# sources

# Table of Contents

# Indexing content

**Abstract**

Make your content searchable in Sitecore Search by creating sources to index content. Use pull sources (crawlers) or a push source to create index documents.

In Sitecore Search, a *source* is a configuration that defines the content you want to index, making it searchable by visitors.

Indexing is important because Search does not connect to your original content when a visitor makes a search query. Instead, Search looks in indexes and applies its algorithm to identify content that matches the query. Similarly, content recommendations are also based on the indexes.

You can configure a source to access content in different locations and in different formats, such as HTML pages, PDFs, Microsoft Office documents, and JSON objects. Your search implementation can have multiple sources.

We've provided some guidelines to help you choose the best type of source configuration for your content.

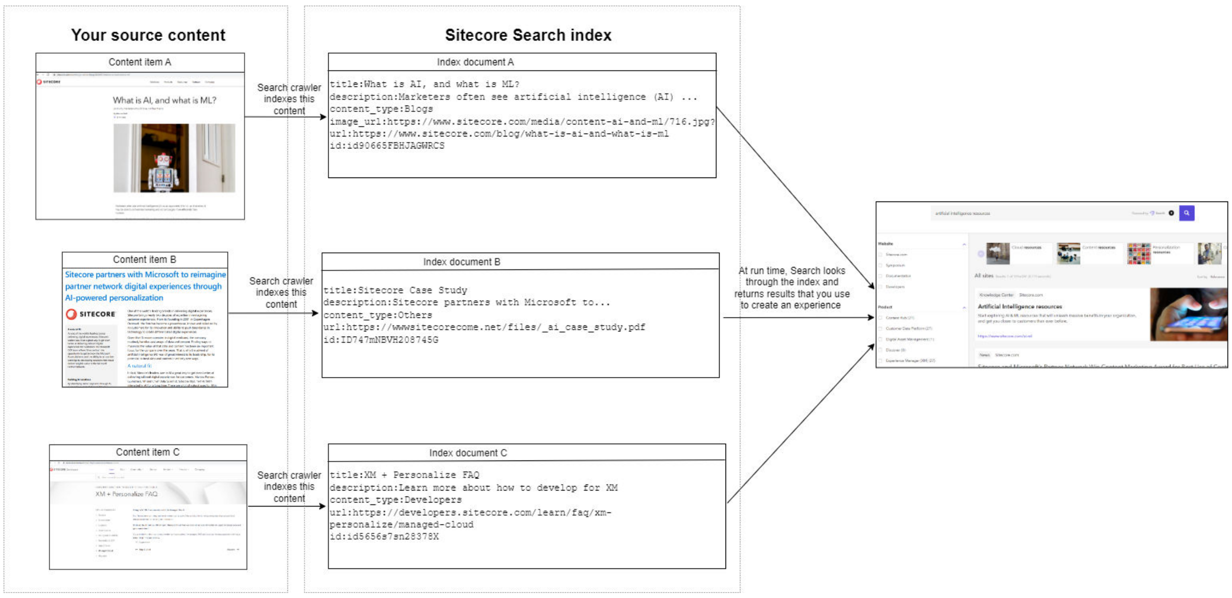
**IMPORTANT**

When you configure sources, remember that you are determining which content Search examines to return results. Consider your entities and attributes and the intended search experiences. Doing this ensures that you can configure the correct number of sources and define the appropriate scope of each source.

## Sources and indexes

You won't directly interact with the indexes or index documents created by Sitecore Search, but understanding what they are and how they work can be useful when configuring a source.

The following diagram illustrates how your original content, indexes, and index documents relate to each other:


Your source content
Content item A
What is AI, and what is ML?
Content item B
Sitecore partners with Microsoft to reimagine partner network digital experiences through AI-powered personalization
Content item C
XM + Personalize FAQ
Search crawler indexes this content
Search crawler indexes this content
Search crawler indexes this content
Sitecore Search index
Index document A
Index document B
Index document C
At run time, Search looks through the index and returns results that you use to create an experience

```
title:What is AI, and what is ML?
description:Marketers often see artificial intelligence (AI) ...
content_type:Blogs
image_url:https://www.sitecore.com/media/content-ai-and-ml/716.jpg?
url:https://www.sitecore.com/blog/what-is-ai-and-what-is-ml
id:id90665FBHJAGWRCS
```

```
title:Sitecore Case Study
description:Sitecore partners with Microsoft to...
content_type:Others
url:https://wwwsitecorecome.net/files/_ai_case_study.pdf
id:ID747mNBVH208745G
```

```
title:XM + Personalize FAQ
description:Learn more about how to develop for XM
content_type:Developers
url:https://developers.sitecore.com/learn/faq/xm-
personalize/managed-cloud
id:id5656s7sn28378X
```

An index is a data structure representing content items that are associated with a specific source. For example, a content item could be a URL, a PDF document, a Microsoft Word document, or a JSON object.

An index document represents a single content item. For each locale, there is a 1:1 mapping between index documents and content items. For example, a 1,000-word HTML page is mapped to one index document, and a 10-page PDF is mapped to one index document.

Every index document is a JSON object, with each property representing the name of a content item's attribute and the associated value.

For example, the index document for your company's *About us* page might have the following structure:

```
{
    "_index": "sampleindexnumber12345",
    "_id": "internalID123",
    "_version": 1,
    "_seq_no": 10,
    "_primary_term": 1,
    "found": true,
    "rfk_source": [
        "content": {
            "id": "id83723bBnjh9321",
            "url": "https://www.sitecore.com/company",
            "image_url": "https://wwwsitecorecom.db.net/images/3d-scene05-violet.png?",
            "content_type": "others",
            "title": "About us",
            "description": "We’re growing at a historic rate. 2,200 employees...",
            "subtitle": "Welcome to Sitecore. We help brands create human connections...
            "last_modified": "2023-09-13 17:54:34"
        },
        "_entity": "content"
    ],
    "questions_and_answers": {...},
    "rfk_stats": {...},
...
}
//sample index document for illustration only
```

The `rfk_source` array is the main part of an index document, containing attributes and their corresponding values.

In addition to the attributes that you configure a source to extract, index documents contain other important information that Search uses to generate results. This includes the internal attribute *rfk_stats*, related questions and answers, and more. You can view some of these details in the **RFK Details** section of content items in the **Content Collection**. Sitecore Search populates these items based on your attribute and feature configuration.

**NOTE**

You can view all your searchable content on the **Content Collection** page of the Customer Engagement Console (CEC). When you index content, it will take a few minutes for it to appear in the content collection.

## Types of sources

Sitecore Search offers two methods for indexing content - pull sources and push sources.

- With a *pull source*, Sitecore Search provides a crawler to scan and index your content, but you define rules that determine which content gets indexed and how to extract attributes. You can think of indexing content using a pull source as a collaborative effort between you and Search.

  **NOTE**
  When you use a pull source, you can also make incremental updates by having a developer use the Ingestion API to add or modify index documents.

- With a *push source*, you create an empty index and then have a developer use an API to create index documents. With this, you have full autonomy over indexing content. You're responsible for creating each index document and populating it with the correct attribute and attribute values.

**NOTE**
For most implementations, you can use pull sources to index the bulk of your content and then use a push source for special cases.

### Types of pull sources

Search offers the following pull sources:

- Web crawler, a basic crawler that crawls your content by starting from a point and following hyperlinks or by going through a sitemap. The web crawler is straightforward, easy to configure, and requires no coding.
- Advanced web crawler, a powerful and highly customizable crawler that supports complex use cases like handling authentication requirements, handling localized content, using JavaScript to extract attribute values, and more.
- API crawler, a crawler specifically designed to crawl API endpoints that return JSON. It supports complex use cases like handling authentication requirements, handling localized content, using JSONPath or JavaScript to extract attribute values, and more.

### Types of push sources

Search offers a single type of push source: the API push source , which creates an index to receive index documents. A developer can use the Ingestion API to push index documents to an index.

# Indexing PDFs

**Abstract**

Describes using Sitecore Search to index PDF content, including things to keep in mind when configuring source settings.

Sitecore Search can index [3] PDFs and have them appear in search results. Similar to when you configure a source to index HTML or JSON content, when you index PDF content, each PDF becomes an index document with attributes like *title*, *description*, and *url*, for example.

Sitecore Search document extractors can only parse HTML or JSON . To effectively extract attribute values from PDF content, you have to understand the HTML structure of your PDFs.

Unlike HTML pages in a browser, you can't directly inspect or view the source of a PDF. However, you can configure a temporary Search source to view the HTML structure of PDFs [8]. This source is called *temporary* because its only purpose is to allow you to view the HTML structure of PDFs. You don't use the index documents from this source to create a search experience.

**IMPORTANT**

To view the HTML structure of a PDF, use a Search source and not an external tool. External converters might give you HTML with syntax variations, which might lead to unexpected attribute values.

After you understand the HTML structure of your PDFs, configure a source to index PDF content.

**NOTE**

We recommend that you use an advanced web crawler source to index PDF content because you usually require the logical power that a JavaScript document extractor provides.

## Document extractors and indexing PDFs

Just like when you index HTML content, you can use any type of document extractor to extract PDF content. However, you usually use a JavaScript document extractor for PDFs because it can support complex use cases like sanitizing text and applying conditions.

When you configure a document extractor for PDF content, keep the following considerations in mind :

- To ensure the extraction rules you define apply only to PDFs and not other content like HTML, define URLs to Match. This ensures the crawler only applies rules to URLs that match a defined pattern. Usually, the GLOB expression `**/*.pdf` suffices because it searches recursively.
- To ensure that all PDFs are marked as such, we recommend that you set the *type* attribute to the fixed value of *pdf*.
- To extract other attributes like *title*, *description*, *URL*, and *parent_url*, use the HTML structure of PDFs you extracted through the temporary source [8].

To help you, we've created some sample [10] document extractors.

## Crawler settings for indexing PDFs

Crawler settings define the scope and working of the crawler. In crawler settings, you define things like how many links to follow from a URL, whether to avoid any URLs, and any header information that needs to be passed to each URL.

When you index PDF documents, we recommend that you review at least the following settings:

- **Max Depth** - if you want the crawler to find PDF URLs by following hyperlinks, ensure that the **MAX DEPTH** is at least *1*. If the **MAX DEPTH** is *0* , the crawler will not follow any hyperlinks, including PDF URLs.

  **NOTE**
  If your trigger is a sitemap or sitemap index that has PDF URLs in it, leave the default **MAX DEPTH** as the default of *0*.

- **Allowed Domains** - if you specify allowed domains, and your PDFs are hosted on a domain that is different from HTML pages, add the domain that contains PDFs. For example, if your HTML pages are on *www.bank.com* but your PDFs are on *wwwbank.hostingservice.net*, add *wwwbank.hostingser-vice.net* as an allowed domain.

## Request extractors and indexing PDFs

Usually, triggers provide the starting points for the crawler. However, if the trigger does not cover PDFs, you will need to use request extractors to create additional requests.

**NOTE**
Configure the **MAX DEPTH** to at least *1* for request extractors to work.

For example, if the trigger is a *Sitemap* with all HTML and PDF URLs, you don't need to configure request extractors. However, if the trigger is a *Sitemap* with only HTML pages, and you have PDF URLs hidden within these HTML pages, create a request extractor to generate PDF URLS for the crawler.

Here's a sample JavaScript request extractor to get only PDF URLs in a page:

```
function extract(request, response) {
  const $ = response.body;
  const regex = /.*\.pdf(?:\?.*)?$/;
  return $('a')
    .toArray()
    .map((a) => $(a).attr('href'))
    .filter((url) => regex.test(url))
    .map((url) => ({ url }));
}
```

## Attributes specific to indexing PDFs

Sitecore Search does not require any additional attributes to index PDFs. However, you can create custom attributes that improve how your search results are organized and presented.

For example, if you want to track the parent page that a PDF belongs to, you can optionally create and publish an attribute called *parent_url*, of the type *string*. You'll then need to configure the document extractor to extract a value [12] for this attribute.

The *parent_url* attribute is useful when PDF URLs look different from the URL of the page they are hosted on. For example, the page *https://www.bank.com/legal/* might contain PDFs with URLs like *https://wwwbank.hostingservice.net/february-2023--global.pdf?md=20230215T151008Z.*, which is in a different domain.

Use the following settings for the *parent_url* attribute:

- **Entity**: **Content**, or any other entity you want.
- **Display Name**: *Parent URL*, or similar.
- **Attribute Name**: *parent_url*, or similar.
- **Placement**: **Standard**
- **Data Type**: **String**

## Walkthrough: Configuring a temporary source to extract HTML from select PDF content

**Abstract**

Configure a temporary source in Sitecore Search to view the HTML structure of PDF documents.

Sitecore Search document extractors can parse only HTML. As a result, you have to know what the HTML structure of your PDFs looks like before you can set up accurate document extractors to extract attributes like *title*, *description*, *tags*, and more.

To do this, configure a temporary source whose only purpose is to reveal the HTML structure of your PDFs. In this source, you select a JavaScript document extractor and use the `html()` jQuery method to extract HTML from the *entire* PDF. Then, after you view the HTML structure of the PDF in the Content Collection, you can decide how best to extract individual attributes.

This walkthrough describes how to:

- Create a dummy attribute to enable HTML extraction
- Create a temporary advanced web crawler source and configure triggers
- Configure a JavaScript document extractor to extract HTML
- View the HTML structure of the PDF in the Content Collection

**NOTICE**

Before you begin

- Gather a few URLs that represent the different types of PDF content you have.

We recommend that you scan your PDFs and look for patterns. For example, you might notice some that are text-heavy with tables, some that are image-heavy with limited text, and some that are a list of questions and answers. You can note down one PDF URL from each group, which gives you three representative PDFs.

Identifying representative PDFs is important. When you later configure a source to extract *all* PDFs, you create document extractors based on the HTML structure of the sample PDFs you got.

## Create a dummy attribute to enable HTML extraction

You need an attribute that you can use to extract the entirety of the PDF document in HTML format. To avoid confusion, we recommend creating an attribute specifically for this purpose.

To create an attribute to enable extraction of the HTML version of the PDF:

Create and publish an attribute with the following details:

- **Entity**: **Content**.
- **Display Name**: *PDF to HTML*, or similar.
- **Attribute Name**: *pdf_to_html*, or similar.
- **Placement**: **Standard**
- **Data Type**: **String**

## Create a temporary advanced web crawler source and configure triggers

Create an advanced web crawler to crawl only the PDFs you selected to represent your PDF content. The only triggers you need are the URLs of the PDFs you identified before you began.

To create an advanced web crawler and configure triggers:

1. Create a source with the **CONNECTOR** type of **Web Crawler (advanced)**. Name the source clearly to identify its purpose, like *Temp PDF to HTML*.
2. Create a request trigger with the **URL** of one of the PDFs you identified in the prerequisite. Leave all other settings as the default.
3. Repeat Step 2 for the other PDF URLs you identified. For example, if you identified 3 sample PDFs, each having a distinct pattern, you'll have 3 *Request* triggers.
4. To stop the crawler from indexing hyperlinked URLs, click **Crawler Settings** > **MAX DEPTH** and change the value to *0*.

## Configure a JavaScript document extractor to extract HTML

Document extractors create index documents [3] from your original content. Each index document has attributes and attribute values. In this example, you only care about the value of the *pdf_to_html* attribute. To extract the HTML structure of a PDF, configure a JavaScript document extractor that uses the `html()` method to extract the value of the *pdf_to_html* attribute.

To configure a JavaScript document extractor to extract HTML:

1. Create a JavaScript document extractor that uses the following function:

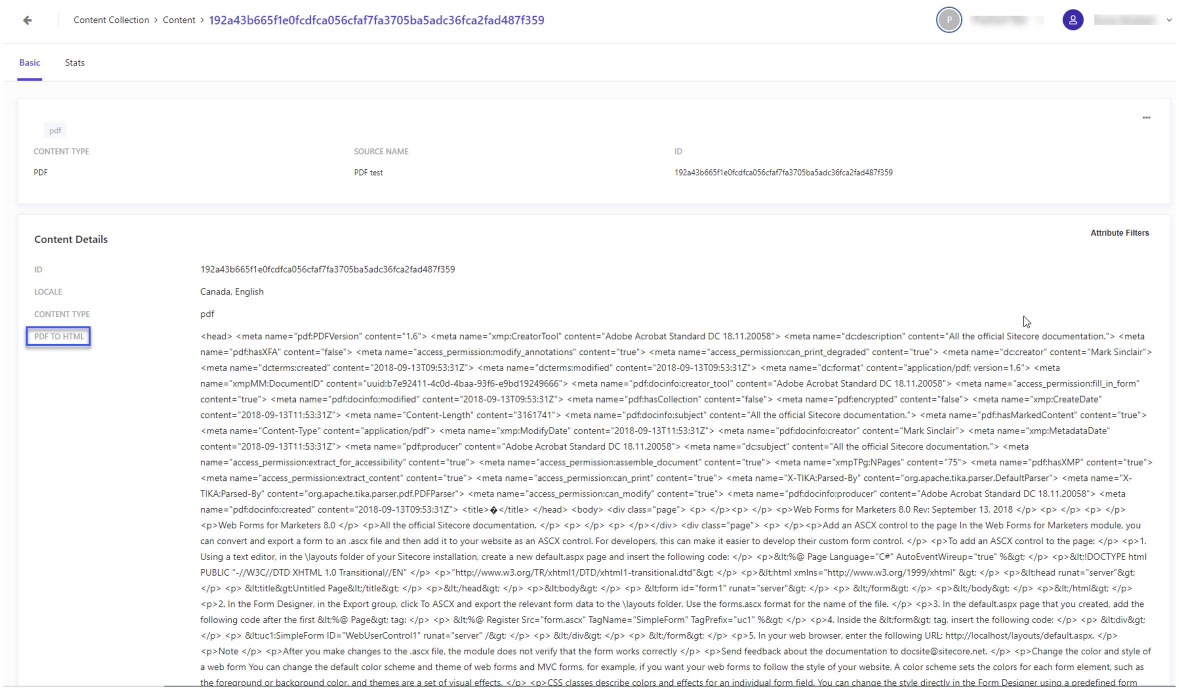
   ```
   // Sample document extractor function to get HTML from PDF content.
   function extract(request, response) {
       $ = response.body;

       return [{
           'pdf_to_html': $('html').html(), // gets the HTML structure of the document's
           'type': "pdf" //Mandatory attribute. Uses the fixed value of 'pdf'
       }];
   }
   ```
2. Publish and scan the source.

## View the HTML structure of the PDF in the Content Collection

Use the Content Collection to find indexed documents and view the content extracted for the *pdf_to_html* attribute.

To view the HTML version of the PDF in the **Content Collection**:

1. In the CEC, click **Content Collection**.
2. Filter by **Sources** and select the source you created previously when you created a temporary advanced web crawler.
   You see a list of content items that represent the PDFs you indexed.
3. Click a content item.
4. In the **Content Details** section, look for the **PDF to HTML** attribute. The value of this attribute is the HTML structure of the PDF.


Content Collection > Content > 192a43b665f1e0fcdfca056cfaf7fa3705ba5adc36fca2fad487f359
Basic
Stats
pdf
Attribute Filters

| CONTENT TYPE | SOURCE NAME | ID |
|---|---|---|
| PDF | PDF test | 192a43b665f1e0fcdfca056cfaf7fa3705ba5adc36fca2fad487f359 |

Content Details

| | |
|---|---|
| ID | 192a43b665f1e0fcdfca056cfaf7fa3705ba5adc36fca2fad487f359 |
| LOCALE | Canada, English |
| CONTENT TYPE | pdf |
| PDF TO HTML | (see below) |

```
<head> <meta name="pdf:PDFVersion" content="1.6"> <meta name="xmp:CreatorTool" content="Adobe Acrobat Standard DC 18.11.20058"> <meta name="dc:description" content="All the official Sitecore documentation."> <meta
name="pdf:hasXFA" content="false"> <meta name="access_permission:modify_annotations" content="true"> <meta name="access_permission:can_print_degraded" content="true"> <meta name="dc:creator" content="Mark Sinclair">
<meta name="dcterms:created" content="2018-09-13T09:53:31Z"> <meta name="dcterms:modified" content="2018-09-13T09:53:31Z"> <meta name="dc:format" content="application/pdf; version=1.6"> <meta
name="xmpMM:DocumentID" content="uuid:b7e92411-4c0d-4baa-93f6-e9bd19249666"> <meta name="pdf:docinfo:creator_tool" content="Adobe Acrobat Standard DC 18.11.20058"> <meta name="access_permission:fill_in_form"
content="true"> <meta name="pdf:docinfo:modified" content="2018-09-13T09:53:31Z"> <meta name="pdf:hasCollection" content="false"> <meta name="pdf:encrypted" content="false"> <meta name="xmp:CreateDate"
content="2018-09-13T11:53:31Z"> <meta name="Content-Length" content="3161741"> <meta name="pdf:docinfo:subject" content="All the official Sitecore documentation."> <meta name="pdf:hasMarkedContent" content="true">
<meta name="Content-Type" content="application/pdf"> <meta name="xmp:ModifyDate" content="2018-09-13T11:53:31Z"> <meta name="pdf:docinfo:creator" content="Mark Sinclair"> <meta name="xmp:MetadataDate"
content="2018-09-13T11:53:31Z"> <meta name="pdf:producer" content="Adobe Acrobat Standard DC 18.11.20058"> <meta name="dc:subject" content="All the official Sitecore documentation."> <meta
name="access_permission:extract_for_accessibility" content="true"> <meta name="access_permission:assemble_document" content="true"> <meta name="xmpTPg:NPages" content="75"> <meta name="pdf:hasXMP" content="true">
<meta name="access_permission:extract_content" content="true"> <meta name="access_permission:can_print" content="true"> <meta name="X-TIKA:Parsed-By" content="org.apache.tika.parser.DefaultParser"> <meta name="X-
TIKA:Parsed-By" content="org.apache.tika.parser.pdf.PDFParser"> <meta name="access_permission:can_modify" content="true"> <meta name="pdf:docinfo:producer" content="Adobe Acrobat Standard DC 18.11.20058"> <meta
name="pdf:docinfo:created" content="2018-09-13T09:53:31Z"> <title>�</title> </head> <body> <div class="page"> <p> </p><p> </p> <p>Web Forms for Marketers 8.0 Rev: September 13, 2018 </p> <p> </p> <p> </p>
<p>Web Forms for Marketers 8.0 </p> <p>All the official Sitecore documentation. </p> <p> </p> <p> </p></div> <div class="page"> <p> </p><p>Add an ASCX control to the page In the Web Forms for Marketers module, you
can convert and export a form to an .ascx file and then add it to your website as an ASCX control. For developers, this can make it easier to develop their custom form control. </p> <p>To add an ASCX control to the page: </p> <p>1.
Using a text editor, in the \layouts folder of your Sitecore installation, create a new default.aspx page and insert the following code: </p> <p>&lt;%@ Page Language="C#" AutoEventWireup="true" %&gt; </p> <p>&lt;!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" </p> <p>"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"&gt; </p> <p>&lt;html xmlns="http://www.w3.org/1999/xhtml" &gt; </p> <p>&lt;head runat="server"&gt;
</p> <p> &lt;title&gt;Untitled Page&lt;/title&gt; </p> <p>&lt;/head&gt; </p> <p>&lt;body&gt; </p> <p> &lt;form id="form1" runat="server"&gt; </p> <p> &lt;/form&gt; </p> <p>&lt;/body&gt; </p> <p>&lt;/html&gt; </p>
<p>2. In the Form Designer, in the Export group, click To ASCX and export the relevant form data to the \layouts folder. Use the forms.ascx format for the name of the file. </p> <p>3. In the default.aspx page that you created, add the
following code after the first &lt;%@ Page&gt; tag: </p> <p> &lt;%@ Register Src="form.ascx" TagName="SimpleForm" TagPrefix="uc1" %&gt; </p> <p>4. Inside the &lt;form&gt; tag, insert the following code: </p> <p> &lt;div&gt;
</p> <p> &lt;uc1:SimpleForm ID="WebUserControl1" runat="server" /&gt; </p> <p> &lt;/div&gt; </p> <p> &lt;/form&gt; </p> <p>5. In your web browser, enter the following URL: http://localhost/layouts/default.aspx. </p>
<p>Note </p> <p>After you make changes to the .ascx file, the module does not verify that the form works correctly </p> <p>Send feedback about the documentation to docsite@sitecore.net. </p> <p>Change the color and style of
a web form You can change the default color scheme and theme of web forms and MVC forms, for example, if you want your web forms to follow the style of your website. A color scheme sets the colors for each form element, such as
the foreground or background color, and themes are a set of visual effects. </p> <p>CSS classes describe colors and effects for an individual form field. You can change the style directly in the Form Designer using a predefined form
```

5. Repeat Steps 4 and 5 for all content items.

Now that you've seen what the HTML structure of your PDFs looks like, you can use this information to create accurate document extractors to extract PDF content.

## Sample document extractors for indexing PDF content

**Abstract**

Examples of simple and complex sample JavaScript document extractors to get attribute values from PDF content.

We've provided some sample JavaScript document extractors to help you extract attribute values from PDF content [6] . Each example has an extract of the HTML structure of the PDF [8] and a corresponding document extractor.

### Sample 1: Simple JavaScript extractor to get attribute values from PDF content

This example shows how to create a simple JavaScript extractor that uses conditional functions to get attribute values.

The PDF is available at *https://archive.doc.sitecore.com/xp/en/legacy-docs/web-forms-for-marketers-8.0.pdf*.Here's what the HTML structure looks like, shortened for brevity:

```
<head>
  <meta name="pdf:PDFVersion" content="1.6">
  <meta name="xmp:CreatorTool" content="Adobe Acrobat Standard DC 18.11.20058">
  <meta name="dc:description" content="All the official Sitecore documentation.">
  <meta name="dc:creator" content="John Doe">
  <meta name="dcterms:created" content="2018-09-13T09:53:31Z">
  <meta name="dcterms:modified" content="2018-09-13T09:53:31Z">
  <meta name="dc:format" content="application/pdf; version=1.6">
  ...
  <title></title>
</head>

<body>
  <div class="page">
    <p> </p>
    <p> </p>
    <p>Web Forms for Marketers 8.0 Rev: September 13, 2018 </p>
    <p> </p>
    <p> </p>
    <p>Web Forms for Marketers 8.0 </p>
    <p>All the official Sitecore documentation. </p>
  </div>
  <div class="page">
    <p> </p>
    <p>Add an ASCX control to the page In the Web Forms for Marketers module, you can co
      .ascx file and then add it to your website as an ASCX control. For developers, thi
      their custom form control. </p>
    <p>To add an ASCX control to the page: </p>
    <p>1. Using a text editor, in the \layouts folder of your Sitecore installation, cre
      insert the following code: </p>
    <p>&lt;%@ Page Language="C#" AutoEventWireup="true" %&gt; </p>
    <p>&lt;!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" </p>
    <p>"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"&gt; </p>
    <p>&lt;html xmlns="http://www.w3.org/1999/xhtml" &gt; </p>
    <p>&lt;head runat="server"&gt; </p>
    <p> &lt;title&gt;Untitled Page&lt;/title&gt; </p>
    <p>&lt;/head&gt; </p>
    <p>&lt;body&gt; </p>
    <p> &lt;form id="form1" runat="server"&gt; </p>
    <p> &lt;/form&gt; </p>
    <p>&lt;/body&gt; </p>
    <p>&lt;/html&gt; </p>
    ....
    <p> </p>
  </div>
  </body>
```

Here's a sample JavaScript document extractor to extract the *name*, *type*, *website*, *description*, *abstract*, *author* , and *last_modified* attributes from this PDF:

```
function extract(request, response) {
    $ = response.body;

    return [{
        'name':  $('div.page:eq(0)').text().trim().substring(0, 40) || 'No Name',
        'type': 'pdf',
        'website':'Sitecore Documentation',
        'description' : $('div.page:eq(0)').text().trim().substring(0, 100) || 'No Descr
    'author': $('meta[name="dc:creator"]').attr('content') || 'No Author',
    'last_modified': $('meta[name="dcterms:created"]').attr('content') || 'No Last Modif
    }];
}
```

This function uses the following logic to get attribute values:

- `name` - trim text in the first `div` tag with the `class` of `page` . Then, use only the first 40 characters. If there is no text, set the value to *No Name*.
- `type` - use a fixed value, *pdf*.
- `website` - use a fixed value, *Sitecore Documentation*.
- `description` - use the text in the first `<div>` tag of the class `page`, taking only the first 100 characters. If there is no text, set the value to *No Description*.
- `author` - use the content value of the first `meta` tag with the `name` of `dc:creator`. If there is no text, set the value to *No Author*.
- `last_modified` - use the text in the first `meta` tag with the `name` of `dcterms:created`. If there is no text, set the value to *No Last Modified Date*.

## Sample 2: Complex JavaScript extractor to get attribute values from PDF content

This example shows how to create a complex JavaScript extractor with many nested functions to get the attribute values you want. It also defines how to extract a `parent_url` attribute to track the parent page the PDF is on.

The PDF is available at *https://www.sitecore.com/customers/associations/us-masters-swimming*, by clicking **Download case study**. Here's what the HTML structure looks like, shortened for brevity:

```
<head>
    <meta name="pdf:PDFVersion" content="1.4">
    <meta name="xmp:CreatorTool" content="Adobe InDesign 15.0 (Macintosh)">
    <meta name="dcterms:created" content="2020-01-28T22:16:01Z">
    <meta name="dcterms:modified" content="2020-01-28T22:16:02Z">
    ...
    <title></title>
</head>

<body>
    <div class="page">
        <p> </p>
        <p> </p>
        <p> </p>
        <p>Industry: Associations • Founded: 1970 • Employees: 17 </p>
        <p>Headquarters: Boca Raton, Florida, USA • usms.org </p>
        <p>....
        <p> </p>
        <div class="annotation"> <a href="https://usms.org">https://usms.org</a> </div>
    </div>
    <div class="page">
        <p> </p>
        <p> </p>
        <p>Sitecore is the global leader in experience management software that combines
            and customer insights. The Sitecore® Experience Cloud™ empowers marketers </
       ...
        <div class="annotation"> <a href="https://sitecore.com">https://sitecore.com</a>
        <div class="annotation"> <a href="https://buildabonfire.com ">https://buildabonf
    </div>
</body>
```

Here's a sample JavaScript document extractor to extract the *id*, *type*, *last_modified*, *name*, *description*, and *parent_url* attributes from this PDF:

```
function extract(request, response) {
    const translate_re = /&(nbsp|amp|quot|lt|gt);/g;

    function decodeEntities(encodedString) {
        return encodedString.replace(translate_re, function(match, entity) {
            return translate[entity];
        }).replace(/&#(\d+);/gi, function(match, numStr) {
            const num = parseInt(numStr, 10);
            return String.fromCharCode(num);
        });
    }

    function sanitize(text) {
        return text ? decodeEntities(String(text).trim()) : text;
    }

    $ = response.body;
    url = request.url;
    id = url.replace(/[.:/&?=%]/g, '_');
    name = sanitize($('name').text());
    description = $('body').text().substring(0, 7000);

    $p = request.context.parent.response.body;
    if (name.length <= 4 && $p) {
        name = $p('name').text();
    }

    parentUrl = request.context.parent.request.url;
    last_modified = request.context.parent.documents[0].data.last_modified;

    return [{
        'id': id,
        'type': "pdf",
        'parent_url': parentUrl
        'last_modified': last_modified,
        'name': name,  //
        'description': description,

    }];
}
```

This function uses the following logic to get attribute values:

- `id` - replace special characters in the URL with an underscore (_).
- `type` - use a fixed value, *pdf*.
- `parent_url` - within the parent context (`request.context.parent`) access the `request` object. Then, access the `url` parameter.
- `last_modified` - within the parent context (`request.context.parent`) access the first documents array (`documents[0]`). Then, access the *data* object of the URL's *last_modified* attribute.
- `name` - use either the `name` HTML element or the parent document's `name` HTML element, as follows:
  - First, sanitize the text of the `<name>` HTML element.
  - Then, to check if the sanitized name is too short, see if the length is less than or equal to 4 characters.

  - If the name is too short, AND the parent document has a defined body (`$p`), use the `name` tag from the parent.
- `description` - sanitize the text of the `<body>` HTML element, limiting it to the first 7000 characters.

# Configuring tags for a crawler

**Abstract**
Describes the different types of tags in a Sitecore Search source that help customize indexing and control attributes to create a specific search experience.

In Sitecore Search, a tag is a source-level mechanism that gives you precise control over the attributes you want to use to create a specific search experience. Tags establish a connection between entities and sources, allowing the flexibility to create index [3] documents with combinations and aggregations of attributes from the same or different entities.

**NOTE**
You can use tags in the advanced web crawler and API crawler sources.

When you configure attribute extraction rules, you do so per tag and not per entity. As a result, you get one set of index documents for each tag for which you configure extraction rules. Keep this in mind when you plan and configure tags.

Tags are specific to a source and do not persist across sources. For example, if you define a tag called *actor with movies* in source *A*, the tag is unique to source *A*. If you create source *B* and define another tag named *actor with movies*, Search considers it a new tag with its own logic.

When you create a source, you get some *default tags*. You can also create *custom tags* to build specific search experiences.

## Default tags

Sitecore Search automatically generates default tags when you create a source. Each tag corresponds to an entity within your domain. You can use default tags to define attribute extraction rules in the document extractor.

Default tags suffice when:

- You want to use one entity's attributes to create only one search experience.
  And
- You want each index document to have attribute values from only one content item. In other words, you do not want to combine or aggregate attributes from multiple content items.

For example, consider a film and television website where visitors can search for actors, TV shows, and movies. You want to create a search experience where visitors see only the attributes of each entity that a result belongs to. To do this, you can simply define extraction rules for the default tags in the document extractor.

## Custom tags

Custom tags are any tags you create. Use custom tags when default tags are not enough. You can create as many custom tags as you want.

Usually, you'll need to configure custom tags when:

- You want to use the attributes of one entity to create more than one search experience.

Or
- You want to enhance index documents with aggregations of attribute values or combinations of attributes from more than one entity into an index document.

To continue with the example of the film and TV website, you might want to create a search experience where you show a list of movies an actor has acted in, along with general information like *name* and *age*. In this example, movie information is not available in any attribute of the *Actor* entity. However, the *Movie* entity has a *movie_actor* attribute. You can use this relationship to tie actors to a list of movies they acted in [20].

There are two ways to configure custom tags in Search: *Basic tag configuration* and *Aggregated tag configuration*.

## Basic tag configuration

A basic tag configuration is a simple setup you use when you want to create more than one set of index documents from an entity. Usually, you define one or more tags and assign them to an entity.

**TIP**

We recommend that you assign one basic tag to only one entity. Assigning basic tags to multiple entities defeats the purpose of entities as distinct, search-indexed object types. Because there is no mapping definition in basic tags, if you use the same basic tag across multiple entities, you might get disorganized index documents that are not conducive to useful search experiences.

Instead, if you want to pull attributes from multiple entities into a single index document, we recommend using an *aggregated tag configuration*.

### Example of using basic tags

For example, you have a website that hosts blogs. Some blogs are written by internal writers, and some by external freelancers. Both types of blogs have the same attributes but require different attribute extraction logic. You want your visitors to have a unified search experience where they can search among all blogs. Your implementation has one entity, *Blog*.

Here's one way to handle this requirement: You can configure basic tags called *internal blogs* and *external blogs* under the *Blog* entity. When you extract attributes for the *internal blogs* tag, use extraction logic that suits the internal content. When you extract attributes for the *external blogs* tag, use extraction logic that suits the external content.

**NOTE**

Another way to handle this requirement is to configure two sources, one for external content and one for internal content. Using tags is easier because developers don't need to keep track of multiple source IDs to pass at runtime.

Here's a sample index document extracted for the *internal* tag:

```
"id": xvsusdua232533ygd,
"name": "Top 10 Most Anticipated Cars in 2024",
"type": "blog",
"description": "some description"
```

Here's a sample index document extracted for the *external* tag:

```
 "id": agahsfeev1nnbsa7d,
 "name": "How to Save 10% on Your New Car",
 "type": "blog",
 "description": "some other description"
```

As you can see, index documents extracted by both tags look identical. Attributes from both sets of index documents are from the Blog entity. The only difference was in their extraction logic. To a visitor, there is no difference between *freelancer* content and *internal* content.

## Aggregated tag configuration

An aggregated tag configuration [20] is an advanced setup where you can create index document that have related attributes from more than one content item. You define mappings to aggregate related attributes. Then, you create a new attribute to combine and hold this aggregated information. Configuring tags through aggregation is a powerful way to enhance search experiences when dealing with relational data.

Use aggregated tags when you want to take advantage of a relationship between attributes in the same entity or different entities. An example of this relationship is when items have a shared attribute value, like *name* or *id*.

Depending on whether you want to define a relationship between attributes in the same entities or other entities, use *flat* aggregation or *hierarchical* aggregation.

### Using flat aggregation to combine attributes from different entit

Use flat aggregation when you want to enhance index documents with a new attribute that contains aggregated information from related items across multiple entities.

For example, you have a website that has information about actors, movies, and TV shows. There is a search bar where visitors can search for results. Your implementation has the *Actor* and *Movie* entities. The *Actor* entity has attributes like *actor_name*, *actor_age*, and so on. It does not have attributes about movies the actor has been in. The *Movie* entity has attributes like *movie_name*, *movie_year*, and a *movie_cast* attribute with a list of actors.

You want to create the following search experience: when an actor appears, also show a list of movies they acted in.

To support this experience, you'll need index documents with *Actor* attributes and an attribute that lists movies they have acted in.

To get these index documents, you can create a tag through flat aggregation under the *Actor* entity and associate it with the *Movie* entity. Then, configure this tag to use the actor's name as a baseline to find movies in which they've acted.

Here's a sample index document extracted for this tag:

```
{
  "actor_id": "A101",
  "actor_name": "Tom Cruise",
  "actor_description": "A critically acclaimed actor...",
  "actor_age": 61,
  "actor_gender": "male",
  "filmography": [
    {
      "movie_name": "Top Gun",
      "movie_year": 1986,
      "movie_genre": "Action"
    },
    {
      "movie_name": "Edge of Tomorrow",
      "movie_year": 2014,
      "movie_genre": "Science Fiction"
    },
    ...
  ]
```

**NOTE**

In this sample index document, attributes within the `filmography` array come from other items in the *Movie* entity, which have one cast value of *Tom Cruise* (assuming that cast is an array of strings).

## Using hierarchical aggregation to combine attributes from one entity

Use hierarchical aggregation when you want to enhance index documents with a new attribute that contains aggregated information from related items in one entity.

For example, you have a website where visitors can search for employees who work for your company. Your implementation has an entity called *Employee*, and one of the attributes in this entity is *manager*.

You want to create the following search experience: when an employee appears in the search results, also show a list of their colleagues.

To support this experience, you'll need index documents that have regular *Employee* attributes as well as an attribute that lists colleagues.

To get these index documents, you can create a tag under the *Employee* entity. Then, configure this tag to use the *manager* attribute as a baseline to find employees with the same manager, that is, to find colleagues.

Here's a sample index document extracted for this tag:

```
{
  "employee_id": 1,
  "name": "John Doe",
  "position": "Senior developer",
  "department": "R&D",
  "manager": "Mary Smith",
  "colleagues": [
    {
      "employee_id": 2,
      "name": "Jane Smith",
      "position": "Lead Developer",
    },
    {
      "employee_id": 3,
      "name": "David Johnson",
      "position": "Senior  Developer",
        }
    ]
}
```

**NOTE**
In this sample index document, attributes within the `colleagues` array come from other content items in the *Employee* entity who have a `manager` value of *Mary Smith*.

## Sample source and tag configuration for an implementation with three entities

Consider a film and television website where visitors can search for actors, TV shows, and movies. Your implementation has the *Actor*, *Movie* , and *TVShow* entities.

The *Actor* entity has the following attributes: *actor_awards*, *actor_age*, *actor_gender*, *actor_movies*, *actor_name*, *actor_image*, *actor_tv*, *actor_bio*

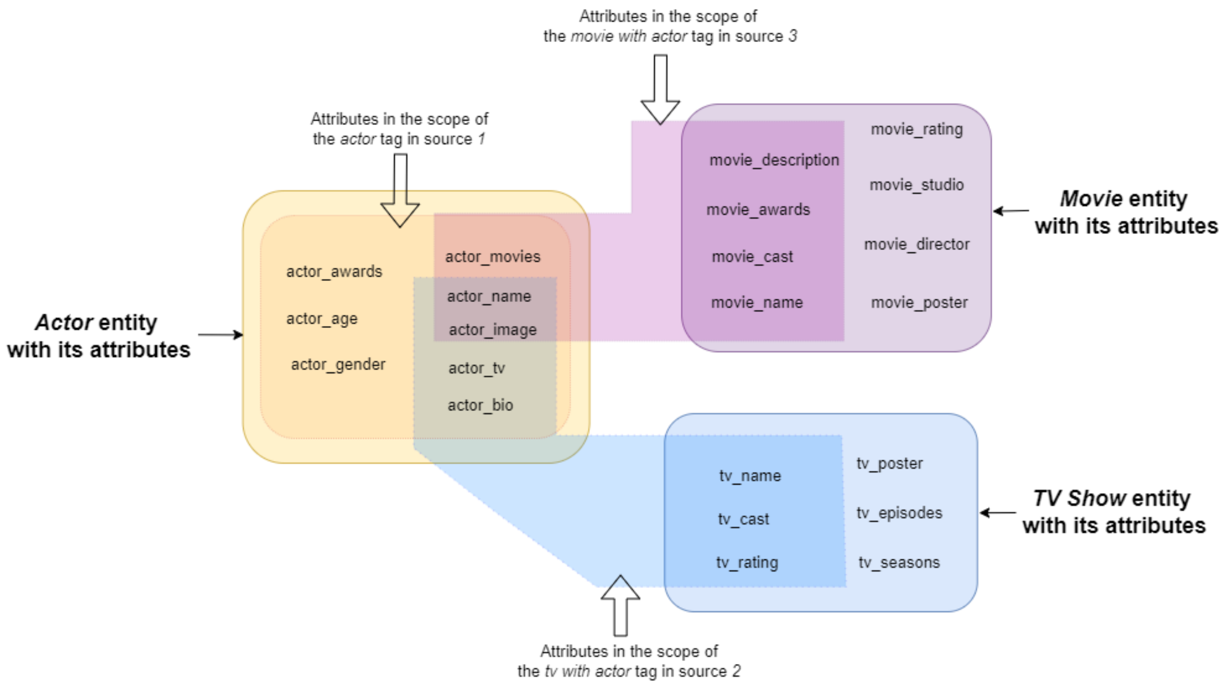
The *Movie* entity has the following attributes: *movie_description*, *movie_awards*, *movie_actor*, *movie_name*, *movie_rating*, *movie_studio*, *movie_director*, *movie_poster*.

The *TV Show* entity has the following attributes: *tv_name*, *tv_cast*, *tv_rating*, *tv_poster*, *tv_episodes*, *tv_seasons*.

You want to create the following search experiences:

- When an actor appears, visitors see all the details of that actor.
- When a TV show appears, visitors see TV show details and some related cast details.
- When a movie appears, visitors see movie details and some related cast details.

The following image shows a sample source and tag configuration that you can use to get the desired search experiences:


Attributes in the scope of the *movie with actor* tag in source *3*
Attributes in the scope of the *actor* tag in source *1*
movie_rating
movie_description
movie_studio
movie_awards
movie_director
movie_cast
movie_name
movie_poster
*Movie* entity with its attributes
actor_movies
actor_awards
actor_name
actor_age
actor_image
actor_gender
actor_tv
actor_bio
*Actor* entity with its attributes
tv_poster
tv_name
tv_episodes
tv_cast
tv_rating
tv_seasons
*TV Show* entity with its attributes
Attributes in the scope of the *tv with actor* tag in source *2*

Here's an explanation of this image:

- To show all actor information, you can create a source (source *1* ) and then use the default *actor* tag Search creates for the *Actor* entity. In the document extractor, for the *actor* tag, extract all available attributes. This will get you an index document that has all the actor details.

  **NOTE**
  You do not need to define any custom tags to achieve this search experience.

- To show cast details along with TV show details when a TV show appears, you can create another source (source *2*). In this source, you can create a tag through flat aggregation [20] called *tv with actor,*, associate it with the *TVShow* entity, and configure it to borrow from the *Actor* entity. In the document extractor, for this tag, extract required attributes from both *TVShow* and *Actor* entities.
- To show cast details along with TV show details when a TV show appears, you can create another source (source *3)*. In this source, you can create a tag through flat aggregation [20] called *movie with actor*, associate it with the *Movie* entity and configure it to borrow from the *Actor* entity. In the document extractor, for this tag, extract required attributes from both *TVShow* and *Actor* entities.

**NOTE**
For brevity, this example is restricted to three sources with one tag each. In reality, you might need more than one tag in a source, or you might need more sources.

For example, you might want to have a search experience that draws only from the movie details. To enable this, create a source to extract only movie content and use the default *movie* tag to extract all attributes.

## Use aggregated tags to combine attributes from multiple entities into one index document

**Abstract**
Configure a tag to combine attributes from multiple entities by using flat aggregation in a Sitecore Search source.

Configure tags [15] through aggregation when you have relational data, and you want to combine information from multiple content items into a new attribute.

**NOTE**
If you need a simple tag configuration where you only assign a tag to one or more entities, use basic tags. With basic tags, Search does not evaluate and match based on shared attributes.

When you configure tags through aggregations, you define a mapping between attributes for Search to evaluate and match. You also specify which attributes you want to include in an index document upon a successful match.

**NOTE**

This topic describes how to create a tag when there is a relationship between attributes in different entities, that is, when you need flat aggregation [17]. To define a relationship between attributes in the same entity, define tags through hierarchical aggregation. To do this, follow the same process configuring a tag through flat aggregation, but ensure that you accurately define fields within the same entity.

This topic shows you how to create a tag through flat aggregation in the following scenario:

- You have a website that has information about actors and movies.
- Your implementation has the *Actor* and *Movie* entities.
  - The *Actor* entity has the following attributes:
    - *actor_id*
    - *actor_name*.
    - *actor_description*
    - *actor_age*
    - *actor_gender*
  - The *Movie* entity has the following attributes:
    - *movie_id*
    - *movie_name*
    - *movie_genre*
    - *movie_year*
    - *movie_cast*
    - *movie_director*
    - *movie_awards*
- You want to configure this search experience: when an actor appears as a search result, show a list of movies they acted in.

To support this experience, you'll need index documents that have *Actor* attributes as well as an attribute that lists movies they have acted in. To get this, you can create a tag through flat aggregation under the *Actor* entity and also associate it with the *Movie* entity. Then, use the actor's name as the common attribute between entities and map actors to movies.

Here's why you do this: While the *Actor* entity does not have movie information, the *Movie* entity has the actor information contained with the *movie_cast* attribute.

**NOTICE**

Before you configure a tag through aggregations:

- Create a new attribute to hold the aggregated information. Ensure that the attribute is of the type `array of objects` and that it belongs to the same entity as the tag you're creating.

You'll need to do this for tags you create through flat or hierarchical aggregation.

In this example, you can create an attribute called *filmography* that belongs to the *Actor* entity.

To configure a tag through flat aggregations:

1. Log in to the Customer Engagement Console (CEC), click **Sources**, and select the source you created. Then, on the **Source Settings** page, next to **Document Extractors**, click edit.
2. At the bottom of the **Tags Definition** page, click **Add Tag** + .
3. To associate the tag with an entity, in the **Entity** drop-down menu, click an entity.

   **NOTE**
   When you configure tags though aggregation, you can think of the **Entity** as the primary entity or parent entity for the tag.

   In this example, click the *Actor* entity.
4. To specify that you want to configure tags through aggregation, in the **From** drop-down menu, click **Aggregations**. To configure the type of aggregation to use, in the **Type** drop-down menu, click the type of aggregation you need.
   In this example, click **Flat**.
5. To name the tag, in the **Source Tag** field, enter a meaningful name that describes the purpose of the tag.

   **NOTE**
   Name the tag so that any user can understand the tag's purpose just by looking at its name. This is to avoid confusion because, in the document extractor, you select from a list of tag names and then configure extraction logic for that tag.

   In this example, enter *actor with movies*.
6. Define the entity *from* which you want to borrow attributes. To do this, in the **Foreign Tag** field, enter the name of the entity.

   **NOTE**
   When you configure tags though aggregation, you can think of **Foreign Tag** as the foreign entity for the tag.

   In this example, enter *Movie*.
7. Define which attribute in the parent entity (the **Entity** you defined in *Step 2*) you want to use as a baseline to check if a mapping is possible. To do this, in the **Source Field** field, enter the name of an attribute from the parent entity.
   In this example, enter *actor_name*.
8. Define which attributes of the foreign entity you want to use as a comparison to check if a mapping is possible. To do this, in the **Foreign Field** field, enter the attribute name.
   In this example, enter *movie_cast*. You can do this because *movie_cast* is an array of strings, with each string being the name of an actor from this movie.
9. Specify the attribute to combine and hold the content from the foreign entity. To do this, in the **Target Tag** field, enter the name of the attribute you configured in the prerequisite.
   In this example, enter *filmography*.
10. Specify which attributes of the foreign entity you want to pull into the index document. To do this, in the **Fields to Project** field, enter attribute names from the foreign entity.
    After extraction, where mapping is successful, these attributes appear nested within the new `array of objects` attribute you specified in the **Target Tag** field.
    For example, enter *movie_name*, *movie_year*, and *movie_genre*.
11. Click **Save**.

**NOTE**
Optionally, if you also want to include attributes from another entity in this index document, repeat Steps 2 through 9. Ensure that the tag name you enter in the **Source Tags** field is identical to the first tag you created. This is because you want to expand the scope of the existing tag, not define a new tag.

After you configure a tag, in the document extractor for the source, select the tag you created and extract attributes according to your use case.