

How to build a simple blog with Next.js + Tailwind CSS + Contentful

In this tutorial we're going to explore how we can build a basic blog with the following technology:

- [Next.js](#) as main React framework, with support for **Static Rendering**
- [Tailwind](#) as **CSS** framework
- [Contentful](#) for creating and **managing the content** without worrying about building an actual backend

Before diving into the actual process, let's have a quick overview of the topics we are going to cover.

Static Rendering

One of the powerful features provided by Next.js is that we can write [Server Components](#). This allows us to use an approach called **Static Rendering**, which means that all the HTML pages will be pre-rendered at build time. The front-facing site will not send any requests to the backend (in this case Contentful). The Contentful API will be, in fact, queried only when we run the `next build` command and generate the static files. This will result in excellent page performance since the content doesn't have to be fetched on the client side.

Tailwind

Tailwind is, as per their definition, a “utility-first CSS framework”. What makes it different compared to other CSS frameworks is that it doesn't impose any design specifications. Class names are pre-defined and each class we are going to use is, in most cases, doing one thing and one thing only.

Let's say I want to apply some very basic styling to an H1 element. With a *Vanilla* CSS approach, I would do something like this:

```
<h1 class="myCustomClass">This is a sample text</h1>
```

And then, in the CSS file:

```
h1.myCustomClass {  
  font-size: 36px;  
  text-align: center;  
  color: rgb(34 197 94);  
  font-weight: bold;  
}
```

In Tailwind, to obtain the exact same result, we would use this syntax:

```
<h1 class="text-4xl text-center text-green-500 font-bold">This  
is a sample text</h1>
```

As we can observe, we used a specific class for each one of the CSS definitions.

The full list of available classes can be found in the [official documentation](#).

Is this really an improvement?

This was the first question that came to my mind the first time I stumbled across Tailwind and I still think it's a valid concern. At first glance it might look like we're writing inline styles, which goes against the concept of [separation of concerns](#). Also, the HTML markup already looks quite messy even with the very basic styling of the example, therefore we can expect it to be way more *polluted* once we apply some more advanced styling.

However, this approach **brings several benefits** as well:

- **No class names.** Having to choose names for classes in a consistent and organized way can be a difficult task, especially on projects with multiple contributors. Here we don't have to worry about this since we are going to use pre-existing utility classes.

- **Responsiveness out of the box.** Tailwind comes with an excellent mechanism for creating responsive websites. We just need to use prefixes like `sm:`, `md:` etc. to target the most common viewport breakpoints, without having to write our own media queries.
- **CSS bundle is optimized.** The build process of Tailwind includes only the classes that are actually used in the project, making sure that we end up with the smallest possible CSS file (especially when this is combined with minification and network compression).

Project Setup

Now that we have an overview of what we want to build, let's dive into the actual project creation.

The good news is that, since March 2023, the `--tailwind` flag is officially part of the `create-next-app` CLI tool. This means that we can bootstrap a Next.js project with Tailwind in less than a minute.

Let's open the terminal and run:

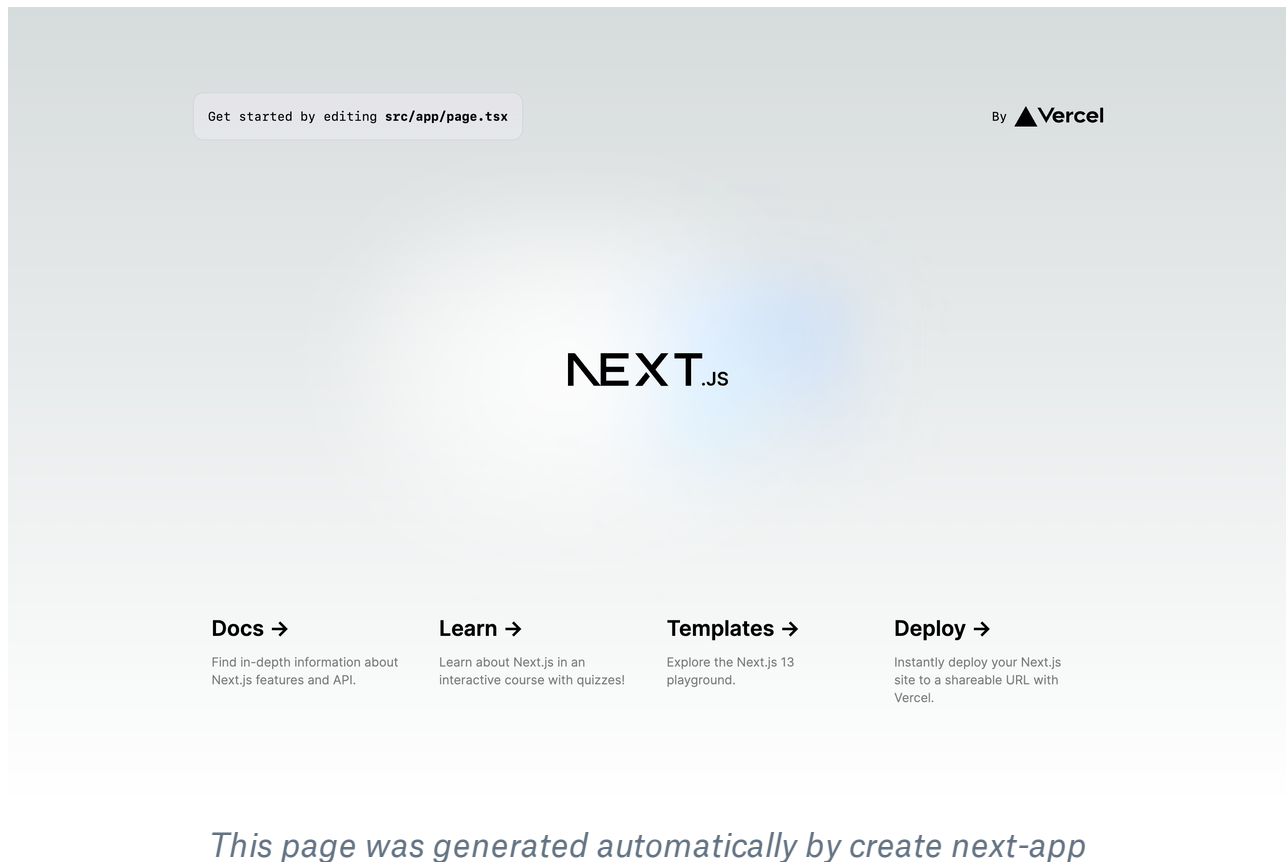
```
yarn create next-app
```

When prompted, let's answer the questions as follows:

```
What is your project named?  simple-blog-demo
Would you like to use TypeScript?  Yes
Would you like to use ESLint?  Yes
Would you like to use Tailwind CSS?  Yes
Would you like to use `src/` directory?  Yes
Would you like to use App Router? (recommended)  Yes
Would you like to customize the default import alias?  No
```

After the installation let's go to our newly created directory, run `yarn start dev` and open the browser on <http://localhost:3000/> to see what we have so far.

If everything worked as expected we should see this:



Our project with Next.js + Tailwind is ready. Let's now do some preparation work for the content part.

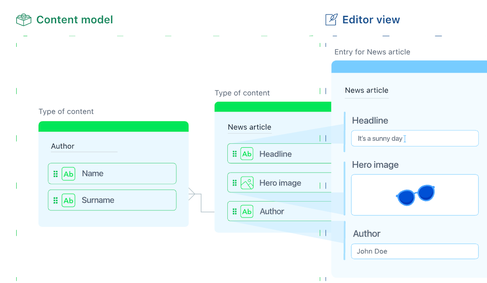
Set up content in Contentful

As mentioned earlier, the blog content will be entirely managed by Contentful. We can create a new account, or use an existing one, and then click on "Content Model" on the top menu.

First, design the content model

The content model is a collection of all different types of content for a project (like the green boxes on the right 🟢). It's a schema that editors repeatably use and fill with content. [Learn More](#)

Design your content model



If we don't have any pre-existing content types we just have to click on "Design your content model". Otherwise, we can just click on "Add Content Type" in the top right.

We will be asked to choose a name, let's go with "Blog". We can also add a description if we want, but it's not necessary for our demo.

Now that we have the content model it's time to define our fields. Let's click on "Add field", and select the type of field we need to create. I created a quick recap of the fields that we will need and their type. Please keep in mind that Field ID will be auto-filled when typing the name, so we will probably not need to type it at all.

Name	Field ID	Field Type	Notes
Title	title	Text (Short text, exact search)	"This field represents the Entry title" should be checked in the field properties
Slug	slug	Text (Short text, exact	In Appearance

		search)	make sure that Slug is selected. In Validation check the “ Unique Field ” checkbox.
Date	date	Date and Time	
Content	content	Rich Text	

Now that our Content Model is ready, we can create our blog posts. In my case, I created some dummy articles using **Chat GPT** just for the sake of the demo. Another option could be using the good old **Lorem Ipsum** or, even better, writing our own actual content.

The screenshot shows a web application interface for managing content. At the top, there's a navigation bar with 'Home', 'Content model', 'Content', 'Media', and 'Apps'. The 'Content' section is active. Below the navigation bar, there's a sidebar on the left with 'Scheduled Content', 'Shared views', and 'My views'. The main area displays a table of content entries. The table has columns for 'Name', 'Content Type', 'Updated', 'Status', and a settings icon. Three entries are listed, all of type 'Blog' and status 'Published'. The first entry is 'The Art of Demonstrative Writing: An Analysis of an Article with No Define...', the second is 'Unveiling the Mystery: Decoding the Significance of a Test Page', and the third is 'Welcome to my blog'. The interface also includes a search bar, a filter button, and a 'View' dropdown menu.

Name	Content Type	Updated	Status
<input type="checkbox"/> The Art of Demonstrative Writing: An Analysis of an Article with No Define...	Blog	a few seconds ago	Published
<input type="checkbox"/> Unveiling the Mystery: Decoding the Significance of a Test Page	Blog	a minute ago	Published
<input type="checkbox"/> Welcome to my blog	Blog	2 minutes ago	Published

Three sample articles, created with the help of Chat GPT

API keys

Before we jump back into the code we need to do one more thing: create an API key. This will be used by our project in order to retrieve the content from Contentful.

Let's go to **Settings → API keys → Add API key**. Once this is done, let's take note of these credentials:

- Space ID
- Content Delivery API - access token

That's all we needed to do on the Contentful dashboard, therefore it's time to go back to our project.

Install Contentful client in the project

Contentful provides a Javascript library for both the **Content Delivery API** and the **Content Preview API**. This is great news for us because we don't need to *re-invent the wheel* for fetching our previously created content.

Let's go back to our terminal and run:

```
yarn add contentful
```

Now that the Contentful library is installed we can go to `src/app/page.tsx`, a file that was created during the installation process and that we will use as the blog listing page.

We will find a some existing code here, all related to the demo page that we saw earlier. We can clean that up and just replace the whole `main` content with a temporary "blog goes here" text.

We can also remove the existing import from `next/image` since we are not going to use that.

The whole content of `page.tsx` should now look like this:

```
/* page.tsx */
```

```
export default function Home() {
  return (
    <main className="flex min-h-screen flex-col items-center justify-between p-24">
      <p>blog listing goes here</p>
    </main>
  )
}
```

Configure the client

Now we want to set up the connection with Contentful, in order to retrieve our blog data.

To do that we need to import the library and then set up the client with the credentials that we stored earlier. Let's add this on top of the file:

```
const contentful = require("contentful");

const client = contentful.createClient({
  space: "{{put here your space id}}",
  accessToken: "{{put here your access token}}",
});
```

We can now create the utility for fetching the data. It will be outside of the `Home` functional component.

```
const getBlogEntries = async () => {
  const entries = await client.getEntries({ content_type: "blog" });
  return entries;
};
```


As you can see we used “blog” as content type. This is the name that we defined earlier when we created the content model on Contentful.

Now we need to invoke the newly created function. And since our function is asynchronous, the *Home* component needs to be asynchronous as well.

What before was:

```
export default function Home() {  
  . . . .
```

will be updated to:

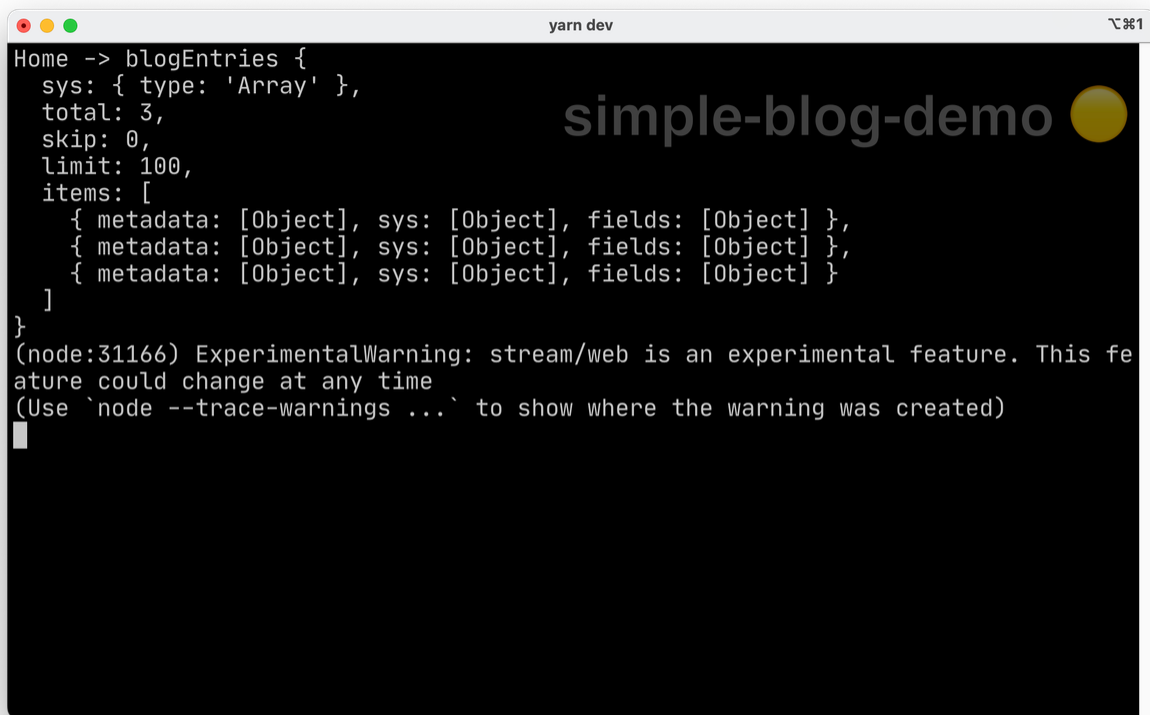
```
export default async function Home() {  
  . . .
```

Now that Home is asynchronous we can invoke the `getBlogEntries` method, just before the return statement. We also want to make sure that it’s working, so let’s add a temporary console log there:

```
const blogEntries = await getBlogEntries();  
console.log("Home -> blogEntries", blogEntries)
```

Time to check if it all works. Keep in mind that data fetching is not happening in the browser but in the Next.js local development server. As mentioned at the beginning of the article we are, in fact, taking full advantage of the Static Rendering capabilities of Next.js and pre-rendering this page at build time. Therefore our console log is not going to be shown in the browser console but in the terminal console.

```
Home -> blogEntries {
  sys: { type: 'Array' },
  total: 3,
  skip: 0,
  limit: 100,
  items: [
    { metadata: [Object], sys: [Object], fields: [Object] },
    { metadata: [Object], sys: [Object], fields: [Object] },
    { metadata: [Object], sys: [Object], fields: [Object] }
  ]
}
(node:31166) ExperimentalWarning: stream/web is an experimental feature. This fe
ature could change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
```



Our 3 sample blog posts are fetched successfully

Looks like it's working!

What about the return type?

A careful Typescript developer will have noticed that we haven't defined the return type of `getBlogEntries`. If we want to use the full potential of Typescript and minimize the risk of potential mistakes we should declare our expected return type.

First of all we need to install a new dev dependency: Since we are using *Rich Text* for the `content` field, we also need to install the related types in our project:

```
yarn add --dev @contentful/rich-text-types
```

Let's create a new file called `types.ts` inside `src/app/` and then define the types for our data structure:

```
/* types.ts */
```

```
import { Document } from '@contentful/rich-text-types';

export type BlogItem = {
  fields: {
    title: string;
    slug: string;
    date: Date;
    content: Document;
  }
}

export type BlogItems = ReadonlyArray<BlogItem>;

export type BlogQueryResult = {
  items: BlogItems;
}
```

Now we can use `BlogQueryResult` as the return type for `getBlogEntries`. Let's keep in mind that this function is asynchronous, therefore the return type will be a `Promise`:

```
const getBlogEntries = async ():Promise<BlogQueryResult> => {
  ...
}
```

The type `BlogQueryResult` needs to be imported from `types.ts`. Our IDE is likely going to add the import automatically but, if not, we need to add this on top of the file:

```
import { BlogQueryResult } from "../types";
```

Outputting the content

The connection to Contentful works and we have typed our custom data structure. It's time to output this data into the page and define the basic markup, without any styling for now.

Where we previously put the `<p>blog listing goes here</p>` placeholder is where we will do our actual iteration of content.

To do that we will use the `.map` function, like this:

```
{blogEntries.items.map((singlePost) => {  
  ....  
})}
```

Now, inside the map, let's extract the fields that we need:

```
const { slug, title, date } = singlePost.fields;
```

Finally we will output the preview of our blog post:

```
return (  
  <div key={slug}>  
    <Link href={` /articles/${slug}`}>  
      <h2>{title}</h2>  
      <span>  
        Posted on{" "}   
        {new Date(date).toLocaleDateString("en-US", {  
          year: "numeric",  
          month: "long",  
          day: "numeric",  
        })}  
      </span>  
    </Link>  
  </div>
```

```
);
```

A couple of notes on what we did here:

- We are using `slug` as the key. React in fact [requires us](#) to specify a unique key prop when we use JSX elements inside a map and slug is a unique identifier of the blog post (as defined in our Content Model as well)
- We are using the [Link](#) component from Next.js which will be needed to navigate between routes. We are pointing to a route that doesn't exist yet (`/articles/{{slug}}`), we will fix that in a bit.
- Since `date` is of type `Date` we are converting it to a readable format (like, for example, August 17, 2023) using `toLocaleDateString`. Potentially we can move this functionality to an external utility file but to keep it simple let's leave it here for now.

Our `page.tsx` should now look like this:

```
/* page.tsx */
import Link from "next/link";
import { BlogQueryResult } from "../types";

const contentful = require("contentful");
const client = contentful.createClient({
  space: "{{put here your space id}}",
  accessToken: "{{put here your access token}}",
});

const getBlogEntries = async (): Promise<BlogQueryResult> => {
  const entries = await client.getEntries({ content_type: "blog" });
  return entries;
};

export default async function Home() {
```

```

const blogEntries = await getBlogEntries();
return (
  <main className="flex min-h-screen flex-col items-center justify-between p-24">
    {blogEntries.items.map((singlePost) => {
      const { slug, title, date } = singlePost.fields;
      return (
        <div key={slug}>
          <Link href={` /articles/${slug}`}>
            <h2>{title}</h2>
            <span>
              Posted on{" "}
              {new Date(date).toLocaleDateString("en-US", {
                year: "numeric",
                month: "long",
                day: "numeric",
              })}
            </span>
          </Link>
        </div>
      );
    })}
  </main>
);
}

```

We didn't apply any custom styling so the page will now look a bit messed up. It's time to make it look nice.

Styling the listing page with Tailwind

The reason why posts are distributed in a strange way on the page is because of these 2 Tailwind classes that are assigned to the `<main>` element:

- **items-center**: equals to `align-items: center` in plain CSS.
- **justify-between**: equals to `justify-content: space-between` in plain CSS.

Since the flex direction of the element is `column` (see `flex-col` class) this means that the content is horizontally centered and vertically equally distributed on the page.

Let's get rid of those 2 classes, and leave all the other ones:

```
<main className="flex min-h-screen flex-col p-24">
```

The content will now be aligned on the top left and we can proceed with styling the individual elements.

In case our IDE is Visual Studio Code, it's highly recommended to install the [Tailwind CSS IntelliSense](#) extension. This will in fact help us with autocompleting and syntax highlighting.

As you probably noticed, the blog entries are now all collapsed, without margins. We could solve that by adding a margin (top or bottom) to the wrapper div. But, by doing so, we will have an unnecessary margin on the first (or last) item. Instead, since we are working with flex, the ideal solution is to use the `row-gap` property. We will therefore use the `gap-y-8` class from Tailwind, which equals to `row-gap: 2rem` in plain CSS.

```
<main className="flex min-h-screen flex-col p-24 gap-y-8">
```

Let's now go to the `<h2>` element, which is used for the post title. We want to make this text bold and bigger:

```
<h2 className="font-extrabold text-xl">{title}</h2>
```

- **font-extrabold**: equals to `font-weight: 800;`
- **text-xl**: equals to `font-size: 1.25rem; line-height: 1.75rem;`

The Art of Demonstrative Writing: An Analysis of an Article with No Defined Topic

Posted on August 17, 2023

Unveiling the Mystery: Decoding the Significance of a Test Page

Posted on August 16, 2023

Welcome to my blog

Posted on August 15, 2023

The current state of the blog listing page

It's already looking way better, but we want to add an extra touch to it: we want the title to change the color when hovered. If you check our DOM structure you will probably notice a problem there: the `Link` component (which will render a `<a>` tag in the page) is wrapping two elements: the `h2` title and the `span`. We ideally want the title to change color when the whole block is hovered.

In standard CSS we would use a child selector. Something like:

```
a:hover h2 {  
  color:  rgb(59 130 246);  
}
```

But how do we do that in Tailwind? This is actually quite simple. We just need to add a `group` class to the `Link` component:


```
<Link className="group" href={` /articles/${slug}`}>
  ...
```

and then, in the `h2` definition we will add this class:

```
group-hover:text-blue-500
```

In this way we are telling our title element to change color when its parent element is hovered.

We also want a smooth hover transition, therefore let's add the `transition-colors` class as well.

The `h2` element should now look like this:

```
<h2 className="font-extrabold text-xl group-hover:text-blue-500 transition-colors">
  {title}
</h2>
```

The Art of Demonstrative Writing: An Analysis of an Article with No Defined Topic

Posted on August 17, 2023

Unveiling the Mystery: Decoding the Significance of a Test Page

Posted on August 16, 2023

Welcome to my blog

Posted on August 15, 2023

Current state, including hover effect

Creation of the article page

We successfully created our Blog Listing page. Now it's time to create the article page. As we saw earlier, when using the `Link` component, we want the page to be available at this path:

```
/articles/{article slug}
```

In order to do this, we need to use a feature of Next.js called [Dynamic Routes](#).

Let's create a new file at this location: `src/app/articles/[slug]/page.tsx`. By using the square brackets convention on the folder name we are creating a **Dynamic Segment**. Our component will receive a prop called `slug` which will contain the dynamic part of the route.

Let's now create a very basic output on this page:

```
/* src/app/articles/[slug]/page.tsx */
type BlogPageProps = {
  params: {
    slug: string;
  };
};

export default async function BlogPage(props: BlogPageProps) {
  const { params } = props;
  const { slug } = params;
  return (
    <main className="min-h-screen p-24 flex justify-center">
      <div className="max-w-2xl">
        <h1>you are in {slug}</h1>
      </div>
    </main>
  );
}
```

```
);  
}
```

And now let's navigate to <http://localhost:3000/articles/this-is-a-test>

As we can see, `slug` is received as part of the props, and is therefore dynamic based on the route. In this specific case it will be: "this-is-a-test".

However, since this is a server component, we have an issue: as long as we are in development mode, every route we will use (regardless if the slug exists or not) will work. However, at build time, Next.js will not know which routes should be created, and therefore none of these pages will be generated at build time.

generateStaticParams

This is where the `generateStaticParams` *magic* happens. This Next.js function, used in combination with dynamic routes, allows us to statically generate routes at build time:

```
export async function generateStaticParams() {  
  const queryOptions = {  
    content_type: "blog",  
    select: "fields.slug",  
  };  
  const articles = (await client.getEntries(queryOptions)) as BlogQueryResult;  
  return articles.items.map((article) => ({  
    slug: article.fields.slug,  
  }));  
}
```

We can test if this works as intended by running `yarn build` and then checking the content of the build folder.

```
chunks/596-c294a7d39d9fe754.js      26.1 kB
chunks/fd9d1056-a99b58d3cc150217.js  50.5 kB
chunks/main-app-67d23e2457dac531.js  219 B
chunks/webpack-ddd118fa46ddd176.js    1.64 kB

Route (pages)                          Size      First Load JS
- ○ /404                               182 B      76.4 kB
+ First Load JS shared by all          76.3 kB
  | chunks/framework-8883d1e9be70c3da.js  45.1 kB
  | chunks/main-d7299844fa56b5c7.js      29.4 kB
  | chunks/pages/_app-52924524f99094ab.js  195 B
  | chunks/webpack-ddd118fa46ddd176.js    1.64 kB
  ○ (Static) automatically rendered as static HTML (uses no initial props)
  ● (SSG) automatically generated as static HTML + JSON (uses getStaticProps)

✨ Done in 14.39s.
➔ simple-blog-demo git:(main) X cd .next/server/app/articles
➔ articles git:(main) X ls
[slug]                                the-art-of-demonstrative-writing.meta
significance-of-a-test-page.html      the-art-of-demonstrative-writing.rsc
significance-of-a-test-page.meta       welcome-to-my-blog.html
significance-of-a-test-page.rsc        welcome-to-my-blog.meta
the-art-of-demonstrative-writing.html  welcome-to-my-blog.rsc
➔ articles git:(main) X
```

The build folder contains an HTML file for each of the articles

Back to the project, we now need a function for fetching the content of a single blog post, based on the slug.

```
const fetchBlogPost = async (slug: string): Promise<BlogItem>
=> {
  const queryOptions = {
    content_type: "blog",
    "fields.slug[match]": slug,
  };
  const queryResult = (await client.getEntries(
    queryOptions
  )) as BlogQueryResult;
  return queryResult.items[0];
};
```

Now that we have `fetchBlogPost` we can use it before our return statement.

```
const article = await fetchBlogPost(slug);  
const { title, date, content } = article.fields;
```

And we can now replace the content of the `h1` tag, to show the actual article title:

```
<h1>{title}</h1>
```

Now we need to render the content. Since we used the Rich Text format, this data is not received as a string but as a structured object. Luckily for us, Contentful is already providing a utility for transforming this data structure into actual JSX.

Let's install it in the project:

```
yarn add @contentful/rich-text-react-renderer
```

and import it at the beginning of the file:

```
import { documentToReactComponents } from '@contentful/rich-text-react-renderer';
```

We can now use this to render the article content, right below the H1 title:

```
{ documentToReactComponents(content) }
```

Some final touches

The H1 element is currently unstyled, so it doesn't stand out from the rest of the article.

Let's add some Tailwind classes to make it look better:

```
<h1 className="font-extrabold text-3xl mb-2">{title}</h1>
```

We also are not rendering the post date at the moment. Let's do that, similarly to what we did on the listing page:

```
<p className="mb-6 text-slate-400 ">
  Posted on{" "}
  {new Date(date).toLocaleDateString("en-US", {
    year: "numeric",
    month: "long",
    day: "numeric",
  })}
</p>
```

As you probably noticed, the article content doesn't have any formatting, since Tailwind is resetting all the browser default styles.

Obviously, we can't use inline styles here, because the html elements (h2, p, etc.) are rendered by `documentToReactComponents`. However, Tailwind also allows us to target the children of an element with a special selector. Let's wrap our content in a div and use this syntax:

```
<div className="[&>p]:mb-8 [&>h2]:font-extrabold">
  { documentToReactComponents(content) }
</div>
```

As you can see we are targeting the children and assigning a margin-bottom to all `p` elements and making all the `h2` elements extra bold.

Finally let's have a look at our article page:

The Art of Demonstrative Writing: An Analysis of an Article with No Defined Topic

Posted on August 17, 2023

Welcome to a literary journey that defies convention and embraces the boundless world of demonstrative writing. In this unique exploration, we will dissect an article that dares to step into the realm of the undefined, challenging the very essence of what we expect from the written word.

Breaking Free from Conventions

In a world where articles, essays, and blogs often conform to rigid structures and predefined topics, we present an experiment in creative expression. This article, or rather, this canvas of words, embarks on a liberating quest—free from the constraints of traditional subject matter.

The Power of Words Unleashed

As we navigate through this uncharted territory of writing, we will uncover the immense power that words possess when set free from the confines of a predetermined theme. Demonstrative writing invites us to observe how language itself can shape meaning, leaving room for interpretation and imagination to take the reins.

Embracing Ambiguity and Interpretation

Prepare to embrace ambiguity and relish the beauty of interpretation. In this analysis, we will explore how an article without a defined topic invites readers to participate actively in the process of meaning-making. Each word becomes a brushstroke on the canvas of your mind, allowing you to paint your own mental picture.

A Journey of Discovery

Our journey will take us through the intricacies of demonstrative writing, discussing the techniques and stylistic choices that transform an ordinary article into an extraordinary piece of art. We'll delve into the author's intention, the reader's role, and the

Time to deploy!

First of all, let's push our code in a new repository on GitHub. Once this is done we can head over to <https://vercel.com/new> (If we don't have an account we can sign up with GitHub).

We will be asked to install the GitHub application, in order to be able to choose our repository. Once this is done we should be able to see the repository in the dropdown:

Import Git Repository



submedia



Search...



simple-blog-demo  · 20m ago

Import

Missing Git repository? [Adjust GitHub App Permissions →](#)

After the import is done all we have to do is to click **Deploy**, in the Configure Project panel.

Configure Project

Project Name

simple-blog-demo

Framework Preset



Next.js



Root Directory

./

Edit

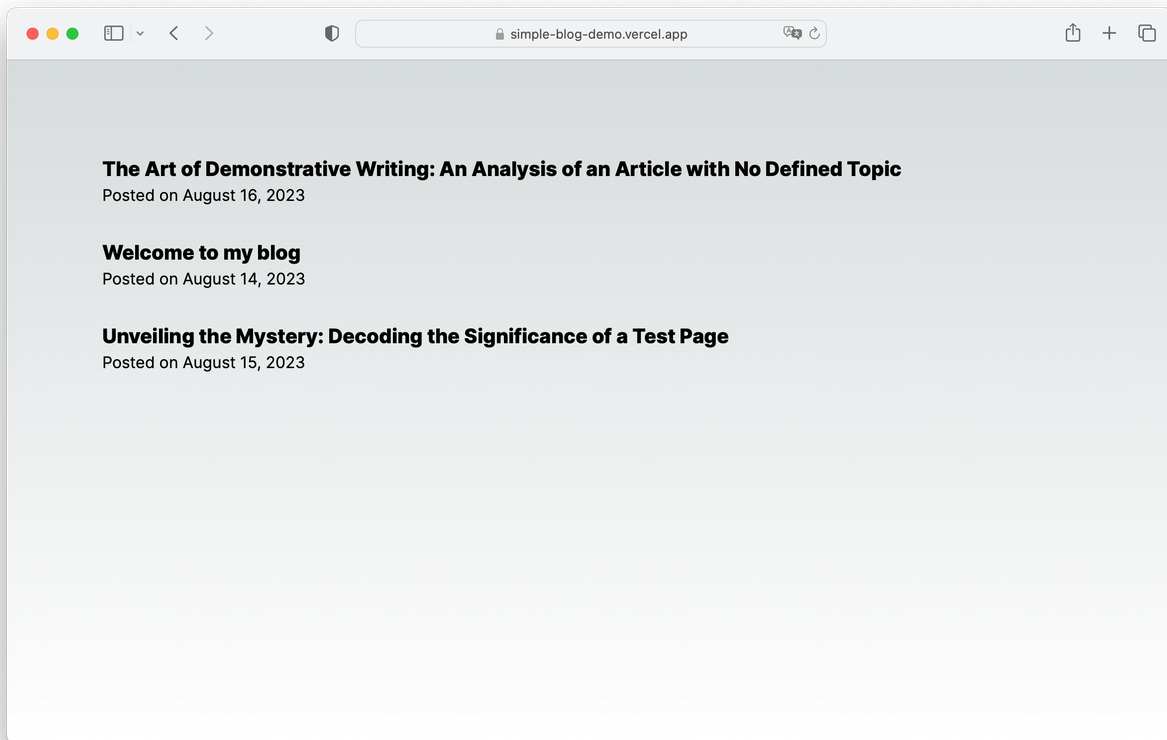
> Build and Output Settings

> Environment Variables

Deploy

If everything worked as expected, we should see a “**Congratulations! You just deployed a new Project to Vercel**” message.

We can navigate to the preview URL and see our demo blog in action:



The live version of our blog

One more thing...

Our blog is now deployed and it's entirely statically generated at build time, giving us the best possible page performance. But what happens if we edit some content or create a new article?

With the current implementation, we will need to go to the Vercel dashboard and trigger a re-deploy of the application. A new build will, in fact, fetch again the content from Contentful API and therefore use the updated content in the new deploy. However there's a simple procedure to automate this process so that we don't have to worry anymore about manual deploys.

Setting up a Webhook

In the Vercel dashboard, let's go to **Settings → Git**.

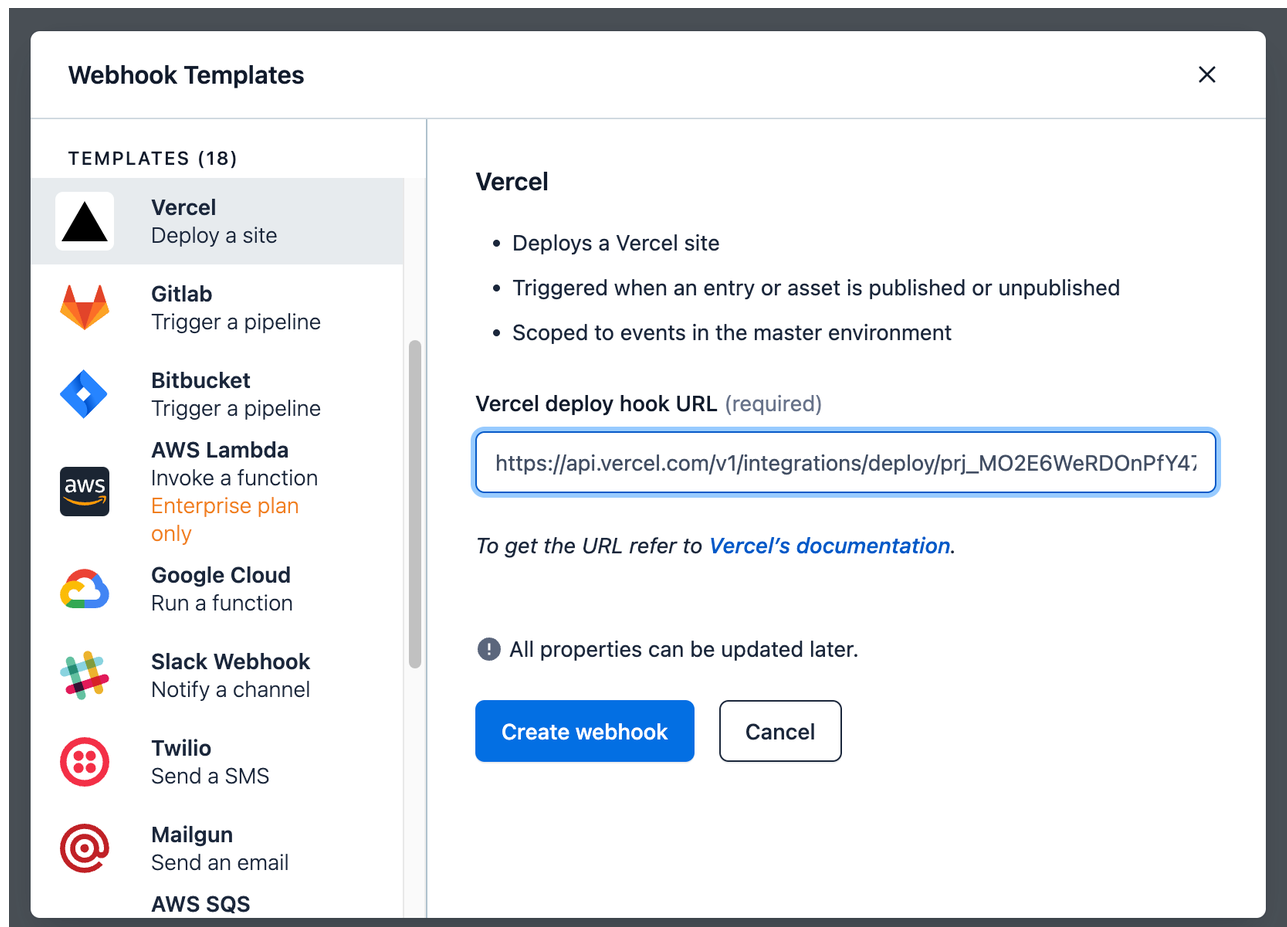
Now let's scroll to **Deploy Hooks**. Here we can create a new Hook and give it a custom name. We also need to specify a branch ("main" in my case).

When the hook is created we will get a link. Let's copy it, we will need it in a bit.

Let's log again in to the Contentful dashboard. Then **Settings → Webhooks**.

Here we can set up a new Webhook connection. Contentful provides a Webhook template for Vercel (in the right sidebar). Let's find it and then click "Add".

The modal will ask for the **Vercel deploy hook URL**. Let's paste the link that we previously copied.



Webhook Templates ×

TEMPLATES (18)

- Vercel**
Deploy a site
- Gitlab**
Trigger a pipeline
- Bitbucket**
Trigger a pipeline
- AWS Lambda**
Invoke a function
Enterprise plan only
- Google Cloud**
Run a function
- Slack Webhook**
Notify a channel
- Twilio**
Send a SMS
- Mailgun**
Send an email
- AWS SQS**

Vercel

- Deploys a Vercel site
- Triggered when an entry or asset is published or unpublished
- Scoped to events in the master environment

Vercel deploy hook URL (required)

`https://api.vercel.com/v1/integrations/deploy/prj_MO2E6WeRDOnPfY4;`

To get the URL refer to [Vercel's documentation](#).

ⓘ All properties can be updated later.

Create webhook **Cancel**

The automated deploy Webhook is now created. From this moment on, any published change on Contentful will automatically trigger a new deploy on Vercel. We can also check the history of Webhook calls in the Contentful dashboard, by going to **Settings → Webhooks**.

Conclusion

Our minimal blog is now live and functional. If you made it until here, you now have a basic but solid understanding on how to create a project from scratch with **Next.js + Tailwind + Contentful**.

My recommendation, in order to extend our knowledge and confidence with these frameworks, is to start our own project (not necessarily a blog, it can be anything) and use this tutorial as a foundation. Thanks for reading!