    {
        new SourcePropertyDefinition()
        {
            Name = WordpressUtility.TitlePropertyName,
            Aliases = { "MSSearchTitle" },
            DefaultSemanticLabels = { SearchPropertyLabel.Title },
            Type = SourcePropertyType.String,
            DefaultSearchAnnotations = (uint)(SearchAnnotations.IsSearchable | S
            RequiredSearchAnnotations = (uint)(SearchAnnotations.IsSearchable),
        },
        new SourcePropertyDefinition()
        {
            Name = WordpressUtility.LinkPropertyName,
            Aliases = { "MSSearchUrl"},
            DefaultSemanticLabels = { SearchPropertyLabel.Url },
            Type = SourcePropertyType.String,
            DefaultSearchAnnotations = (uint)(SearchAnnotations.IsSearchable | S
            RequiredSearchAnnotations = (uint)(SearchAnnotations.IsSearchable),
        },
```

Custom connector properties

Microsoft connectors

For a connector made by Microsoft (ex: Confluence, SQL), you can use property aliases when setting the schema:

Search & intelligence > Data sources > Edit

- Name the connection
- Connection settings
- Select properties
- Manage search permissions
- Assign property labels
- **Manage schema**
- Refresh settings
- Review Connection

Manage schema

Now, you can manage the schema for the results that will be indexed. For a source property to be indexed, at least one of the following must be set: Queryable, Searchable, Retrievable.

Content Property

The content property you select will be used to generate result snippets and descriptions in search results.

content

27 items Search

Source properties	Alias	Type	Labels	Query	Search	Retrieve	Refine
AccessUrl	MSSearchUrl	String	url	✓		✓	
Active		Boolean					
ArticleType	MSSearchFileType	String		✓		✓	✓
Author		StringCollection	authors	✓	✓	✓	

Native connector properties

SharePoint and OneDrive

For SharePoint and OneDrive content, you must define your <u>schema at tenant level</u> reusing the "`RefinableXXX`" properties using aliases:

Property Name	Type	Multi	Query	Search	Retrieve	Refine	Sort	Safe	Mapped Crawled Properties	Aliases
RefinableString00	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	OWS_SITENAME	MSSEARCHSITENAME
RefinableString01	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	OWS_SPSITEURL	MSSEARCHSITEURL
RefinableString02	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	OWS_ENCODEDABSURL, OWS_VIDEOSETDEFAULTENCODING	MSSEARCHURL
RefinableString03	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	Office:8, LASTMODIFIEDBY, LASTMODIFIER	MSSEARCHMODIFIEDBY
RefinableString04	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	FILETYPE, OWS_FILE_X0020_TYPE	MSSEARCHFILETYPE
RefinableString05	Text	Multi	Query	-	Retrieve	Refine	Sort	Safe	Basic:10, Office:2, Mail:5	MSSEARCHTITLE

SPO Tenant schema

Be careful, if some sites override the search schema at site or site collection level using the same properties it will prevail over tenant defined ones. That is why using aliases is important here. This can lead to inconsistent values on the search experience (filters or item metadata).

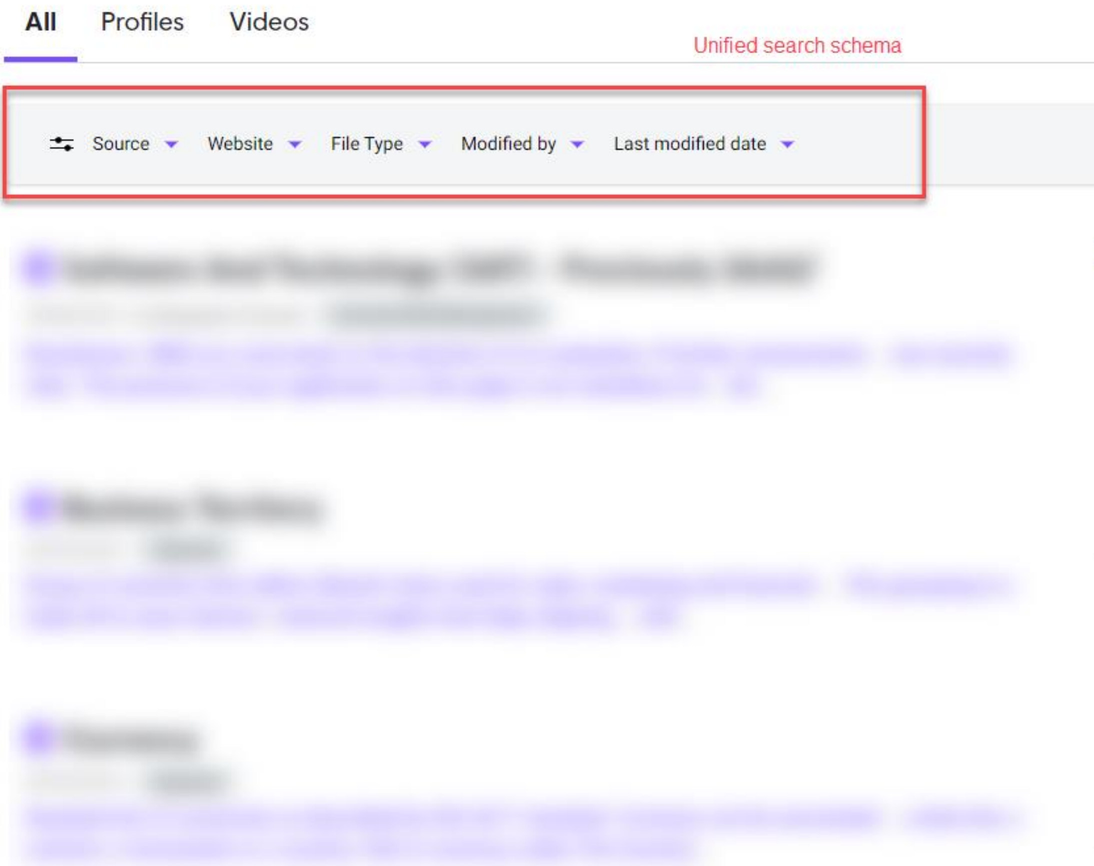
For a unified property, we recommend adding fallback aliases targeting the default SharePoint schema when applicable. This is to cover the scenario when aliases for SharePoint managed properties return no value for whatever reason (we had this issue with `LastModifiedTime`). In the end, aliases for unified property looks like this:

Original property name from source	Unified alias	Fallback alias to SharePoint schema
ArticleModifiedDate or *RefinableDate00*	*MSSearchLastModified Time*	*LastModifiedTime*

In our implementation, verticals and filters are not tightly coupled, meaning verticals definition does not embed filters to be displayed. We essentially rely on the search schema to display the correct filters by verticals, playing with results visibility and properties returned by sources as refiners.

In our search interface, verticals component controls visibility of other results components on the page. When a vertical tab is clicked, the correct result component is displayed fetching results for one or more sources. Other results components not matching the vertical key are hidden. Because refiners are deduced from results, only relevant filters are sent and displayed in the filters component (we only use one filters component on the page).

Here is an example with the '**All**' verticals, retrieving results from multiple sources at the same time using interleaving capability of the API:

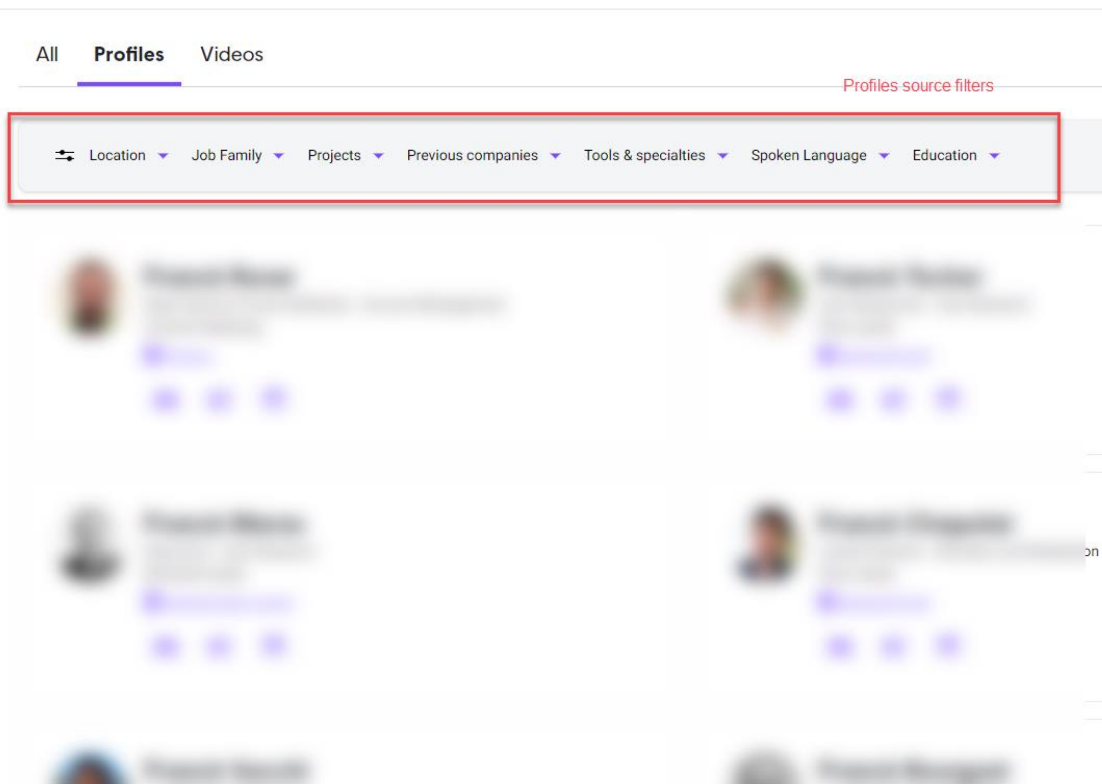


Unified search schema and « All » tab

With our **unified search schema**, we ensure sources return the same properties as refiners so we can aggregate them.

To make it work, we had to add some aggregation logic on the filter's component regarding the Microsoft Search API behavior: **by default, a property with the same alias coming from two different sources will be returned as two separate refiner entries (i.e. aggregation) in the search response**. For users, it means two filters with the same name.

Here is the same example but this time, only retrieving results from our "Profiles" source:

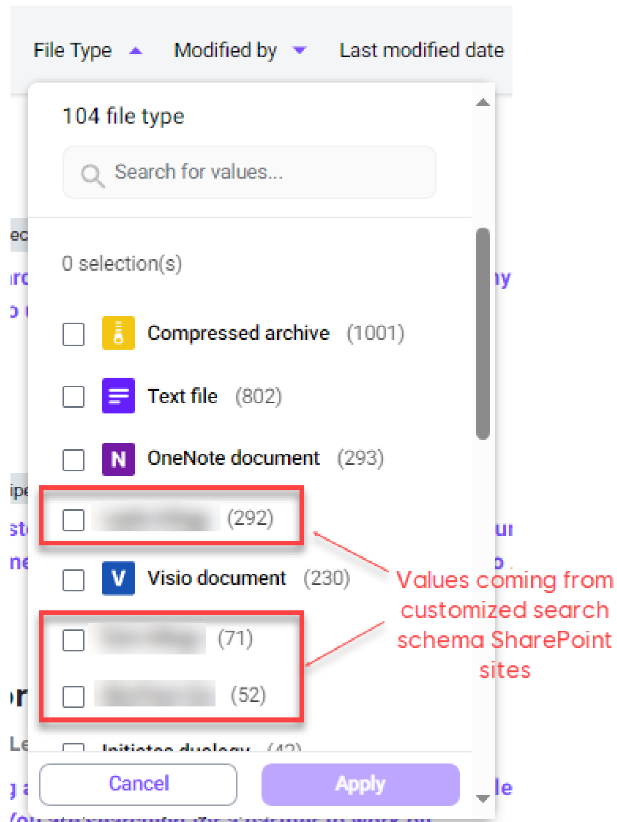


Profiles vertical

In this case and because we display results from only one source, we can use the original specific source properties that don't conflict with the unified schema.

⇨ Case of SharePoint sites with customized search schema

In SharePoint sites, this is not rare to encounter customized search schema, especially if users use the PnP Modern Search solution. If site admins have overridden the search schema with the same `RefinableXX` slot mappings as the unified schema, you may end up with odd filter values in the global UI:



To avoid this situation, we:

- Gradually discourage the use of the PnP Modern Search solution and replace it with our own search WebParts. In the end the PnP Modern Search solution will be removed from our tenant in the near future.
- Do internal communication to site owners about the search behavior and unified schema.

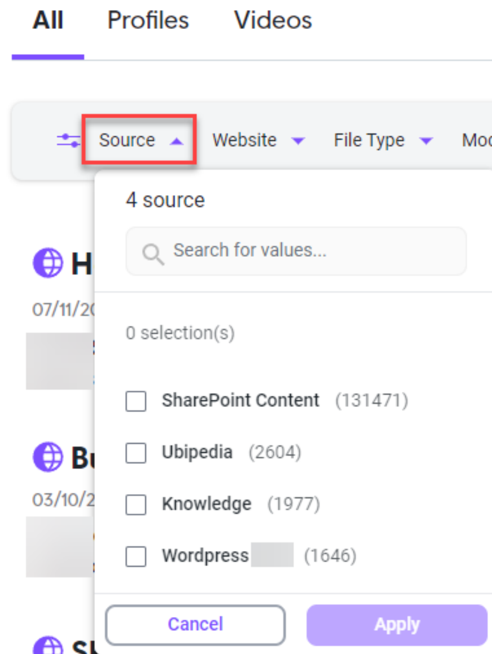
If we notice some unusual values coming from sites, we generally contact the site owner to change the site search mappings and use another range.

Augmenting source metadata

Another strategy we implemented was to add properties that were not available in their original source to have them as refiners.

"Source" property

We have added a "source" property, to let users filter by data source when browsing interleaved results (ex: Confluence, ServiceNow, etc.). By default, there is not such property, even in the default Microsoft Search experience:



Augmented metadata

"Refinable" and "Searchable" for the same property

By default, the search schema, a property cannot be searchable and refinable at the same time. However, in some cases we needed both behaviors for a specific property. For instance, in our "*Profiles*" source we have a "*Job family*" property. For this one, we want users to be able to search against this property but also have it as refiner at the same time. As a workaround, we created two distinct search properties with different attributes but targeting the same source field:

Manage schema

Now, you can manage the schema for the results that will be indexed. For a source property to be indexed, at least one of the following must be set: Queryable, Searchable, Retrievable.

Content Property

The content property you select will be used to generate result snippets and descriptions in search results.

EmployeeName

52 items Search Filter

Source properties	Alias	Type	Labels	Query	Search	Retrieve	Refine
Interests		StringCollection					
JobFamily		String					
JobFamilyRefiner		String					

Augmented refinable

This technique works well for custom built connectors. However, for Microsoft connectors, your only option is to create these additional properties at source when possible.

How did we build our custom data connectors?

Building a custom Graph connector, our experience

At the time we created our connectors, the Microsoft Graph API was already available but not the SDK. Because we were part of the Fastrack program with Microsoft, we had the opportunity to test the SDK in preview and start building our custom connectors and give feedback to Microsoft along the way.

Overall, connector implementation experience is straightforward even for non-advanced C# developers (like me :D). The SDK provides a couple of methods to be implemented by developers, some for the crawling process itself like `GetCrawlStream` and `GetIncrementalCrawlStream` and others for connection management like `GetDataSourceSchema`, `ValidateCustomConfiguration` and `ValidateAuthentication` called when the connection is created in the UI.

Full crawl algorithm

In the end, and regardless the underlying source technology, a full crawl algorithm typically looks like this:

```
OFFSET = 0

ITEMS = GET (PAGESIZE) ITEMS STARTING FROM (OFFSET)

WHILE ITEMS COUNT GREATER THAN 0
|    OFFSET = OFFSET + PAGESIZE
|    ITEMS = GET (PAGESIZE) ITEMS STARTING FROM (OFFSET)
END WHILE
```

⇨ Crawl optimization

Quite honestly, we found no value to put efforts to parallelize or multithread the crawl algorithm to improve performance (does not mean you cannot). In the end, you will be limited by the Graph API.

Incremental crawl algorithm

For an incremental crawl, the difference is about the data source query. Most of the time we rely on a last "modified/updated/created" property that we compare to the last incremental crawl execution date to determine items to retrieve. The logic is pretty much the same as a full crawl:

```
OFFSET = 0

ITEMS = GET (PAGESIZE) ITEMS WHERE LAST MODIFIED DATE GREATER OR EQUAL THAN (LAST_C
RAWL_TIME) AND STARTING FROM (OFFSET)

WHILE ITEMS COUNT GREATER THAN 0
|    OFFSET = OFFSET + PAGESIZE
|    ITEMS = GET (PAGESIZE) ITEMS WHERE LAST MODIFIED DATE GREATER OR EQUAL THAN (L
AST_CRAWL_TIME) AND STARTING FROM (OFFSET)
END WHILE
```

Managing errors during crawls

Managing crawl errors is an important process to ensure failures are reflected in the Microsoft Search admin UI allowing you to take actions and debug easily when needed. If you don't implement the error flow, the connector has no way to know if items have been retrieved correctly or not.

For each item you retrieve from a source, you need to convert it to a `CrawlStreamBit` containing an `OperationStatus` indicating the crawl status for that item (Success, Failed ,etc.) and write into the response stream so it can be indexed by Microsoft Search. In the case of an error, you can also set a `CustomMarkerData` that will be sent on subsequent crawl attempts. A marker data can be anything relevant according to your data source (sequential item ID, start offset, page number, etc.):

```
foreach (var crawlItem in itemsCrawlItems)
{

    CrawlStreamBit crawlStreamBit = this.GetCrawlStreamBit(crawlItem, postsOffset);
    await responseStream.WriteAsync(crawlStreamBit).ConfigureAwait(false);
    i++;
    crawlItemsCount++;
}
...
private CrawlStreamBit GetCrawlStreamBit(CrawlItem crawlItem, int currentOffset)
{
    return new CrawlStreamBit
    {
        Status = new OperationStatus
        {
            Result = OperationResult.Success,
        },
        CrawlItem = crawlItem,
        CrawlProgressMarker = new CrawlCheckpoint
        {
            CustomMarkerData = currentOffset.ToString() // = Current page,
        },
    };
}
```

When you look at the Microsoft connector demo, you notice `CrawlStreamBit` items returned have always `OperationStatus.Success` set regardless of if there was an error or not.

Most of the time, if the crawl encounters an error, it is because the underlying data source remote call failed when retrieving a particular items batch (ex: with a HTTP 500/404/403/etc. if the source uses HTTP endpoint). Because it is uncommon for a data source to implement a "per item" error in the response, you do not really know which item caused the error or if it was even related to a specific item or completely something else.

According to the `CrawlStreamBit` structure, the logic would be to raise the issue at item level with an `OperationStatus.DatasourceError` and the item ID in the associated `CrawlStreamBit` so you can have the exact number of failed items in the Microsoft Search admin portal. However, in practice, this is likely impossible because, in the case of a failed call, item IDs are not known and can't be necessarily deduced (IDs may not be sequential) so we can't set corresponding `CrawlStreamBit`.

That is why in practice, the error flow is handled in the main loop, when iterating through items batches. Considering our pseudo algorithm above the error flow is like this:

```
OFFSET = 0

IF CUSTOMMARKERDATA IS PRESENT
    OFFSET = CUSTOM_MARKER_DATA
END IF

TRY
```

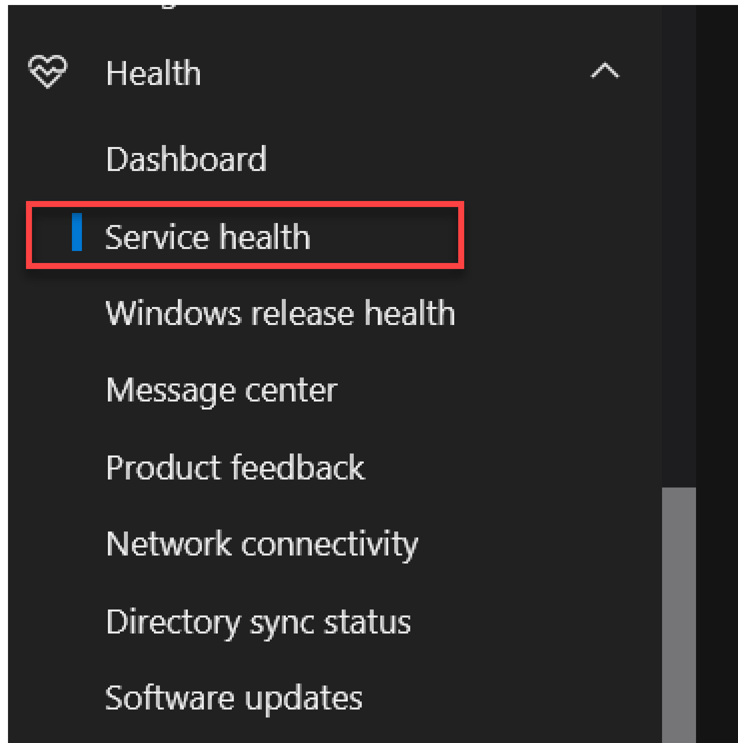
```
        ITEMS = GET (PAGESIZE) ITEMS STARTING FROM (OFFSET)

        WHILE ITEMS COUNT GREATER THAN 0
        |     ITEMS = GET (PAGESIZE) ITEMS STARTING FROM (OFFSET + PAGESIZE)
        |     OFFSET = OFFSET + PAGESIZE
        END WHILE
    CATCH
        CUSTOM_MARKER_DATA = OFFSET
        STATUS MESSAGE = EXCEPTION_DETAILS
        WRITE DATA SOURCE ERROR IN RESPONSE STREAM
```

```
try
{
    ... // Crawl logic here
}
catch (Exception ex)
{
    Log.Error(ex.ToString());
    CrawlStreamBit crawlStreamBit = new CrawlStreamBit
    {
        Status = new OperationStatus
        {
            Result = OperationResult.DatasourceError,
            StatusMessage = $"Fetching items from datasource failed - Details: {ex.ToString()}",
            RetryInfo = new RetryDetails
            {
                Type = RetryDetails.Types.RetryType.Standard,
            },
        },
        CrawlProgressMarker = new CrawlCheckpoint
        {
            CustomMarkerData = postsOffset.ToString()
        },
    };

    await responseStream.WriteAsync(crawlStreamBit).ConfigureAwait(false);
}
```

If something goes wrong, it will be caught by the main loop as a data source error. The custom marker data will be the last offset used to retrieve the last batch and the status message, the exception details, right from the API response call.

Configuring alerts

When a crawl fails, **<u>you are not notified automatically</u>**. You need to configure alerts in the Microsoft 365 portal for that. However, the notifications are sent to Service Health Dashboard where they appear under "*Issues in your environment that require action*" section. An alert is sent to the admin center home page as well.



Service Health Menu

The notification is live in the dashboard for 7 days and post that it moves to "Issue history". You can also subscribe to get these notifications directly in your e-mail:

- Go to "Customize".
- Select "Email" tab.
- Select the appropriate issue type you want to be notified for.
- All Graph connector notifications are rolled up under Microsoft 365 suite, select the checkbox for the same under "Include these services" section.

Customize

Page view Email

- [x] Send me email notifications about service health

Enter up to 2 email addresses, separated by a semicolon

Include these issue types

- [ ] Incidents
- [ ] Advisories
- [x] Issues in your environment that require action

Include these services

- [ ] Azure Information Protection
- [ ] Dynamics 365 Apps
- [ ] Exchange Online
- [ ] Microsoft 365 apps
- [ ] Microsoft 365 Defender
- [ ] Microsoft 365 for the web
- [x] Microsoft 365 suite

Service Health Notification customization

Key tips when building a custom connector

- The main difficulty when building connectors is really about the underlying data source and how to connect to it and query the data. For instance, connecting and querying an SQL database and execute a stored procedure is quite different from connecting and querying a MongoDB database or using the WordPress REST API. If your source supports pagination and has a field to match during an incremental crawl, you are basically good to go.
- The `CrawlStreamBit` structure is misleading as it provides an `OperationStatus` lets you think you can manage errors at item level. Spoiler, you can't. Errors are likely managed at batch level in the main loop. You can set a custom marker data, typically the last successfully crawled item to retry the crawl in case of a failure.
- The `GetCrawlStream` and `GetIncrementalCrawlStream` methods don't return any values. A crawl is considered successful when the methods exited without errors written in the response stream meaning you never explicitly notify the agent app the crawl is actually finished.
- If your data source does not contain a lot of items, you can skip the incremental crawl implementation as crawl will only take a couple of seconds/minutes.
- Don't forget to set aliases and semantic labels on the properties in the `GetDataSourceSchema` methods. They will be prefilled during the connection creation avoiding you to set them manually in the UI.
- After a connection is published, unlike native Microsoft connectors, **you can't update the search schema even if you update the `GetDataSourceSchema` implementation and redeploy your connector**. If you need to update the schema, you will have to delete and recreate the connection. **This is a big concern if your source has many items.**
- You should implement status logs at least in the console (for instance displaying current offset, call made, etc.). This will help you to debug your connector if something goes wrong, even in production.

Data source permissions strategy

To ensure people see only results they have permission to see, at least with "read" access, it is needed to implement an access control list at item level in the connector during crawling process. The concept is quite simple: for a particular item, you need to set the list of identities that can see or not this item in results. In our case, all users use the search application using their organizational Azure AD account, so all permissions need to be resolved as an Azure AD identity, basically a user or a group.

Here is an example of ACL for "*Everyone*" identity using the SDK:

```
private AccessControlEntry GetEveryoneAccessControlEntry(AccessControlEntry.Types.AclAccessType accessLevel)
{
    return new AccessControlEntry
    {
        AccessType = AccessControlEntry.Types.AclAccessType.Grant // Or "Deny",
        Principal = new Principal
        {
            Type = Principal.Types.PrincipalType.Everyone,
            IdentitySource = Principal.Types.IdentitySource.AzureActiveDirectory,
            IdentityType = Principal.Types.IdentityType.AadId,
            Value = "EVERYONE",
        }
    };
}
```

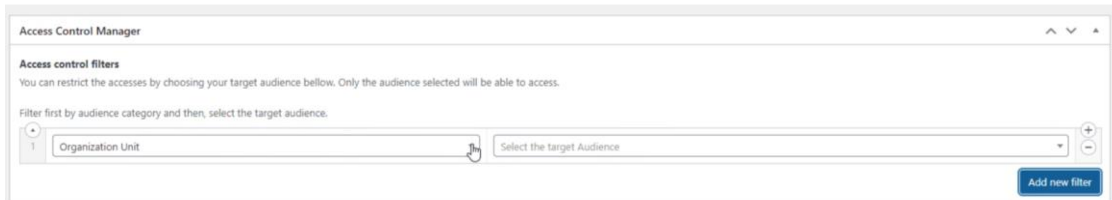
It is one access control entry per identity so having a set of 50 people having access to a particular item means 50 entries in the list.

As long as the data source uses Azure AD and/or relies on user's principal name to manage permissions, and you have a way to retrieve this information (ex: via API), the mapping is done quite easily in the connector via a direct mapping. However, it is preferable to use groups instead of individual users for the below reasons:

- Without groups, if the membership changes at source, it will not be reflected until the next crawl and ACL entries update. Permissions are resolved at query time by Microsoft Search and are always up to date regarding the group memberships. Also, depending on your crawl schedules and incremental crawl conditions (ex: item modified date), some items with new permissions may not be crawled for a long period exposing data to people that are not supposed to have access anymore (even a title or an item summary can be sensitive in search results even if you don't have access to source document anymore).
- Using groups simplifies the ACL mapping and improves crawling performance as it does not require to expand all users for a particular group. You just need to set the group ID in the ACL entry. This way you avoid the hard limit on the API regarding payload size and the number of ACL entries (i.e., users) set for a specific crawl item. Let's say you have a list of 5000 users and 50 0000 items configured with this list, we can quickly see the performance issue here :D.

Manage source custom permissions

Sometimes, sources have their own permissions system not related to Azure AD. This is the case for one of our sources implementing its own permissions mechanism, based on user attributes, and leveraging information from our HR database to construct access conditions (ex: users from a specific location or organizational unit). To restrict/allow access to a particular page, these conditions are matched against the current user information using data from our HR application at **runtime** (i.e., client side).



Custom permission system

To get it to work with Microsoft Search, these permissions must be translated in Azure AD principles like and user or a group. That is why we leveraged Azure AD dynamic groups (requires Azure AD Premium 2) to resolve this particular scenario: for every condition value, we create a dynamic group with a naming convention to be able to retrieve it easily (ex: <attribute_value>-Users-DYN like MTL-Users-DYN for Montréal users).

- Example: Get group with extension value using Microsoft Graph:

```
https://graph.microsoft.com/v1.0/groups?$filter=groupTypes/any(s:s eq 'DynamicMembership') and startswith(displayName,'MTL-Users-DYN')&$top=1&$select=id
```

Whenever a new value is added for a particular condition, for example if a new location is added (ex: "*New York*") for attribute "*Office Location*", a new dynamic group needs to be created based on that value and the extension set to "*NY*".

However, because new values are not added that often, instead of pulling values from a referential (that needs to be created and maintained specifically for that purpose) on a regular basis and compare with existing ones, it is more optimal to detect this case directly during the crawling process.

- New group creation logic

If a group does not exist for a value (like API calls returns no value), it means it must be created. In this case, we create a new ServiceNow ticket automatically from the connector to notify IT admins. For security reasons, we don't handle group creation directly from the crawler.

Next time the item is crawled, if the group exists, the correct ACL will be set. No new ServiceNow request will be sent if one already exists.

Managing connector settings

Credentials

Unless your data source is anonymous, you will probably need to set up credentials to access your data.

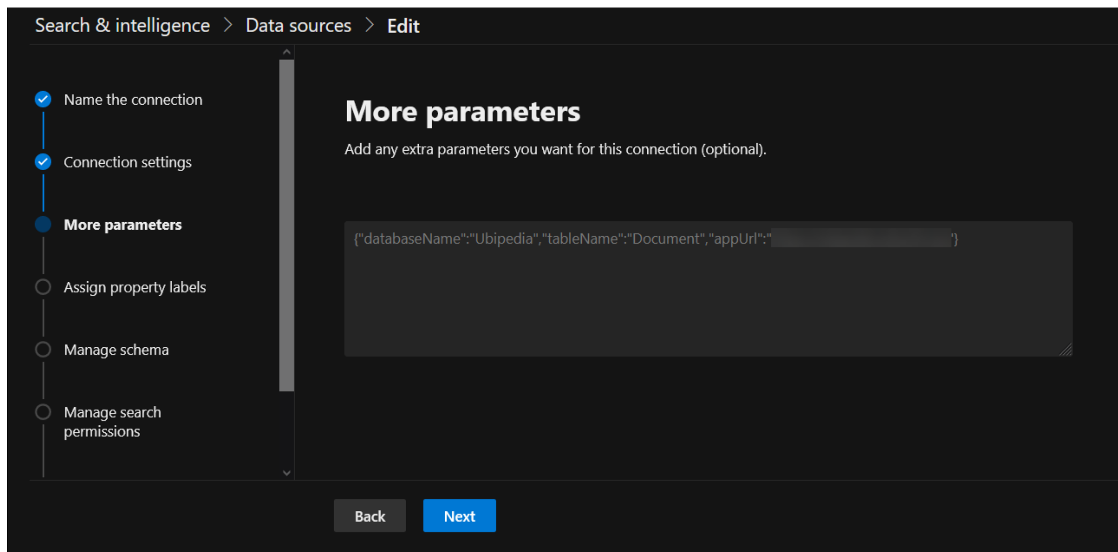
In development mode, credentials are set directly in the `C:\Program Files\Graph connector agent\TestApp\Config\ConnectionInfo.json` file for testing purposes. In production, credentials are passed to your application dynamically when performing crawls according to the credentials you set when you created the connection. **It means, in production, no credentials should be stored directly on the VM!!!**

```
public override async Task GetCrawlStream(
    GetCrawlStreamRequest request,
    IServerStreamWriter<CrawlStreamBit> responseStream
    , ServerCallContext context)
{
    int postsOffset = 0;

    try
    {
        Log.Information($"Crawl start");
        var watch = System.Diagnostics.Stopwatch.StartNew();
        int crawlItemsCount = 0;

        // Use credentials from Microsoft Search connector to connect to MongoDB
        mongoDbService = new MongoDbService(
                request.AuthenticationData.DatasourceUrl,
                request.AuthenticationData.BasicCredential.Username,
                request.AuthenticationData.BasicCredential.Secret);

        // Extract connector custom configuration from Microsoft Search connector
        var providerParameters = JsonConvert.DeserializeObject<ProviderParameters>(
            request.CustomConfiguration.Configuration);
```

Getting connector settings

Connector settings

In addition to credentials, it is usual to have a couple of different settings for a data source depending on the environment, for instance a table name for a SQL instance, an Azure App ID to connect to a remote API using OAuth etc. For these ones, the concept is the same as you have the possibility to pass data as parameter (as text) at connection creation time and retrieve them at runtime. We usually use a stringified JSON object for convenience and for better type checking. You can even validate connection parameters at creation time to comply with a specific schema (method `ValidateCustomConfiguration`).



Connector parameter

⇨ Be careful, if you have data source settings that could change often, it is better to manage them in a separate store location instead of connection parameters (file or database your connector has access to at runtime, preferably well secured). The reason is because **you can't update these settings once the connection is created**. If you need to update settings, you'll have to recreate your connection from scratch…

Test and debug a connector

Test and debug locally

As described here, the local setup is quite easy, you need to configure the `ConnectionInfo.json`, `CustomConnectorPortMap.json`, `Manifest.json` files with all the required settings. However, keep in mind: **this is not because your connector works locally that it will works correctly once deployed**. For instance, the test utility app does not ensure the search schema is correctly defined as it uses a mocked connection. To fully validate a connector, you must test it first on a real Microsoft 365 by creating a new test connection and fully crawl items.

Debug a deployed connector

Sometimes, you will need to debug a connector already deployed. You have multiple ways to do it:

- To **live** debug a connector:
  1. Stop the associated Windows service.
  2. Run the connector `.exe` file.
  3. From the Microsoft Search admin portal, launch a new crawl.
  4. See live console logs (remember the importance to setup logs in your connector code :D)

```
Select \Ubisoft.Connectors.Ubipedia.exe
[01:21:30 INF] Starting Connector Logger
[01:21:30 INF] Starting server ...
[01:21:30 INF] Server started. Listening for calls from Graph connectors platform.
[01:21:30 INF] ConnectorId: 38D85D41-80CA-40FB-B33C-408152A4BA0F running on port:
[01:22:01 INF] Crawl start
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
[01:22:03 INF]  Indexing item
```

Live debug

- To debug **previous** crawls: inspect logs files as described here. The logs will contain the output from the console basically.

Logs debug

There are basically two categories of errors that can occur for a custom connector:

- Data source issue (ex: a failed call for a particular batch, credentials/permissions issues, etc.). For these ones, refer to this section.
- Agent error, meaning the items have been retrieved correctly but can't but ingested for some reason. In the case, you will see some error codes in the Microsoft Search portal for that connection:

GCA error

The document error codes list can be found in the official documentation. However, to us, they were pretty useless quite honestly. Most of the time, the error code we had from the Microsoft Search admin portal was "*5000: Something went wrong. If this issue continues, contact support.*" at item level. Not so ideal to find the root cause…

In addition to documented codes, here are some undocumented ones that you may find useful:

Error code	Signification
3104	*MultivalueCharacterCountExceeded* (4k elements in a collection; 5k characters)
3102	*Null or empty ACLs.*
3004	*InvalidRequest* (ex: *{"ErrorCode":"UnableToMapProperties"},{"ErrorMessage":"Message: Was unable to map 1 out of 53 source properties., additional properties: {"PropertyNames":"ContentSource"}"},{"ErrorType":"InvalidRequestException"}*).
3100	*UnableToMapProperties.* Most of the time it is because the schema declared at connection level and data sent by the connector mismatches (more or less properties than expected)
3007	*Internal Microsoft API throttling error. You generally don't have to do anything particular. The operation will be retried on next crawl.*

From our personal experience, most of time, the graph connector agent related errors were because of a wrong search schema (ex: data type not compatible, missing property or value, incompatible search attributes)

DevOps with custom connectors

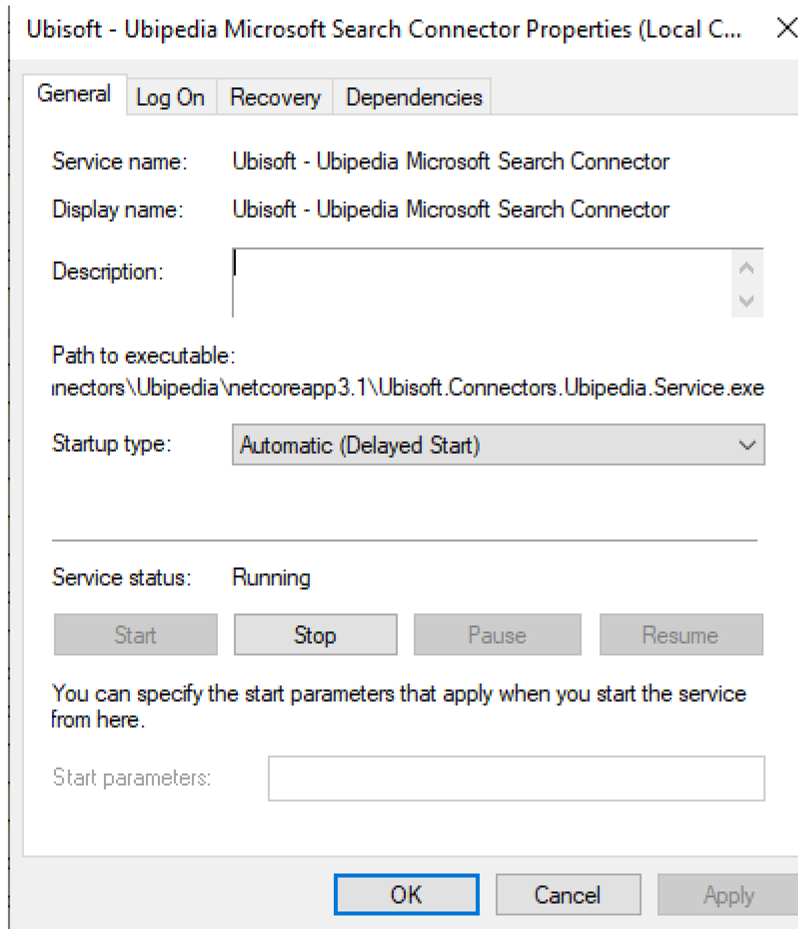
In the end a connector is a simple dotnet console application listening to crawl signals from the Microsoft Connector Agent application. To avoid opening the connector application of the target VM and let it run forever, Microsoft suggests running it as a Windows service. From here, the DevOps process is quite simply and consists to the following steps:

2. Build and version the application in `Release` or `Debug` mode depending on the environment (i.e. `dotnet build -c Release -p:Version=$AssemblySemVer`)
3. Open a remote PowerShell session on the targeted VM.
    - Get the build output and copy files to the target VM.
    - Create or update the Windows service pointing to the `.exe` file of your application.

   For a connector update, you will have to stop the service before updating the settings and restart it after.

At the end of deployment, you get your services up and running and your connector ready to crawl:

Windows services

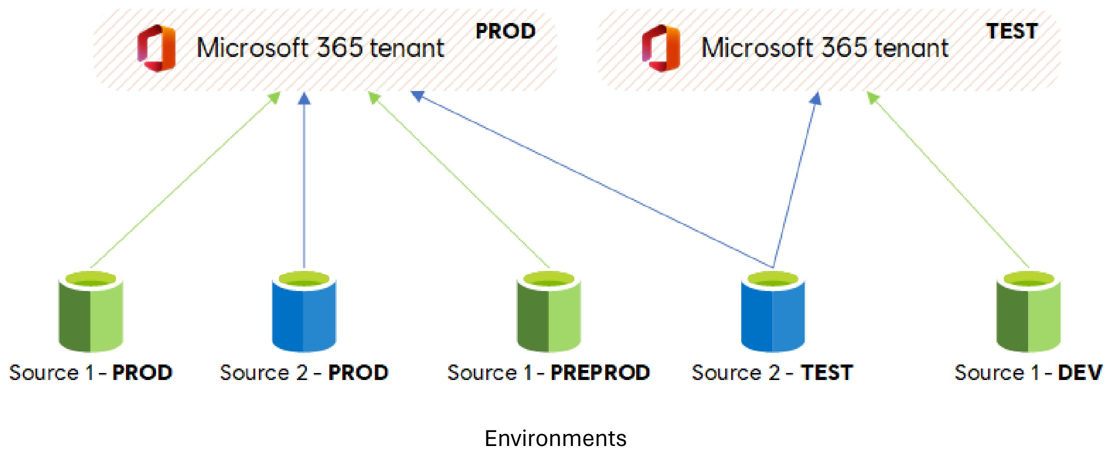


Windows service config

When updating a connector (i.e. the application itself on the VM), you don't need to update the associated connection on the admin portal. Just be careful to not update it during an ongoing crawl :).

Managing environments (tenants/sources)

At Ubisoft, we have two different Microsoft 365 tenants, one for production used by everyday users, and one for integration and test purposes. They are both on different domains and do not communicate with each other. On the other side, we have multiple indexed sources with their own independent environments as well (for example, a DEV, QA, PROD, etc. instances of a database). The mix between the tenant where you index data and source target environments is not necessarily a one-to-one association (TEST tenant only targets TEST database envs). Ideally, the test environments should match together and production ones, the same. However, depending on some technical constraints, like permissions or available source data, you can end-up with a different mix:



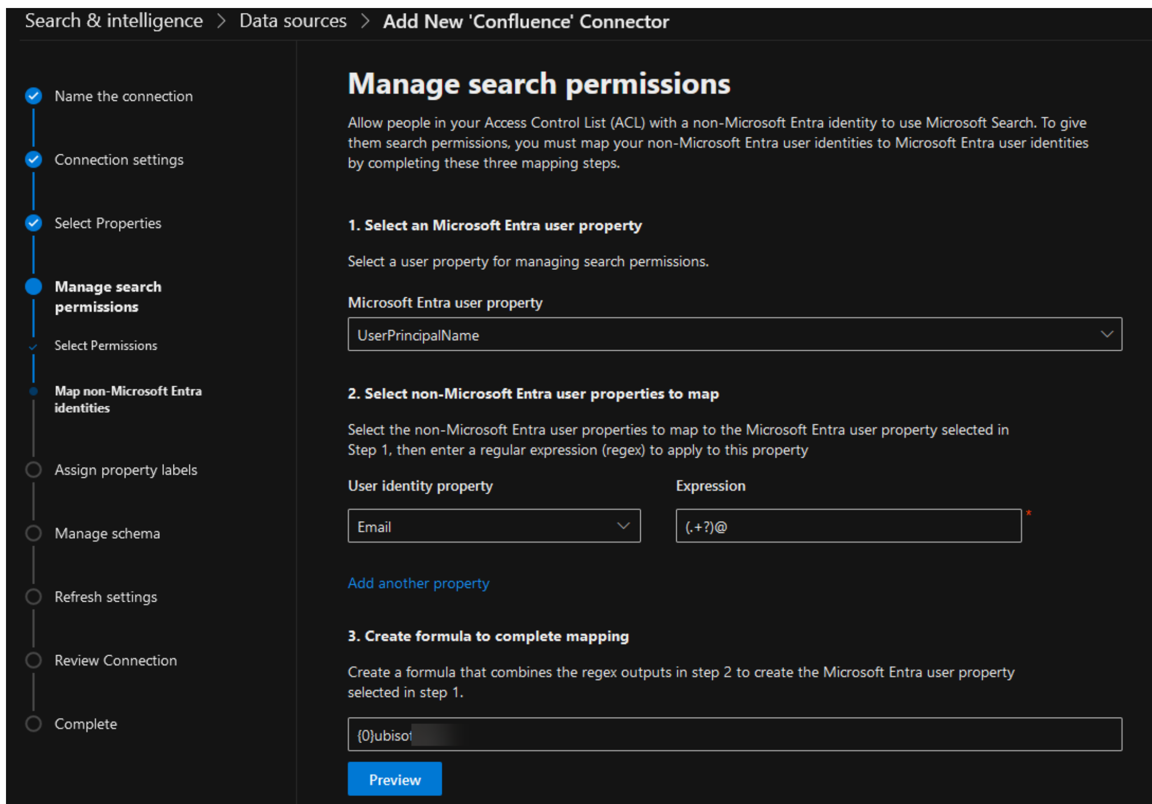
Environments

➩ *Why not use only one environment, like the production one for all sources?*

We reduced the risk of making a mistake, for instance a search schema update that impacts everyone, more possibilities to test new configurations, etc.

Permissions and domain constraint

If the target source is crawled as anonymous, it means it can be integrated seamlessly in any Microsoft tenant regardless of its domain. However, if the source needs to follow the original permissions, then managing environments could be tricky. If source data uses permissions from a different domain where the tenant is indexed, you could not test the search results as no ACL will not match and you will get an empty feed (ex: crawling source with permissions using `@domain1.com` on a tenant domain using `@domain2.com`). To bypass this issue, sometimes, default Microsoft connectors propose some ACL mappings configuration, like the Confluence connector:



ACL mappings

For custom connector, you can implement this kind of feature directly.

Data

Sometimes, this is just the data available on the target is not relevant or missing to properly test the search integration (missing properties for the schema, small amount of data, etc.)

Deploying custom connectors

The development workflow we use for custom built connectors is the following:

Development phase

As connectors require the Graph Connector Agent application, each developer has the application installed on its local machine. Locally, we mainly ensure technical requirements are met:

- Are we able to crawl all the items of the data source (i.e., is it the expected count)? How long does it take roughly?
- Are we able to run incremental crawls? For instance, does it pick latest added items?
- Are the connection parameters correctly validated? If we have dynamic settings or external integration with other systems, are they working correctly?
- etc.

At this step, we don't care much about the search schema.

Deployment Phase(s)

Using our DevOps process, the connector is deployed to a staging VM machine where the GCA agent is installed and setup with a staging Microsoft 365 tenant. In the phase we ensure:

- Search schema works correctly.
- Items are crawled correctly crawled. For this part, we test the results live in our global search applications, that is a mirror of our PROD application:
  - New items are picked up by incremental crawl.
  - Filters are correctly displayed for each vertical according to the search schema.
  - Fields values are correctly set on result templates.
  - Items with searchable properties appear first on results.

Usually, we always use the same well-known set of data to perform tests.

If tests are satisfying, the connector is deployed to a production VM on our production tenant, the connection is created and configured.

⇨ *How many crawling VM machines do you have?*

We first started to use only one VM to host all connections for crawls. It worked fine for a while until we found out some disk space and memory consumption issues for biggest sources (i.e., Confluence) with millions of items to crawl. In such scenarios, a better solution to isolate such sources in a dedicated VM to not compact the others and improve performances.

CONCLUSION AND CHALLENGES

Among all challenges we've encountered along the way, these are the most important:

Managing search quotas (i.e., licenses) between our two Microsoft tenants.

Because we do not have the same number of users and licenses in our Microsoft 365 TEST tenant. For some source it was not possible to index everything. In this scenario, our partners at Microsoft from the FastTrack project helped us by temporarily increasing the quotas while we tested.

Relevancy in the interleaved results feed.

This one was a thought one and is still active as of today (dec. 2023). We currently struggle to get proper relevancy when all results from all sources are interleaved in the "All" search vertical. SPO is prioritized over connected content for everyone, however, for some users, all SPO results are ranked before items from external sources. Overall, with the default Microsoft relevancy model, it is hard to predict the behavior users will get as we don't have a lot of information on how the underlying ranking logic works. We still have some leverages like semantic labels or relevancy tuning, but it our case it didn't significantly improve relevancy despite configuring properly.

Connector debug process is slow and time consuming.

When a connector starts to fail, there is a lot of inertia between the time you usually notice the failure and the time you find the root cause and apply a fix (when possible). This is especially true for big data sources (1M+). For such sources (like Confluence), a full crawl can take days or even weeks. If it fails during the full crawl, you won't be ~~necessarily~~ notified so you should check regularly. Also, the connection error dashboard does not really provide relevant information to be useful. For custom connectors, most of the time, you must inspect the logs file directly on the VM where the GCA agent is installed or ask Microsoft directly about what is going on. See Connectors strategy chapter so see debug process for a custom connector.

For built-in connectors, that can be even more tricky as you have no idea how the crawl is performed behind the scenes and how data is retrieved (was it an error on the connector code itself? your source data? the API?, etc.).

Another issue we had was regarding the connection settings update (ex: search schema, selected properties retrieved, data query, etc.). Most of the time we had to recreate the connection from scratch with new settings as it was not possible to update it directly. Quite frustrating.

Microsoft slightly improved this part as we can now update the search schema for built-in connectors. However, for custom ones, this is still not possible.

Find workarounds for some API limitations!

Due to some API limitations or issues, we had to find workarounds along the way to be able to deliver on time (ex: no sort feature on the "All" tab as it is not supported for interleaved results, aggregating same refiners from multiple sources at runtime because the API does not support it, approximate number of pages leading to empty pages when browsed or blank pages when sources mixed together, etc.).

Fortunately, Microsoft is reactive to fix issues when flagged so overall, we mitigated all the limitations we had so fair regarding the API.

Confluence on-premises indexation

At Ubisoft, we probably have the biggest on-premises Confluence instance in the world. Therefore, this makes any search crawl operation difficult whatever the tool used. To quickly summarize we've had issues with:

- Some plugins cause memory overflow and crashes.
- Confluence APIs have hard limitations.
- Crawl taking forever.
- Disk space consumption on VM.

BONUS: DO THE SAME AS US WITH THE OPEN-SOURCE VERESION OF WEB COMPONENTS

As mentioned in this document, we've built our entire search experience over the Microsoft Graph Toolkit, which is an open-source solution. To share our work and experience with the community, Ubisoft decided to publish these components as an open-source solution as well!

⇨ Welcome to the new PnP Modern Search Core Components accessible there: https://github.com/microsoft-search/pnp-modern-search-core-components

Behind the scenes, we replaced our internal design system "EGG" by the Microsoft FAST components, removed analytics and Ubisoft's specifics. Otherwise, these components are the same as the ones used by all Ubisoft employees in our production environment.