Contentful & Astro | Docs

docs.astro.build/en/guides/cms/contentful

Contentful is a headless CMS that allows you to manage content, integrate with other services, and publish to multiple platforms.

Integrating with Astro

In this section, we'll use the Contentful SDK to connect your Contentful space to Astro with zero client-side JavaScript.

Prerequisites

To get started, you will need to have the following:

1. **An Astro project** - If you don't have an Astro project yet, our Installation guide will get you up and running in no time.
2. **A Contentful account and a Contentful space**. If you don't have an account, you can sign up for a free account and create a new Contentful space. You can also use an existing space if you have one.
3. **Contentful credentials** - You can find the following credentials in your contentful dashboard **Settings > API keys**. If you don't have any API keys, create one by selecting **Add API key**.
    - **Contentful space ID** - The ID of your Contentful space.
    - **Contentful delivery access token** - The access token to consume *published* content from your Contentful space.
    - **Contentful preview access token** - The access token to consume *unpublished* content from your Contentful space.

Setting up credentials

To add your Contentful space's credentials to Astro, create an `.env` file in the root of your project with the following variables:

.env

```
CONTENTFUL_SPACE_ID=YOUR_SPACE_ID

CONTENTFUL_DELIVERY_TOKEN=YOUR_DELIVERY_TOKEN

CONTENTFUL_PREVIEW_TOKEN=YOUR_PREVIEW_TOKEN
```

Now, you can use these environment variables in your project.

If you would like to have IntelliSense for your Contentful environment variables, you can create a `env.d.ts` file in the `src/` directory and configure `ImportMetaEnv` like this:

src/env.d.ts

```
interface ImportMetaEnv {
readonly CONTENTFUL_SPACE_ID: string;
readonly CONTENTFUL_DELIVERY_TOKEN: string;
readonly CONTENTFUL_PREVIEW_TOKEN: string;
}
```

Tip

Read more about using environment variables and `.env` files in Astro.

Your root directory should now include these new files:

- ▼ src/
  TS**env.d.ts**
- **.env**
- astro.config.mjs
- { }package.json

Installing dependencies

To connect with your Contentful space, install both of the following using the single command below for your preferred package manager:

- `contentful.js`, the official Contentful SDK for JavaScript
- `rich-text-html-renderer`, a package to render Contentful's rich text fields to HTML.

- npm
- pnpm
- Yarn

```
npm install contentful @contentful/rich-text-html-renderer
```

Next, create a new file called `contentful.ts` in the `src/lib/` directory of your project.

src/lib/contentful.ts

```
import contentful from "contentful";


export const contentfulClient = contentful.createClient({

space: import.meta.env.CONTENTFUL_SPACE_ID,

accessToken: import.meta.env.DEV

? import.meta.env.CONTENTFUL_PREVIEW_TOKEN

: import.meta.env.CONTENTFUL_DELIVERY_TOKEN,

host: import.meta.env.DEV ? "preview.contentful.com" : "cdn.contentful.com",

});
```

The above code snippet creates a new Contentful client, passing in credentials from the `.env` file.

⚠Caution

While in development mode, your content will be fetched from the **Contentful preview API**. This means that you will be able to see unpublished content from the Contentful web app.

At build time, your content will be fetched from the **Contentful delivery API**. This means that only published content will be available at build time.

Finally, your root directory should now include these new files:

- ▼ src/
  - TSenv.d.ts
  - ▼ lib/
    - **TScontentful.ts**
- .env
- astro.config.mjs
- { }package.json

Fetching data

Astro components can fetch data from your Contentful account by using the `contentfulClient` and specifying the `content_type`.

For example, if you have a “blogPost” content type that has a text field for a title and a rich text field for content, your component might look like this:

```
---

import { contentfulClient } from "../lib/contentful";

import { documentToHtmlString } from "@contentful/rich-text-html-renderer";

import type { EntryFieldTypes } from "contentful";



interface BlogPost {

  contentTypeId: "blogPost",

  fields: {

    title: EntryFieldTypes.Text

    content: EntryFieldTypes.RichText,

  }

}



const entries = await contentfulClient.getEntries<BlogPost>({

  content_type: "blogPost",

});

---

<body>

  {entries.items.map((item) => (

    <section>
```

```
      <h2>{item.fields.title}</h2>

      <article set:html={documentToHtmlString(item.fields.content)}></article>

    </section>

  ))}

</body>
```

Tip

If you have an empty Contentful space, check out setting up a Contentful model to learn how to create a basic blog model for your content.

You can find more querying options in the Contentful documentation.

Making a blog with Astro and Contentful

With the setup above, you are now able to create a blog that uses Contentful as the CMS.

Prerequisites

1. **A Contentful space** - For this tutorial we recommend starting with an empty space. If you already have a content model, feel free to use it, but you will need to modify our code snippets to match your content model.
2. **An Astro project integrated with the Contentful SDK** - See integrating with Astro for more details on how to set up an Astro project with Contentful.

Setting up a Contentful model

Inside your Contentful space, in the **Content model** section, create a new content model with the following fields and values:

- **Name:** Blog Post
- **API identifier:** `blogPost`
- **Description:** This content type is for a blog post

In your newly created content type, use the **Add Field** button to add 5 new fields with the following parameters:

1. Text field
    - **Name:** title
    - **API identifier:** `title` (leave the other parameters as their defaults)
2. Date and time field
    - **Name:** date
    - **API identifier:** `date`
3. Text field
    - **Name:** slug
    - **API identifier:** `slug` (leave the other parameters as their defaults)
4. Text field
    - **Name:** description
    - **API identifier:** `description`
5. Rich text field
    - **Name:** content
    - **API identifier:** `content`

Click **Save** to save your changes.

In the **Content** section of your Contentful space, create a new entry by clicking the **Add Entry** button. Then, fill in the fields:

- **Title:** `Astro is amazing!`
- **Slug:** `astro-is-amazing`
- **Description:** `Astro is a new static site generator that is blazing fast and easy to use.`
- **Date:** `2022-10-05`
- **Content:** `This is my first blog post!`

Click **Publish** to save your entry. You have just created your first blog post.

Feel free to add as many blog posts as you want, then switch to your favorite code editor to start hacking with Astro!

Displaying a list of blog posts

Create a new interface called `BlogPost` and add it to your `contentful.ts` file in `src/lib/`. This interface will match the fields of your blog post content type in Contentful. You will use it to type your blog post entries response.

src/lib/contentful.ts

```
import contentful, { EntryFieldTypes } from "contentful";


export interface BlogPost {

contentTypeId: "blogPost",

fields: {

title: EntryFieldTypes.Text

content: EntryFieldTypes.RichText,

date: EntryFieldTypes.Date,

description: EntryFieldTypes.Text,

slug: EntryFieldTypes.Text

}

}


export const contentfulClient = contentful.createClient({

space: import.meta.env.CONTENTFUL_SPACE_ID,

accessToken: import.meta.env.DEV

? import.meta.env.CONTENTFUL_PREVIEW_TOKEN

: import.meta.env.CONTENTFUL_DELIVERY_TOKEN,

host: import.meta.env.DEV ? "preview.contentful.com" : "cdn.contentful.com",

});
```

Next, go to the Astro page where you will fetch data from Contentful. We will use the home page `index.astro` in `src/pages/` in this example.

Import `BlogPost` interface and `contentfulClient` from `src/lib/contentful.ts`.

Fetch all the entries from Contentful with a content type of `blogPost` while passing the `BlogPost` interface to type your response.

src/pages/index.astro

```
---
import { contentfulClient } from "../lib/contentful";
import type { BlogPost } from "../lib/contentful";

const entries = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});
---
```

This fetch call will return an array of your blog posts at `entries.items`. You can use `map()` to create a new array (`posts`) that formats your returned data.

The example below returns the `items.fields` properties from our Content model to create a blog post preview, and at the same time, reformats the date to a more readable format.

src/pages/index.astro

```
---
import { contentfulClient } from "../lib/contentful";
import type { BlogPost } from "../lib/contentful";

const entries = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});

const posts = entries.items.map((item) => {
const { title, date, description, slug } = item.fields;
return {
title,
slug,
description,
date: new Date(date).toLocaleDateString()
};
});
---
```

Finally, you can use `posts` in your template to show a preview of each blog post.

src/pages/index.astro

```
---
import { contentfulClient } from "../lib/contentful";
import type { BlogPost } from "../lib/contentful";


const entries = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});


const posts = entries.items.map((item) => {
const { title, date, description, slug } = item.fields;
return {
title,
slug,
description,
date: new Date(date).toLocaleDateString()
};
});
---
<html lang="en">
<head>
<title>My Blog</title>
</head>
<body>
<h1>My Blog</h1>
<ul>
{posts.map((post) => (
<li>
<a href={`/posts/${post.slug}/`}>
<h2>{post.title}</h2>
</a>
<time>{post.date}</time>
<p>{post.description}</p>
</li>
))}
</ul>
</body>
</html>
```

Generating individual blog posts

Use the same method to fetch your data from Contentful as above, but this time, on a page that will create a unique page route for each blog post.

Static site generation

If you're using Astro's default static mode, you'll use dynamic routes and the `getStaticPaths()` function. This function will be called at build time to generate the list of paths that become pages.

Create a new file named `[slug].astro` in `src/pages/posts/`.

As you did on `index.astro`, import the `BlogPost` interface and `contentfulClient` from `src/lib/contentful.ts`.

This time, fetch your data inside a `getStaticPaths()` function.

src/pages/posts/[slug].astro

```
---
import { contentfulClient } from "../../lib/contentful";
import type { BlogPost } from "../../lib/contentful";

export async function getStaticPaths() {
const entries = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});
}
---
```

Then, map each item to an object with a `params` and `props` property. The `params` property will be used to generate the URL of the page and the `props` property will be passed to the page component as props.

src/pages/posts/[slug].astro

```
---
import { contentfulClient } from "../../lib/contentful";
import { documentToHtmlString } from "@contentful/rich-text-html-renderer";
import type { BlogPost } from "../../lib/contentful";

export async function getStaticPaths() {
const entries = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});

const pages = entries.items.map((item) => ({
params: { slug: item.fields.slug },
props: {
title: item.fields.title,
content: documentToHtmlString(item.fields.content),
date: new Date(item.fields.date).toLocaleDateString(),
},
}));
return pages;
}
---
```

The property inside `params` must match the name of the dynamic route. Since our filename is `[slug].astro`, we use `slug`.

In our example, the `props` object passes three properties to the page:

- title (a string)
- content (a rich text Document converted to HTML using `documentToHtmlString`)
- date (formatted using the `Date` constructor)

Finally, you can use the page `props` to display your blog post.

src/pages/posts/[slug].astro

```
---
import { contentfulClient } from "../../lib/contentful";
import { documentToHtmlString } from "@contentful/rich-text-html-renderer";
import type { BlogPost } from "../../lib/contentful";

export async function getStaticPaths() {
const { items } = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
});
const pages = items.map((item) => ({
params: { slug: item.fields.slug },
props: {
title: item.fields.title,
content: documentToHtmlString(item.fields.content),
date: new Date(item.fields.date).toLocaleDateString(),
},
}));
return pages;
}

const { content, title, date } = Astro.props;
---
<html lang="en">
<head>
<title>{title}</title>
</head>
<body>
<h1>{title}</h1>
<time>{date}</time>
<article set:html={content} />
</body>
</html>
```

Navigate to http://localhost:4321/ and click on one of your posts to make sure your dynamic route is working!

Server side rendering

If you’ve opted in to SSR mode, you will use a dynamic route that uses a `slug` parameter to fetch the data from Contentful.

Create a `[slug].astro` page in `src/pages/posts`. Use `Astro.params` to get the slug from the URL, then pass that to `getEntries`:

src/pages/posts/[slug].astro

```
---
import { contentfulClient } from "../../lib/contentful";
import type { BlogPost } from "../../lib/contentful";

const { slug } = Astro.params;

const data = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
"fields.slug": slug,
});
---
```

If the entry is not found, you can redirect the user to the 404 page using `Astro.redirect`.

src/pages/posts/[slug].astro

```
---
import { contentfulClient } from "../../lib/contentful";
import type { BlogPost } from "../../lib/contentful";

const { slug } = Astro.params;

try {
const data = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
"fields.slug": slug,
});
} catch (error) {
return Astro.redirect("/404");
}
---
```

To pass post data to the template section, create a `post` object outside the `try/catch` block.

Use `documentToHtmlString` to convert `content` from a Document to HTML, and use the Date constructor to format the date. `title` can be left as-is. Then, add these properties to your `post` object.

src/pages/posts/[slug].astro

```
---
import Layout from "../../layouts/Layout.astro";
import { contentfulClient } from "../../lib/contentful";
import { documentToHtmlString } from "@contentful/rich-text-html-renderer";
import type { BlogPost } from "../../lib/contentful";

let post;
const { slug } = Astro.params;
try {
const data = await contentfulClient.getEntries<BlogPost>({
content_type: "blogPost",
"fields.slug": slug,
});
const { title, date, content } = data.items[0].fields;
post = {
title,
date: new Date(date).toLocaleDateString(),
content: documentToHtmlString(content),
};
} catch (error) {
return Astro.redirect("/404");
}
---
```

Finally, you can reference `post` to display your blog post in the template section.

src/pages/posts/[slug].astro

```
---

import Layout from "../../layouts/Layout.astro";

import { contentfulClient } from "../../lib/contentful";

import { documentToHtmlString } from "@contentful/rich-text-html-renderer";

import type { BlogPost } from "../../lib/contentful";


let post;

const { slug } = Astro.params;

try {

const data = await contentfulClient.getEntries<BlogPost>({

content_type: "blogPost",

"fields.slug": slug,

});

const { title, date, content } = data.items[0].fields;

post = {

title,

date: new Date(date).toLocaleDateString(),

content: documentToHtmlString(content),

};

} catch (error) {

return Astro.redirect("/404");

}

---

<html lang="en">

<head>

<title>{post?.title}</title>

</head>

<body>

<h1>{post?.title}</h1>

<time>{post?.date}</time>

<article set:html={post?.content} />

</body>
```

```
</html>
```

Publishing your site

To deploy your website, visit our deployment guides and follow the instructions for your preferred hosting provider.

Rebuild on Contentful changes

If your project is using Astro’s default static mode, you will need to set up a webhook to trigger a new build when your content changes. If you are using Netlify or Vercel as your hosting provider, you can use its webhook feature to trigger a new build from Contentful events.

Netlify

To set up a webhook in Netlify:

1. Go to your site dashboard and click on **Build & deploy**.
2. Under the **Continuous Deployment** tab, find the **Build hooks** section and click on **Add build hook**.
3. Provide a name for your webhook and select the branch you want to trigger the build on. Click on **Save** and copy the generated URL.

Vercel

To set up a webhook in Vercel:

1. Go to your project dashboard and click on **Settings**.
2. Under the **Git** tab, find the **Deploy Hooks** section.
3. Provide a name for your webhook and the branch you want to trigger the build on. Click **Add** and copy the generated URL.

Adding a webhook to Contentful

In your Contentful space **settings**, click on the **Webhooks** tab and create a new webhook by clicking the **Add Webhook** button. Provide a name for your webhook and paste the webhook URL you copied in the previous section. Finally, hit **Save** to create the webhook.

Now, whenever you publish a new blog post in Contentful, a new build will be triggered and your blog will be updated.

More CMS guides