netlify + Jamstack + NEXT.JS

Deploying Next.js on the Jamstack

The Definitive Guide

Table of Contents

PART I

How Next.js Became a Top Jamstack Framework

How Next.js Became a Top Jamstack Framework

Next.js has become the most popular framework for building React apps. It's no surprise, then, that we're seeing more and more Jamstack applications built on Next.js.

If you're a long-time Next.js user and had read those sentences in 2016, you'd probably have shaken your head in disbelief.

That's because when Next.js initially came onto the scene, it wasn't very compatible with a core Jamstack principle: **pre-rendering statically generated sites.** Prerendering demands that the entire frontend is prebuilt into highly optimized static pages and assets during a build process. It's what makes Jamstack applications speedy and secure, since prebuilt sites can be served directly from a CDN, reducing the cost, complexity, and risk of dynamic servers as critical infrastructure. With this pattern, any server-side processes are abstracted to microservices—everything runs in the browser. Next.js could be used for Jamstack applications, but it wasn't a very popular use case.

Next.js brought **server-side rendering to React**, which is primarily a client-side rendering library. So initially, Next.js was exciting because it did the opposite—it runs the initial view on the server. Early Next.js users, first and foremost, were excited because rendering components on the server side in React, leads to better performance for users, and massive SEO improvements over client-side rendered applications.

These two data-fetching strategies—the Jamstack's preferred method of static site generation (SSG) and Next.js' method of server side rendering (SSR), are seemingly opposite approaches to building applications. We'll go into the details a bit later in this guide.

But in March 2020, Next.js 9.3 introduced better support for building static sites as well. So now if you were to use Next.js, you could build both a server-side rendering application and a static site generated application and get the best of both worlds. **You can finally pre-render, one of the core principles behind the Jamstack**. And as of 2021, Next.js has introduced support for hybrid models, allowing you to SSG when you want to and SSR when you want to.

So: what does the hybrid model mean for building Jamstack applications on Next.js? Can we get the benefits of the Jamstack and the benefits of Next.js?

In short, yes. That said, there's a bunch of different ways to approach building a Jamstack application with Next.js, with different tradeoffs to keep in mind. In this guide, we'll cover important aspects of Next.js for Jamstack developers, and provide resources (and click-to-deploy starter projects) that'll help you get the most of Next.js and the Jamstack .

PART 2

What you need to know about Next.js

What you need to know about Next.js

Now, first of all, what features should you know about before you start writing any code at all? Why are people choosing it, and what makes it different from other JS frameworks or React Native? Here are some of the most important differentiators:

It's a hybrid framework

Next.js can do both server-side rendering (SSR) and static site generation (SSG), or even both within the same project. This works because some parts of Next.js live in the browser, and some parts live in Node.js.

Depending on project needs, you can either leverage a hybrid approach for generating content or export a completely static site. The hybrid approach allows a choice between server-side rendering or static generation on a per-page basis. Meanwhile, the `next export` command generates a completely static site for deployment.

See SSR and SSG in Action

In this example demo, you'll be able to see how Next.js lets you query an API at runtime and buildtime with NASA's API, and what the load time differences are.

The routing structure is filesystem-based

One of the most important concepts to understand is how routing works in Next.js. If you're not accustomed to it, it might take a bit of getting used to.

In Next.js, the entire routing structure is filesystem-based. Whenever you put a JavaScript file in the pages/ directory, it is automatically a route, no configuration needed. If you add a contact.js file, that will become yourwebsite.com/contact.

```
File                              Route
----                              -----
/pages/contact.js                 /contact
/pages/example/project1.js        /example/projects1
/pages/index.js                   /
```

It makes adding pages nice and intuitive, and also allows for dynamic routes as well (as in, ones with variable names, check out an example here). If we made an ilovenetlify.js file in the pages directory, we could go to `localhost:3000/ilovenetlify` and it would just work. But what about when we have blog posts that have dynamic titles, and we want to name our routes after those titles? Next has a solution for you: dynamic paths.

See Dynamic Routes in Action

Go ahead and make a file in `pages/post` called `[postname].js` The square brackets in the file name indicate to Next that you have a dynamic route. The file path `pages/post/[postname].js` will end up being `localhost:3000/post/:postname` the browser! That's right, we have both a nested route, and a dynamic route here.

```
File                            Route
-----                           ------
/pages/posts/[postname].js      /posts/1
                                /posts/abc
                                /posts/hello-world
```

You can also do shallow routing in Next.js. Shallow routing is when you change your website's URL without re-running data fetching methods again. In the case of Next.js, it means you have one page component that covers multiple URLs. This is particularly useful for adding query strings, and routes that might explain the content of your pages as they change to user behavior.

If you'd like to use shallow routing in your applications, you don't do shallow routing with the `<Link>` component built into the framework, which is what you might expect. Use the built-in `useRouter` hook, and add `{shallow:true}` to your `router.push` commands:

```js
useEffect(() => {
  router.push('/some-other-route', undefined, { shallow: true })
}, [someVariable])
```

See Shallow Routing in Action

In this this story-based Choose Your Own Adventure app, you can navigate through the story **without** querying the database multiple times for new characters to be a part of it!

The code for the routing is on GitHub, and you can see how the entire project was built in this demo video.

Along with regular pages, you can set up API routing as well. With the pages/api/ directory, any file you put in there will be treated as an API endpoint.

It comes with lots of developer experience goodies

Next.js comes with a number of nice-to-have features right out of the box that make the development experience great.

- **Hot code reloading:** As you edit your code, it will automatically be updated in the browser. What's really cool about Next's hot code reloading is that it uses React Fast Refesh. React Fast Refresh allows you to update your code, and Next.js will maintain the state of your application as your code is reloading.
- **Styling options:** There are a ton of styling options built into Next.js. You don't have to do any special configuration at all if you want to use SASS, Less, or any other pre-processors. Configuring global styles in Next.js is relatively easy as well, but takes some configuration. And if you want to use CSS modules or some other CSS and JS library, it just works.
- **TypeScript support:** In addition to styling options, there's also built-in language options. If you don't want to write JavaScript, TypeScript support just works out of the box with Next.js.
- **Automatic code splitting:** Automatic code splitting prevents bloated applications. If you're building up your application and there are some functions that you use a ton, and some functions that you don't use at all, Next.js will cut out all the unused functions from your build.

Again, none of these nice-to-haves require any configuration, so you can get up and running really easily.

About the API

There are a few components and hooks that Next.js provides for devs to work with:

- `next/link` Enables client-side transitions between routes
- `next/head` Allows developers to append elements to the <head> of the rendered page
- `next/router` A hook for accessing the router object inside components on your application
- `next/amp` Allows you to create AMP pages.

About the CLI

The Next.js command line interface is nice and simple to use. It allows you to:

- Create optimized production builds (`next build`)
- Run the application in development mode (`next dev`)
- Run the application in production mode after building (`next start`)
- Export your Next.js app as a static site (`next export`)

Preview mode allows for quick builds in staging environments

Static generation is awesome when you're fetching data from a CMS and pre-building everything, but sometimes you want to be able to view what your content will look like before running a full build of your website. Next.js' Preview Mode feature solves that problem.

Preview Mode allows users to bypass the statically generated page to server-side render a draft page from any data fetching solution. This means you don't have to wait for a build to run to see a preview of what a new piece of content might look like! This is ideal for if you're using a CMS solution for your sites and your team wants to see what their changes might look like before committing them, but you don't want to wait for a full build to run.

***Note:** Preview Mode on Netlify is fully supported via the Essential Next.js plugin. The plugin, which auto-installs upon detecting a Next.js app, wraps your application in a tiny compatibility layer, so that pages can use Netlify Functions to be server-side rendered.*

Custom configuration is pretty easy

If you want to customize more advanced behavior in your application, you can create a next.config.js file (which is really a regular Node.js module). That config file allows you to customize all sorts of things, like adding support for custom page extensions, indicating static optimization benefits, and setting up custom build directories.

PART 3

When is Next.js not a good choice?

When is Next.js not a good choice?

The previous section covered areas where Next.js just works out of the box. But there are some use cases where configuring Next.js takes some upfront configuration, or a deeper understanding of the architecture under the hood.

Understanding logging in Next.js can be tricky

Next.js is a hybrid framework. Some parts of it live in the browser, and some parts live in Node.js. When you're developing, that can make for some fairly confusing development, sometimes.

Depending on where you put your console.log() statements, your logs will appear in different spots. Here's some rules of thumb for figuring out where they are:

If your logs are declared in a function that renders or uses React, they will appear in the browser. Whether it's one of your React hooks, your React components, or a page-level component, that log will be in your browser console. This part of your code is the frontend of your application, so you can remember this: the frontend is client-side, in the browser! This is true for both development and production mode.

If your logs are in a data fetching function, they will appear in your terminal or build/function logs. If you have utilities, call APIs, or render certain routes based on external data, these logs will be in your terminal in development mode. Your terminal is where the "back-end" of your application lives, and where you can see pages being built. Under the hood, it's a Node.js environment. In production mode, these logs will appear at build time in your build logs at build time, or in your function/API logs at runtime.

Building Large Sites on Next.js

One of the reasons Jamstack is so popular is because it delivers performance benefits at scale. When you pre-render as much content as possible up front and serve that content via a global multi-redundant edge network, your site visitors get fast page load times and great experiences.

But as Jamstack architecture continues to grow in popularity, we're seeing developers build increasingly large websites on Netlify. And when you put hundreds of thousands of assets through a sequential build process, things start to get a little dicey. The pre-rendering step becomes a bottleneck, leaving development teams waiting around for builds to complete. Ultimately, this slows down the speed of iteration.

So, how can developers with huge sites get around long build times?

Until now, if you wanted to significantly reduce your build time, you could either break your site into several microsites (sharding) or you could rely on framework-specific caching constructs.

What's ISR and when should I use it?

For Next.js, that caching construct is called Incremental Static Regeneration, or ISR. ISR seeks to extend the power of static sites by adding some server-side rendering (SSR) goodies on top. When you deploy a Next.js site with ISR enabled, you deploy a (mostly) static site. You have your pre-defined static pages that were built, and you have routes on your application that aren't built until your users hit those pages

Typically, when you have a server-side rendered (SSR) page that is one of these unbuilt pages, your users have to wait for the page to be built and served all at once. But in the case of ISR, if your users hit that route, they get a fallback page.

A fallback page is a placeholder for the actual content that will be on that page, and you can have skeleton components in place until data is built and loaded. Once that data has been resolved, that page is cached, added to the rest of the site's bundle, and the next user of your page will see the built page. If the data needs to update, the user will see that cached version instead of the fallback, and the site can set a revalidate timeline so that it can revalidate and update data regularly when your users hit the page.

Each of the new blocks in this diagram is a new page that is built at runtime and added to the "stack."

This method of serving pages is using the stale-while-revalidate caching strategy. It's pretty performant, because you can (nearly) get the performance benefits of a pure static page, with the power of new dynamic data like you would in SSR. That's why this strategy is very often called "hybrid" development, because it combines the best of two approaches.

When to Avoid ISR

There are a few flaws in ISR that you might want to consider before going all-in on the concept.

When you have a user come to your page, you want them to see the most up-to-date version, immediately. With ISR the first visitor to a page will not see that. They will always see a fallback first. And then later, if the data gets stale, the first visitor to see that cached page will see the out-of-date data first before it revalidates. Once again, this inconsistent experience can be pretty difficult to debug if your users experience negative side-effects as a result of old/unbuilt pages.

In addition to difficult debugging, ISR violates two core Jamstack principles: atomic commits and immutable deploys. Let's review what those are before going further:

- **Atomic commits:** Make updates available only when they are complete and totally in place.
- **Immutable deployment:** Guarantee the integrity of previous deploys by insulating them from future actions.

The easy roll-back nature of Netlify deploys? That's due to immutable deploys. Dev-server-like preview links for your whole site? That's due to atomic commits.

Thanks to the fundamental principles of atomic and immutable deploys, development teams can have confidence that their sites are being served as intended. They can be confident that no deployment will ever be accessed while partially completed. And they can be confident that no deployment to their site can be destructive or jeopardize the ability to revert to any previous version of the site, all of which are ready to serve at a moment's notice and with a couple of clicks.

ISR breaks that model. By adding extra pages to your bundle, rollbacks can no longer be instant, and you no longer have that single new version of your site when you update your content.

Let's say you build a site that has a bunch of products for sale, and each of those products are on ISR'd pages. In an ideal scenario, your users can navigate to a products' page, see a product for sale, and buy it. The next users who go to the page will see it, and the page might update to show that the product is out of stock.

If you rollback your site to a different deploy, because your page is cached separately from the bundle, it could exist in a different state for your user than expected. It could be the old version of the site, the new version, or some funky in-between cached version trying to revalidate itself. And unfortunately, debugging this is difficult, because different users (and members of your dev team) would see different pages, and it might be difficult to duplicate.

Clearly there are benefits to ISR, but it does come with caveats. Make sure to weigh the pros and the cons and decide for yourself if it's right for you.

On-Demand Builders (ODB): A Jamstack methodology for faster builds

On-demand Builders is a brand new approach from Netlify that enables you to incrementally build your site and is designed to work with any framework, not just Next.js. On-demand Builders allow you to build a page on demand once, then serve the cached result to subsequent requests. This is similar to how server-based applications work, but without the complexity of configuring Varnish or managing caching rules. Your deployments retain the straightforward deployment strategy and easy rollbacks made possible by adopting a Jamstack architecture — all while gaining a powerful new tool for deploying large sites faster.

With On-demand Builders you can split your site's assets into two groups:

- **Critical content:** Content that is automatically compiled and deployed as part of your traditional Netlify build process.
- **Deferred content:** Content that is built using an on-demand builder when a site visitor requests it for the first time. This content is then cached at the edge, so it's available to load more quickly for subsequent visitors.

On-demand Builders are currently available in early access with documented constraints. We're enthusiastic about the future of this solution, here's why:

First, *shorter builds mean your whole team can be more productive and iterate faster.*

Second*, On-demand Builders are flexible enough to work across multiple frameworks. We've already proven the benefits with Eleventy and are looking forward to seeing examples from the Nuxt community.*

Last but not least*, we're excited about what On-demand Builders mean for Next.js on Netlify. In the past, Netlify has supported Incremental Static Regeneration (ISR) for Next.js, but there were notable performance trade-offs. With On-demand Builders, Next.js developers can now take advantage of caching for better performance.*

See On-Demand Builders in Action

On-demand Builders are configured using Netlify's serverless functions. You can write a function that generates and returns the desired content, and pass this function as a parameter to the builder() method provided as part of the @netlify/functions package.

```
const { builder } = require('@netlify/functions');
async function myfunction(event, context) {
   // logic to generate the required content
}

exports.handler = builder(myfunction);
```

Similar to Netlify Functions, the builder will automatically receive an endpoint relative to the base URL of your site. This endpoint can be called directly or by redirecting traffic from another URL. The first call to the builder will invoke the function and return the generated page. All subsequent calls will return the cached page until the next refresh.

Learn more and see code samples in the docs.

PART 4

In the Wild: Next.js and the Jamstack

In the Wild: Next.js and the Jamstack

Looking for real-life examples of Jamstack applications built with Next.js? We've got you covered. Here's a couple from the Netlify vault:

Why SEO company Backlinko chose Next.js and Netlify to build a faster website

Backlinko is one of the world's top websites for SEO tips, advice, and training. One of those SEO tips is, of course, to have a blazing fast website. That's why Backlinko engaged agency Bejamas to help migrate their monolithic WordPress application over to the Jamstack.

The final results were pretty outstanding. The team managed to improve almost every performance metric without losing any of the websites' functionalities, including Preview Mode, which content editors required. With Next.js on Netlify, the page loading speed is now 3x faster than before the migration.

Read the case study →

How Jamstack Explorers Was Made: Next.js

Jamstack Explorers is a free, open-source learning platform built by the Netlify Developer Experience team. It's packed with video-based lessons to teach students about the Jamstack and surrounding technologies. And it's built with Next.js!

Read more about how the DX team built Jamstack Explorers in just a few months, including all the project management, UX/UI tools, and image optimizers they used along the way.

Read the case study →

PART 5

Getting Started with Next.js on Netlify

Getting Started with Next.js on Netlify

By deploying your Next.js app on Netlify, you get the best of both worlds. You have all the benefits of Netlify — including instant rollbacks, Netlify Identity, continuous deployment whenever you push to your repo, and much more — as well as the benefits of Next.js like Preview Mode, dynamic routing, and API routes.

Even better? Netlify's Essential Next.js plugin requires no installation. Just deploy to Netlify. Netlify auto-detects usage of a Next.js framework and enables support for:

- **Preview Mode:** Server-side render a live preview of a web app to share, gather feedback and make changes in real-time without ever requiring a build.
- **Next.js 10:** Key elements of Next.js 10 and up are enabled on Netlify including internationalized routing and React 17 support.
- **Atomic and immutable deploys:** Full compatibility with incremental static regeneration and server-side rendering techniques by serving via Netlify Functions.
- **On-demand Builders:** The Essential Next.js plugin introduces automatic support for dynamic page rendering (commonly referred to as SSR or ISR).

Ready to get started? Here's how to build a Next.js site on Netlify:

Step 1: Put Your Next.js Site Code in a Git Repo

The magic of Netlify is that every push to the main branch of your repo triggers a new build and deploy of your site. The first step to building a Next on Netlify app, then, is to get your site code into a repo. Netlify supports GitHub, GitLab, or Bitbucket.

If you're migrating an existing Next.js site to Netlify, we've got a guide for that as well.

Don't have a Next.js site yet? Check out this minimal starter project to start experimenting.

This is a Next.js v10.2 project bootstrapped with create-next-app and set up to be instantly deployed to Netlify. This project is a very minimal starter that includes 2 sample components, a global stylesheet, a `netlify.toml` for deployment, and a `jsconfig.json` for setting up absolute imports and aliases.

Step 2: Create a New Netlify Site From Your Next.js Site Repo

Now that our code is available in a repo, we can **configure Netlify to deploy our Next.js site any time a new commit is pushed.**

This is possible in two ways: using the Netlify CLI, and using the Netlify app UI.

Option 1: Use the Netlify CLI

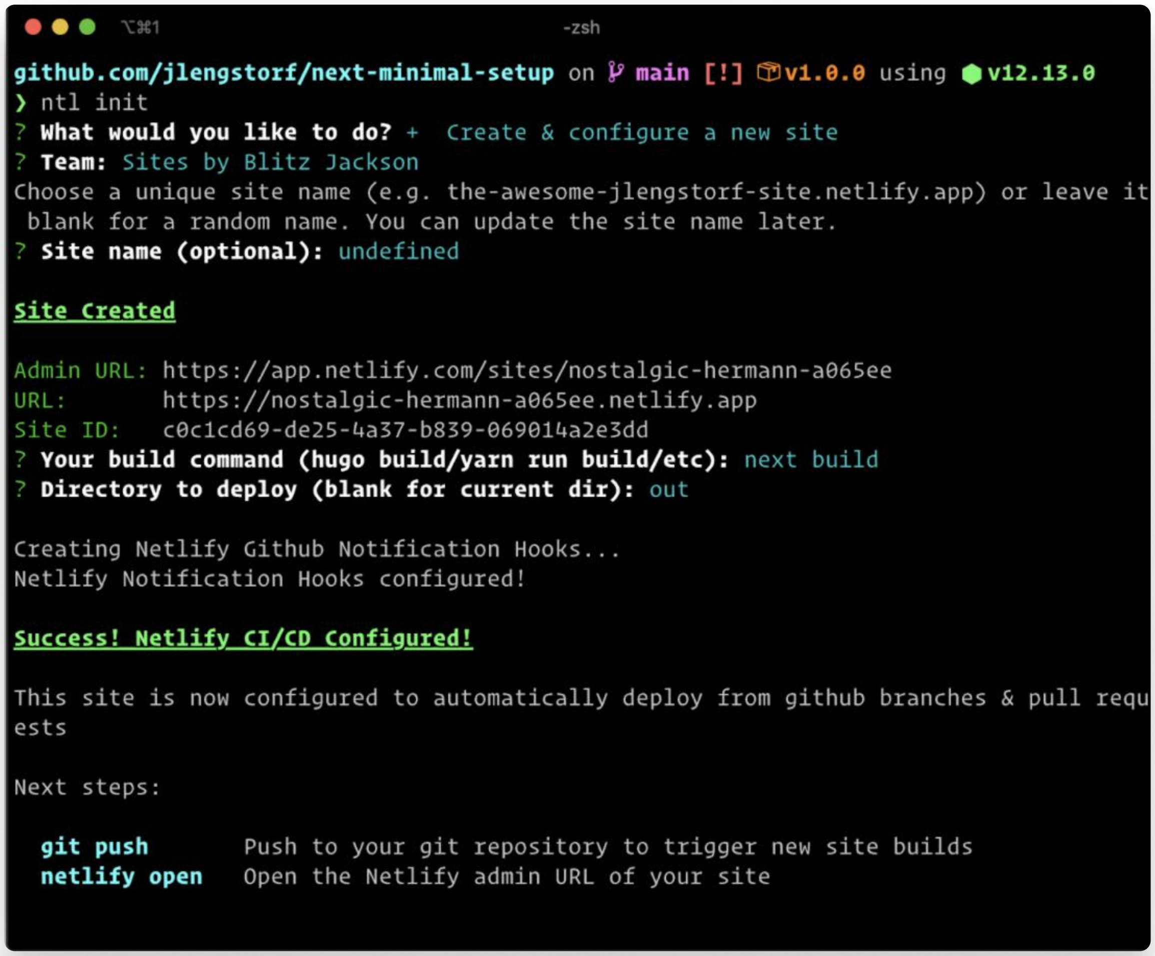
Netlify provides a powerful CLI that allows you to interact with your account. This means you can **deploy a Next.js site to Netlify without ever leaving the command line!** I love this option because it's fast and I can do it at the same time as I push the repo code to GitHub for the first time.

If you don't already have it, install the Netlify CLI globally and log in:

```
# install the Netlify CLI globally
npm install -g netlify-cli
# log into your Netlify account
ntl login
```

Next, navigate to the site's root directory and run `ntl init`:

```
# move into the site's root directory
cd ~/path/to/your/nextjs-site/
# create a new Netlify site
ntl init
```



```
github.com/jlengstorf/next-minimal-setup on main [!] v1.0.0 using v12.13.0
❯ ntl init
? What would you like to do? +  Create & configure a new site
? Team: Sites by Blitz Jackson
Choose a unique site name (e.g. the-awesome-jlengstorf-site.netlify.app) or leave it
 blank for a random name. You can update the site name later.
? Site name (optional): undefined

Site Created

Admin URL: https://app.netlify.com/sites/nostalgic-hermann-a065ee
URL:       https://nostalgic-hermann-a065ee.netlify.app
Site ID:   c0c1cd69-de25-4a37-b839-069014a2e3dd
? Your build command (hugo build/yarn run build/etc): next build
? Directory to deploy (blank for current dir): out

Creating Netlify Github Notification Hooks...
Netlify Notification Hooks configured!

Success! Netlify CI/CD Configured!

This site is now configured to automatically deploy from github branches & pull requ
ests

Next steps:

  git push       Push to your git repository to trigger new site builds
  netlify open   Open the Netlify admin URL of your site
```

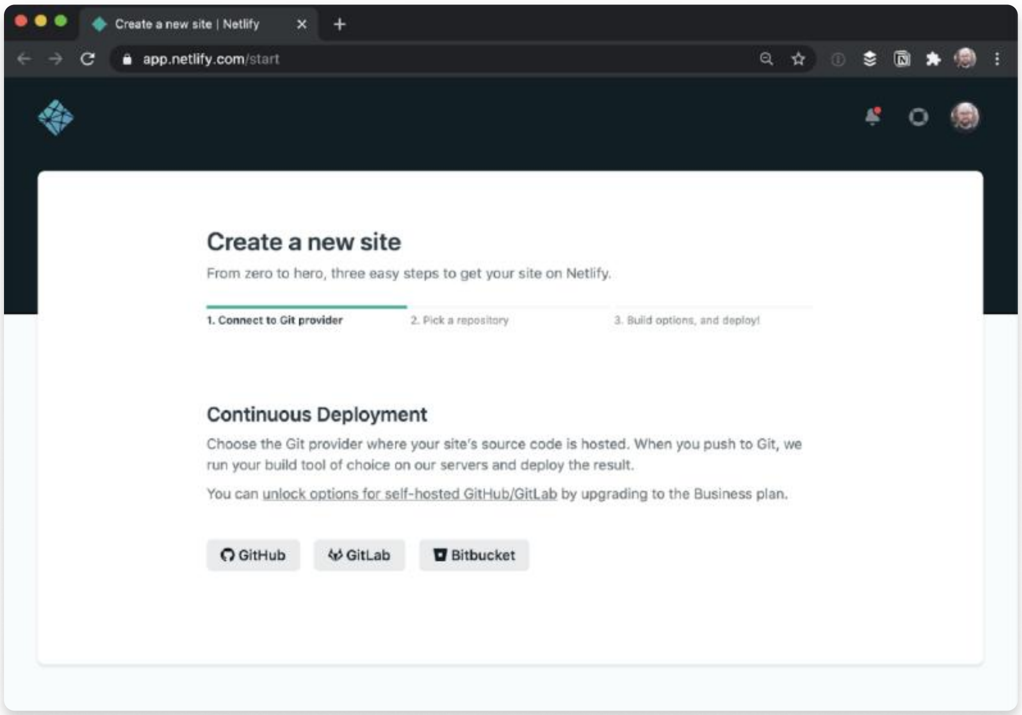
The CLI will guide you through configuring your new site:

- Choose “Create & configure a new site”
- Use the arrow keys to choose the Netlify team you want to deploy to
- Choose a name for your site or leave it blank to have one auto-generated (you can change this later)
- Set the build command to `next build`
- For the directory to deploy, enter `out`

  Heads up! The directory to deploy can be anything you want except the current directory `` `(.)` `` Using the current directory would also publish your source code publicly.

Option 2: Use the Netlify UI

If you prefer not to use the command line, you can **create a new Netlify site using the app UI**. First, visit https://app.netlify.com/start in your browser.


Create a new site | Netlify
app.netlify.com/start
1. Connect to Git provider
2. Pick a repository
3. Build options, and deploy!
GitHub
GitLab
Bitbucket

Create a new site

From zero to hero, three easy steps to get your site on Netlify.

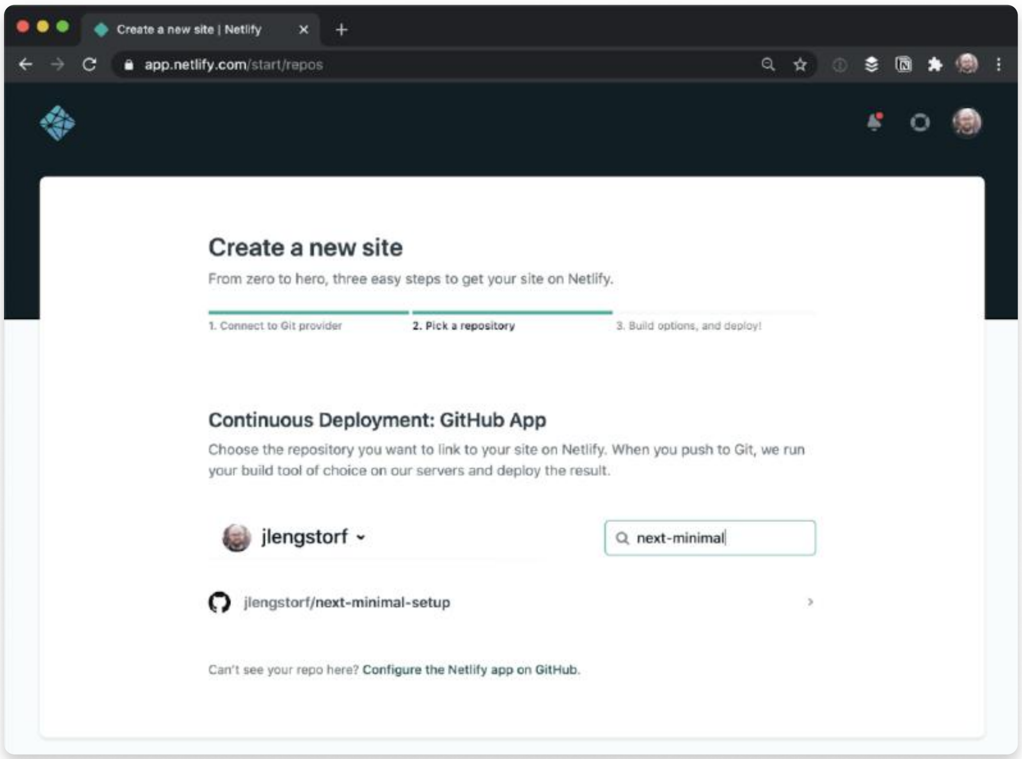
Continuous Deployment

Choose the Git provider where your site's source code is hosted. When you push to Git, we run your build tool of choice on our servers and deploy the result.

You can unlock options for self-hosted GitHub/GitLab by upgrading to the Business plan.

Choose your preferred Git provider from the options and authorize Netlify to access the repo you want to deploy.

In the next screen, select your Next.js repo from the list.



Create a new site

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider
2. Pick a repository
3. Build options, and deploy!

Continuous Deployment: GitHub App

Choose the repository you want to link to your site on Netlify. When you push to Git, we run your build tool of choice on our servers and deploy the result.

jlengstorf

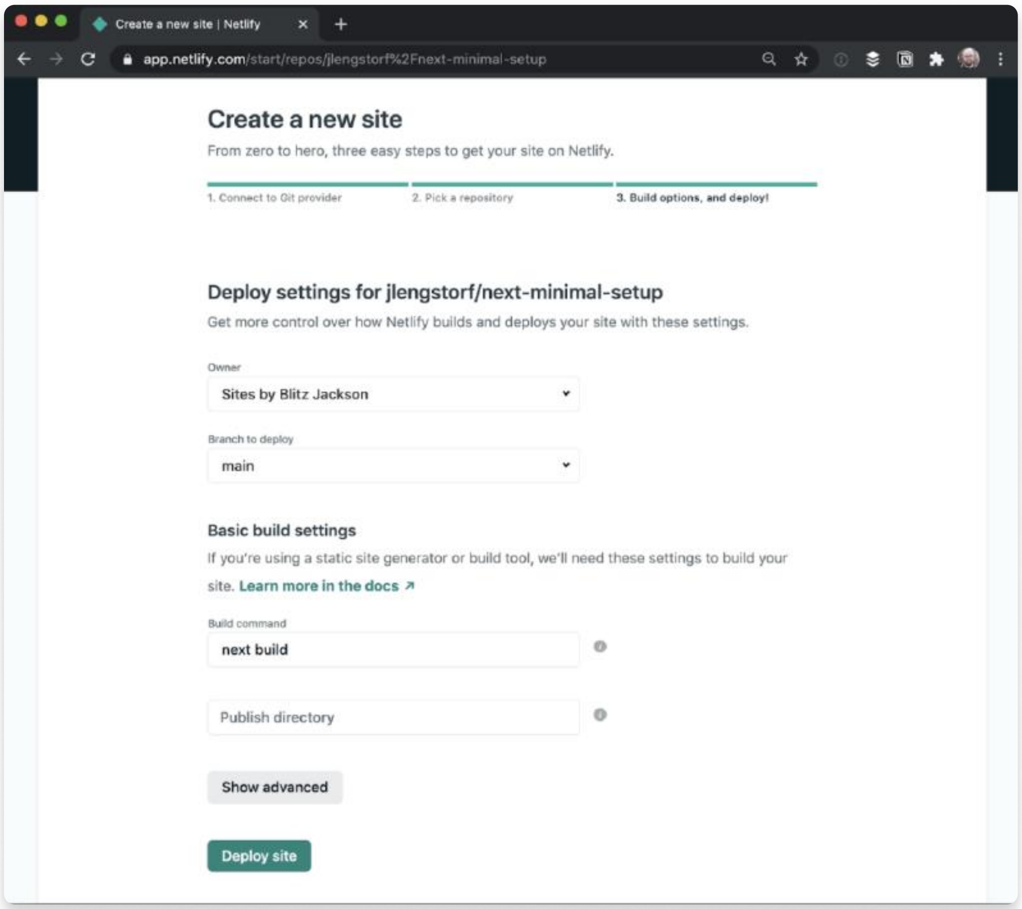
next-minimal

jlengstorf/next-minimal-setup

Can't see your repo here? Configure the Netlify app on GitHub.

The third screen asks for details about where to create the site and how to build it.

- Choose the team and branch to deploy from the dropdowns
- Set the build command to `next build`
- Leave the publish directory blank
- Click “Deploy Site”


Create a new site | Netlify

app.netlify.com/start/repos/jlengstorf%2Fnext-minimal-setup

Create a new site

From zero to hero, three easy steps to get your site on Netlify.

1. Connect to Git provider 2. Pick a repository 3. Build options, and deploy!

Deploy settings for jlengstorf/next-minimal-setup

Get more control over how Netlify builds and deploys your site with these settings.

Owner: Sites by Blitz Jackson

Branch to deploy: main

Basic build settings

If you're using a static site generator or build tool, we'll need these settings to build your site. Learn more in the docs ↗

Build command: next build

Publish directory

That's all!

You've just deployed your Next.js site on Netlify. We can't wait to see what you build.

Get the best of the Jamstack with Netlify's Enterprise plans

In addition to offering a deep integration with Next.js, Netlify has a number of capabilities that make it possible for large enterprise teams to run Next.js applications.

- **Unified development workflow:** Build, test, deploy and manage frontend and serverless code together from a single Git-based workflow. Use powerful Netlify features like Deploy Previews, Split Testing, and instant rollbacks across the entire app.

- **Release management controls:** Handle even the most complex web rollouts with features such as build settings and environmental variables tailored to branches, split testing for managed rollouts, and controls to limit publishing to the main branch.

- **Security and compliance:** Netlify supports businesses with security requirements including SOC 2 Type 2 attestation, SAML single sign-on (SSO) support, role-based access control and two-factor authentication.

- **Self-hosted GitHub and GitLab integration:** Use Netlify with enterprise versions of GitHub and GitLab that are often self-hosted. The integration is enabled and managed from the Netlify UI.

- **24/7/365 production support:** A dedicated support engineer and solutions engineer who knows each team and their tech stack. Available via phone, Slack and email with a guaranteed response SLA.

Contact Sales

netlify