NUXT Docs - Español

Table of contents

Installation

Here, you will find information on setting up and running a Nuxt project in 4 steps.

Online playground

You can play with Nuxt online directly on CodeSandbox or StackBlitz:

:app-button[Play on CodeSandbox] :app-button[Play on StackBlitz]

Prerequisites

- node - *We recommend you have the latest LTS version installed*.
- A text editor, we recommend VS Code with the Vetur extension or WebStorm
- A terminal, we recommend using VS Code's integrated terminal or WebStorm terminal.

Using create-nuxt-app

To get started quickly, you can use create-nuxt-app.

Make sure you have installed yarn, npx (included by default with npm v5.2+) or npm (v6.1+).

```
yarn create nuxt-app <project-name>
```

```
npx create-nuxt-app <project-name>
```

```
npm init nuxt-app <project-name>
```

It will ask you some questions (name, Nuxt options, UI framework, TypeScript, linter, testing framework, etc). To find out more about all the options see the create-nuxt-app documentation.

Once all questions are answered, it will install all the dependencies. The next step is to navigate to the project folder and launch it:

```
cd <project-name>
yarn dev
```

```
cd <project-name>
npm run dev
```

The application is now running on http://localhost:3000. Well done!

Info

Another way to get started with Nuxt is to use CodeSandbox which is a great way for quickly playing around with Nuxt and/or sharing your code with other people.

Manual Installation

Creating a Nuxt project from scratch only requires one file and one directory.

We will use the terminal to create the directories and files, feel free to create them using your editor of choice.

Set up your project

Create an empty directory with the name of your project and navigate into it:

```
mkdir <project-name>
cd <project-name>
```

Replace `<project-name>` *with the name of your project.*

Create the `package.json` file:

```
touch package.json
```

Fill the content of your `package.json` with:

```
{
  "name": "my-app",
  "scripts": {
    "dev": "nuxt",
    "build": "nuxt build",
    "generate": "nuxt generate",
    "start": "nuxt start"
  }
}
```

`scripts` define Nuxt commands that will be launched with the command `npm run <command>` or `yarn <command>`.

What is a package.json file?

The `package.json` is like the ID card of your project. It contains all the project dependencies and much more. If you don't know what the `package.json` file is, we highly recommend you to have a quick read on the npm documentation.

Install Nuxt

Once the `package.json` has been created, add `nuxt` to your project via `npm` or `yarn` like so below:

```
yarn add nuxt
```

```
npm install nuxt
```

This command will add `nuxt` as a dependency to your project and add it to your `package.json`. The `node_modules` directory will also be created which is where all your installed packages and dependencies are stored.

Info

A `yarn.lock` or `package-lock.json` is also created which ensures a consistent install and compatible dependencies of your packages installed in your project.

Create your first page

Nuxt transforms every `*.vue` file inside the `pages` directory as a route for the application.

Create the `pages` directory in your project:

```
mkdir pages
```

Then, create an `index.vue` file in the `pages` directory:

```
touch pages/index.vue
```

It is important that this page is called `index.vue` as this will be the default home page Nuxt shows when the application opens.

Open the `index.vue` file in your editor and add the following content:

```
<template>
  <h1>Hello world!</h1>
</template>
```

Start the project

Run your project by typing one of the following commands below in your terminal:

```
yarn dev
```

```
npm run dev
```

Info

We use the dev command when running our application in development mode.

The application is now running on **http://localhost:3000**.

Open it in your browser by clicking the link in your terminal and you should see the text "Hello World" we copied in the previous step.

Info

When launching Nuxt in development mode, it will listen for file changes in most directories, so there is no need to restart the application when e.g. adding new pages

Warning

When you run the dev command it will create .nuxt folder. This folder should be ignored from version control. You can ignore files by creating a .gitignore file at the root level and adding .nuxt.

Bonus step

Create a page named `fun.vue` in the `pages` directory.

Add a `<template></template>` and include a heading with a funny sentence inside.

Then, go to your browser and see your new page on **localhost:3000/fun**.

Info

Creating a directory named `more-fun` and putting an `index.vue` file inside it will give the same result as creating a `more-fun.vue` file.

Go to TOC

Routing

Most websites have more than just one page. For example a home page, about page, contact page etc. In order to show these pages we need a Router.

Automatic Routes

Most websites will have more than one page (i.e. a home page, about page, contact page etc.). In order to show these pages, we need a Router. That's where `vue-router` comes in. When working with the Vue application, you have to set up a configuration file (i.e. `router.js`) and add all your routes manually to it. Nuxt automatically generates the `vue-router` configuration for you, based on your provided Vue files inside the `pages` directory. That means you never have to write a router config again! Nuxt also gives you automatic code-splitting for all your routes.

In other words, all you have to do to have routing in your application is to create `.vue` files in the `pages` folder.

Next

Learn more about Routing

Navigation

To navigate between pages of your app, you should use the NuxtLink component. This component is included with Nuxt and therefore you don't have to import it as you do with other components. It is similar to the HTML `<a>` tag, except that instead of using a `href="/about"` we use `to="/about"`. If you have used `vue-router` before, you can think of the `<NuxtLink>` as a replacement for `<RouterLink>`

A simple link to the `index.vue` page in your `pages` folder:

```
<template>
  <NuxtLink to="/">Home page</NuxtLink>
</template>
```

For all links to pages within your site, use `<NuxtLink>`. If you have links to other websites you should use the `<a>` tag. See below for an example:

```
<template>
  <main>
    <h1>Home page</h1>
    <NuxtLink to="/about">
      About (internal link that belongs to the Nuxt App)
    </NuxtLink>
    <a href="https://nuxtjs.org">External Link to another page</a>
  </main>
</template>
```

Next

Learn more about the NuxtLink component.

Go to TOC

Directory Structure

The default Nuxt application structure is intended to provide a great starting point for both small and large applications. You are free to organize your application however you like and can create other directories as and when you need them.

Let's create the directories and files that do not exist in our project yet.

```
mkdir components assets static
touch nuxt.config.js
```

These are the main directories and files that we use when building a Nuxt application. You will find an explanation of each of them below.

Info

Creating directories with these names enables features in your Nuxt project.

Directories

The pages directory

The `pages` directory contains your application's views and routes. As you've learned in the last chapter, Nuxt reads all the `.vue` files inside this directory and uses them to create the application router.

Next

Learn more about the pages directory

The components directory

The `components` directory is where you put all your Vue.js components which are then imported into your pages.

With Nuxt you can create your components and auto import them into your .vue files meaning there is no need to manually import them in the script section. Nuxt will scan and auto import these for you once you have components set to true.

Next

Learn more about the components directory

The assets directory

The `assets` directory contains your uncompiled assets such as your styles, images, or fonts.

Next

Learn more about the assets directory

The static directory

The `static` directory is directly mapped to the server root and contains files that have to keep their names (e.g. `robots.txt`) *or* likely won't change (e.g. the favicon)

Next

Learn more about the static directory

The nuxt.config.js file

The `nuxt.config.js` file is the single point of configuration for Nuxt. If you want to add modules or override default settings, this is the place to apply the changes.

Next

Learn more about the nuxt.config.js file

The package.json file

The `package.json` file contains all the dependencies and scripts for your application.

More about the project structures

There are more helpful directories and files, including content, layouts, middleware, modules, plugins and store . As they aren't necessary for small applications, they are not covered here.

Next

To learn about all directories in detail, feel free to read the Directory Structure book.

Go to TOC

Commands and deployment

Nuxt comes with a set of useful commands, both for development and production purpose.

Using in package.json

You should have these commands in your `package.json`:

```
"scripts": {
  "dev": "nuxt",
  "build": "nuxt build",
  "start": "nuxt start",
  "generate": "nuxt generate"
}
```

You can launch your commands via `yarn <command>` or `npm run <command>` (example: `yarn dev` / `npm run dev`).

Development Environment

To launch Nuxt in development mode with hot module replacement on `http://localhost:3000`:

```
yarn dev
```

```
npm run dev
```

List of Commands

You can run different commands depending on the target:

target: `server` (default value)

- **nuxt dev** - Launch the development server.
- **nuxt build** - Build and optimize your application with webpack for production.
- **nuxt start** - Start the production server (after running `nuxt build`). Use it for Node.js hosting like Heroku, Digital Ocean, etc.

target: `static`

- **nuxt dev** - Launch the development server.
- **nuxt generate** - Build the application (if needed), generate every route as a HTML file and statically export to `dist/` directory (used for static hosting).
- **nuxt start** - serve the `dist/` directory like your static hosting would do (Netlify, Vercel, Surge, etc), great for testing before deploying.

Webpack Config Inspection

You can inspect the webpack config used by Nuxt to build the project (similar to vue inspect).

- **nuxt webpack [query...]**

Arguments:

- `--name`: Bundle name to inspect. (client, server, modern)
- `--dev`: Inspect webpack config for dev mode
- `--depth`: Inspection depth. Defaults to 2 to prevent verbose output.
- `--no-colors`: Disable ANSI colors (disabled by default when TTY is not available or when piping to a file)

Examples:

- `nuxt webpack`
- `nuxt webpack devtool`
- `nuxt webpack resolve alias`
- `nuxt webpack module rules`
- `nuxt webpack module rules test=.jsx`
- `nuxt webpack module rules test=.pug oneOf use.0=raw`
- `nuxt webpack plugins constructor.name=WebpackBar options reporter`
- `nuxt webpack module rules loader=vue-`
- `nuxt webpack module rules "loader=.*-loader"`

Production Deployment

Nuxt lets you choose between Server or Static deployments.

Server Deployment

To deploy a SSR application we use `target: 'server'`, where server is the default value.

```
yarn build
```

```
npm run build
```

Nuxt will create a `.nuxt` directory with everything inside ready to be deployed on your server hosting.

Info

We recommend putting `.nuxt` in `.npmignore` or `.gitignore`.

Once your application is built you can use the `start` command to see a production version of your application.

```
yarn start
```

```
npm run start
```

Static Deployment (Pre-rendered)

Nuxt gives you the ability to host your web application on any static hosting.

To deploy a static generated site make sure you have `target: 'static'` in your `nuxt.config.js` (for Nuxt >= 2.13):

```
export default {
  target: 'static'
}
```

```
yarn generate
```

```
npm run generate
```

Nuxt will create a `dist/` directory with everything inside ready to be deployed on a static hosting service.

As of Nuxt v2.13 there is a crawler installed that will now crawl your link tags and generate your routes when using the command `nuxt generate` based on those links.

Warning

Warning: Dynamic routes are ignored by the `generate` command when using Nuxt <= v2.12: API Configuration generate

Info

When generating your web application with `nuxt generate`, the context given to asyncData and fetch will not have `req` and `res`.

Fail on Error

To return a non-zero status code when a page error is encountered and let the CI/CD fail the deployment or build, you can use the `--fail-on-error` argument.

```
yarn generate --fail-on-error
```

```
npm run generate --fail-on-error
```

What's next?

Next

Read our Deployment Guides to find examples for deployments to popular hosts.

Go to TOC

Conclusion

Congratulations you have now created your first Nuxt app and you may now consider yourself a Nuxter. But there is so much more to learn and so much more you can do with Nuxt. Here are a few recommendations.

Next

Check out the Concepts book

Next

Working with asyncData

Next

Choosing between different Rendering modes

alert{type="star"} Did you like Nuxt so far? Don't forget to star our project on GitHub

---

Go to TOC

Upgrading

Upgrading Nuxt is quick, but more involved than updating your package.json

---

If you are upgrading to Nuxt v2.14 and want to use static hosting then you will need to add `target:static` to your nuxt.config.js file in order for the generate command to work properly.

```
export default {
  target: 'static'
}
```

Getting Started

1. Check the release notes for the version you wish to upgrade to see if there are any additional instructions for that particular release.
2. Update the version specified for the `nuxt` package in your `package.json` file.

After this step instructions vary depending upon whether you are using Yarn or npm. *Yarn is the preferred package manager for working with Nuxt as it is the development tool which tests have been written against.*

Yarn

3. remove `yarn.lock` file
4. remove `node_modules` directory
5. Run the `yarn` command
6. After installation has completed and you have run your tests consider upgrading other dependencies as well. The `yarn outdated` command can be used.

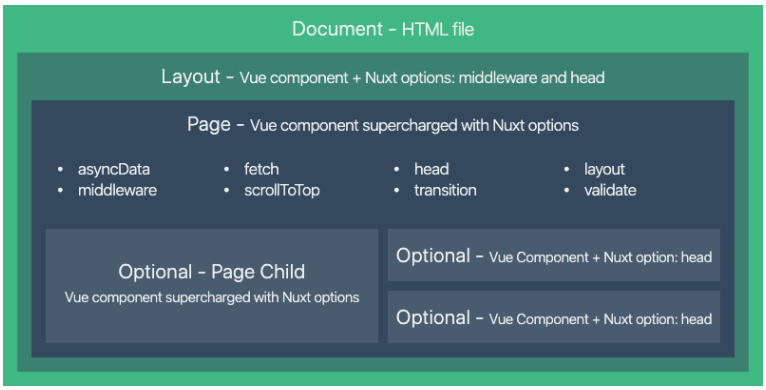
npm

3. remove `package-lock.json` file
4. remove `node_modules` directory
5. Run the `npm install` command
6. After installation has completed and you have run your tests consider upgrading other dependencies as well. The `npm outdated` command can be used.

Go to TOC

Views

The Views section describes all you need to know to configure data and views for a specific route in your Nuxt Application. Views consist of an app template, a layout, and the actual page.



Composition of a View in Nuxt

Composition of a View in Nuxt

Pages

Every Page component is a Vue component but Nuxt adds special attributes and functions to make the development of your application as easy as possible.

```
<template>
  <h1 class="red">Hello World</h1>
</template>

<script>
  export default {
    head() {
      // Set Meta Tags for this Page
    }
    // ...
  }
</script>

<style>
  .red {
    color: red;
  }
</style>
```

Properties of a page component

There are many properties of the page component such as the head property in the example above.

Next

See the Directory Structure book to learn more about all the properties can use on your page

Layouts

Layouts are a great help when you want to change the look and feel of your Nuxt app. For example you want to include a sidebar or have distinct layouts for mobile and desktop.

Default Layout

You can define a default layout by adding a `default.vue` file inside the layouts directory. This will be used for all pages that don't have a layout specified. The only thing you need to include in the layout is the `<Nuxt />` component which renders the page component.

```
<template>
  <Nuxt />
</template>
```

Next

Learn more about the Nuxt component in the components chapter

Custom Layout

You can create custom layouts by adding a `.vue` file to the layouts directory. In order to use the custom layout you need to set the `layout` property in the page component where you want to use that layout. The value will be the name of the custom layout that you have created.

To create a blog layout add a `blog.vue` file to your layouts directory `layouts/blog.vue`:

```
<template>
  <div>
    <div>My blog navigation bar here</div>
    <Nuxt />
  </div>
</template>
```

Warning

Make sure to add the `<Nuxt/>` component when creating a layout to actually include the page component.

We then use the layout property with the value of 'blog' in the page where we want that layout to be used.

```
<template>
  <!-- Your template -->
</template>
<script>
  export default {
    layout: 'blog'
    // page component definitions
  }
</script>
```

Info

If you don't add a layout property to your page, e.g. `layout: 'blog'`, then the `default.vue` layout will be used.

Error Page

The error page is a *page component* which is always displayed when an error occurs (that does not happen while server-side rendering).

Warning

Although this file is placed in the `layouts` folder, it should be treated as a page.

As mentioned above, this layout is special, since you should not include the `<Nuxt/>` component inside its template. You must see this layout as a component displayed when an error occurs (`404` , `500` , etc.). Similar to other page components, you can set a custom layout for the error page as well in the usual way.

You can customize the error page by adding a `layouts/error.vue` file:

```
<template>
  <div>
    <h1 v-if="error.statusCode === 404">Page not found</h1>
    <h1 v-else>An error occurred</h1>
    <NuxtLink to="/">Home page</NuxtLink>
  </div>
</template>

<script>
  export default {
    props: ['error'],
    layout: 'error' // you can set a custom layout for the error page
  }
</script>
```

Document: App.html

The app template is used to create the actual HTML frame of your document for your Nuxt application which injects the content as well as variables for the head and body. This file is created automatically for you and in general rarely needs to be modified. You can customize the HTML app template used by Nuxt to include scripts or conditional CSS classes by creating an `app.html` file in the source directory of your project which by default is the root directory.

The default template used by Nuxt is:

```
<!DOCTYPE html>
<html {{ HTML_ATTRS }}>
  <head {{ HEAD_ATTRS }}>
    {{ HEAD }}
  </head>
  <body {{ BODY_ATTRS }}>
```

```
    {{ APP }}
  </body>
</html>
```

One use case of using a custom app template is to add conditional CSS classes for IE:

```
<!DOCTYPE html>
<!--[if IE 9]><html class="lt-ie9 ie9" {{ HTML_ATTRS }}><![endif]-->
<!--[if (gt IE 9)|!(IE)]><!--><html {{ HTML_ATTRS }}><!--<![endif]-->
  <head {{ HEAD_ATTRS }}>
    {{ HEAD }}
  </head>
  <body {{ BODY_ATTRS }}>
    {{ APP }}
  </body>
</html>
```

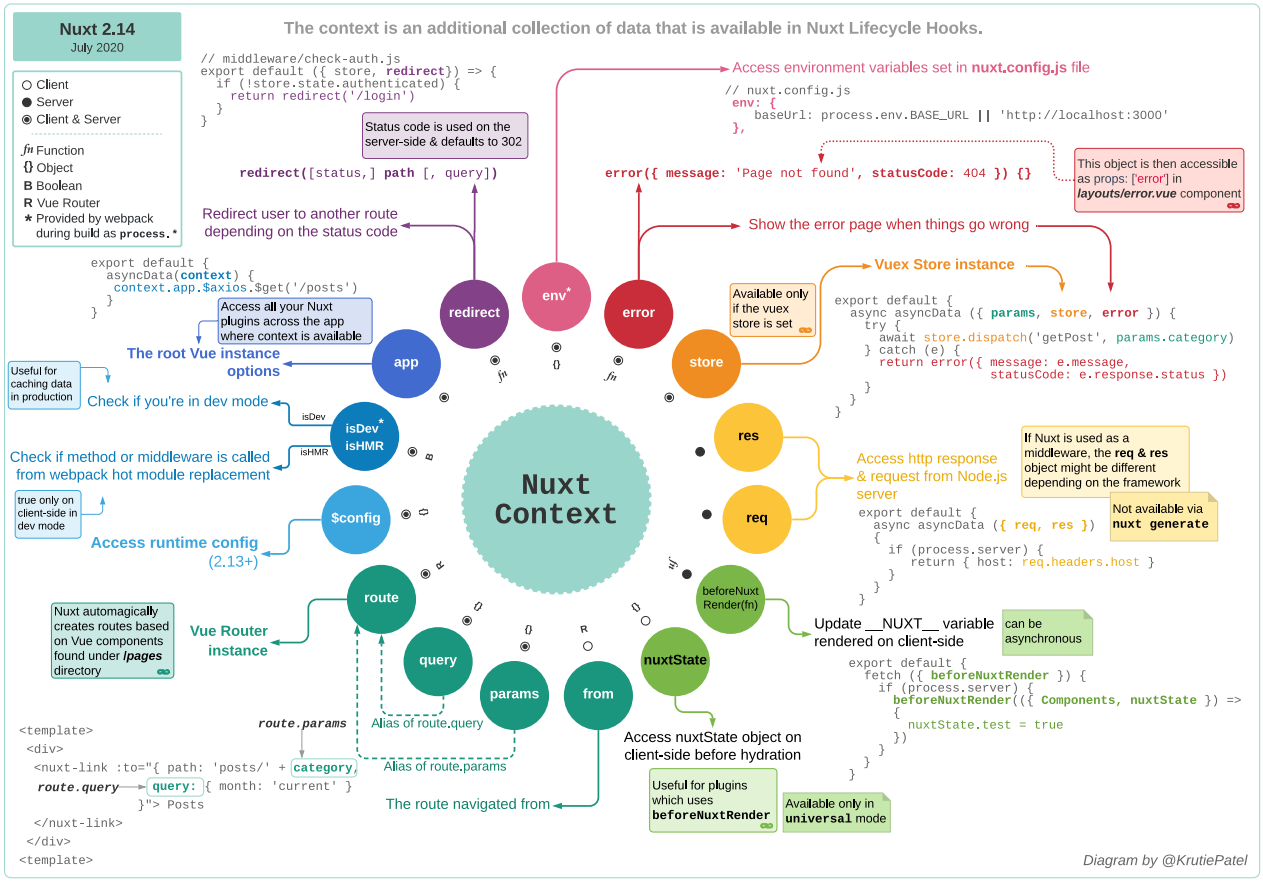
Info

While you can add JavaScript and CSS files in the `app.html`, it is recommended to use the `nuxt.config.js` for these tasks instead!

Go to TOC

Context and helpers

The context provides *additional* and often optional information about the current request to the application.



The `context` object is available in specific Nuxt functions like asyncData, plugins, middleware and nuxtServerInit. It provides *additional* and often optional information about the current request to the application.

First and foremost, the context is used to provide access to other parts of the Nuxt application, e.g. the Vuex store or the underlying `connect` instance. Thus, we have the `req` and `res` objects in the context available on the server side and `store` always available. But with time, the context was extended with many other helpful variables and shortcuts. Now we have access to HMR (Hot Module Reload / Replacement) functionalities in `development` mode, the current `route`, page `params` and `query`, as well as the option to access environment variables through the context. Furthermore, module functions and helpers can be exposed through the context to be available on both - the client and the server side.

All context keys that are present by default

```
function (context) { // Could be asyncData, nuxtServerInit, ...
  // Always available
  const {
    app,
    store,
    route,
    params,
    query,
    env,
    isDev,
    isHMR,
    redirect,
    error,
    $config
  } = context

  // Only available on the Server-side
  if (process.server) {
    const { req, res, beforeNuxtRender } = context
  }

  // Only available on the Client-side
  if (process.client) {
    const { from, nuxtState } = context
  }
}
```

Warning

The *context* we refer to here is not to be confused with the `context` object available in Vuex Actions or the one available in the `build.extend` function in your `nuxt.config.js`. These are not related to each other!

Learn more about the different context keys in our Internals Glossary

Using page parameters for your API query

The context directly exposes possible dynamic parameters of the route via `context.params`. In the following example, we call an API via the `nuxt/http` module using a dynamic page parameter as part of the URL. Modules, like the nuxt/http module, can expose own functions which are then available through the context.app object.

Also, we wrap the API call in a `try/catch` statement to handle potential errors. With the `context.error` function, we can directly show Nuxt's error page and pass in the occurred error.

```
export default {
  async asyncData(context) {
    const id = context.params.id
    try {
      // Using the nuxtjs/http module here exposed via context.app
      const post = await context.app.$http.$get(
        `https://api.nuxtjs.dev/posts/${id}`
      )
      return { post }
    } catch (e) {
      context.error(e) // Show the nuxt error page with the thrown error
    }
  }
}
```

With ES6 you can use this syntax to destructure your context object. You can pass in the objects you want to have access to and then you can use them in the code without using the word context.

```
export default {
  async asyncData({ params, $http, error }) {
    const id = params.id

    try {
      // Using the nuxtjs/http module here exposed via context.app
      const post = await $http.$get(`https://api.nuxtjs.dev/posts/${id}`)
      return { post }
    } catch (e) {
      error(e) // Show the nuxt error page with the thrown error
    }
  }
}
```

Want to use query parameters instead? You then use context.query.id.

Redirecting users & accessing the store

Accessing the Vuex store (when you have it set up through the `store` directory) is also possible through the context. It provides a `store` object which can be treated as `this.$store` in Vue components. In addition, we use the `redirect` method, an exposed helper through the context, to redirect the user in case the `authenticated` state is falsy.

```
export default {
  middleware({ store, redirect }) {
    // retrieving keys via object destructuring
    const isAuthenticated = store.state.authenticated
    if (!isAuthenticated) {
      return redirect('/login')
    }
  }
}
```

Next

Check out the Internals Glossary book for more examples of the redirect method

Helpers

Besides the shortcuts in the context, there are also other tiny helpers present in your Nuxt application.

`$nuxt`: The Nuxt helper

`$nuxt` is a helper designed to improve the user experience and to be an escape hatch in some situations. It is accessible via `this.$nuxt` in Vue components and via `window.$nuxt` otherwise on the client side.

Connection checker

The `$nuxt` helper provides a quick way to find out whether the internet connection of a user is present or not: It exposes the boolean values `isOffline` and `isOnline`. We can use these to show a message as soon as the user is offline (for example).

```
<template>
  <div>
    <div v-if="$nuxt.isOffline">You are offline</div>
    <Nuxt />
  </div>
</template>
```

Accessing the root instance

Besides providing DX/UX (Developer Experience / User Experience) features, the `$nuxt` helper also provides a shortcut to the root instance of your application from every other component. But that's not everything — you can also access the `$nuxt` helper through `window.$nuxt` which can be used as an escape hatch to gain access to module methods like `$axios` from outside your Vue components. This should be used wisely and **only as last resort**.

Refreshing page data

When you want to refresh the current page for the user, you don't want to fully reload the page because you might hit the server again and at least re-initialize the whole Nuxt application. Instead, you often only want to refresh the data, provided by `asyncData` or `fetch`.

You can do so, by using `this.$nuxt.refresh()`!

```
<template>
  <div>
    <div>{{ content }}</div>
    <button @click="refresh">Refresh</button>
  </div>
</template>

<script>
  export default {
    asyncData() {
      return { content: 'Created at: ' + new Date() }
    },
    methods: {
      refresh() {
        this.$nuxt.refresh()
      }
    }
  }
</script>
```

Controlling the loading bar

With `$nuxt`, you can also control Nuxt's loading bar programmatically via `this.$nuxt.$loading`.

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Read more in the corresponding loading feature chapter

onNuxtReady helper

If you want to run some scripts *after* your Nuxt application has been loaded and is ready, you can use the `window.onNuxtReady` function. This can be useful when you want to execute a function on the client-side without increasing the time to interactive of your site.

```
window.onNuxtReady(() => {
  console.log('Nuxt is ready and mounted')
})
```

Process helpers

Nuxt injects three boolean values (`client` , `server` , and `static`) into the global `process` object which will help you to determine whether your app was rendered on the server or fully on the client, as well as checking for static site generation. These helpers are available across your application and are commonly used in `asyncData` userland code.

```
<template>
  <h1>I am rendered on the {{ renderedOn }} side</h1>
</template>

<script>
  export default {
    asyncData() {
      return { renderedOn: process.client ? 'client' : 'server' }
    }
  }
</script>
```

In the example, `renderedOn` will evaluate to `'server'` when using server-side rendering and a user accesses the page directly. When the user would navigate to the page from another part of the application, e.g. by click on a `<NuxtLink>` , it will evaluate to client.

Go to TOC

Server side rendering

Server-side rendering (SSR), is the ability of an application to contribute by displaying the web-page on the server instead of rendering it in the browser. Server-side sends a fully rendered page to the client; the client's JavaScript bundle takes over which then allows the Vue.js app to hydrate.

Node.js server required

A JavaScript environment is required to render your web page.

A Node.js server needs to be configured to execute your Vue.js application.

Extend and control the server

You can extend the server with serverMiddleware and control routes with middleware.

```
export default function (req, res, next) {
  console.log(req.url)
  next()
}
```

```
export default {
  serverMiddleware: ['~/server-middleware/logger']

}
```

Server vs Browser environments

Because you are in a Node.js environment you have access to Node.js objects such as `req` and `res`. You do not have access to the `window` or `document` objects as they belong to the browser environment. You can however use `window` or `document` by using the `beforeMount` or `mounted` hooks.

```
beforeMount () {
  window.alert('hello');
}
mounted () {
  window.alert('hello');
}
```

Server-side rendering steps with Nuxt

Step 1: Browser to Server

When a browser sends the initial request, it will hit the Node.js internal server. Nuxt will generate the HTML and send it back to the browser with results from executed functions, e.g. `asyncData`, `nuxtServerInit` or `fetch`. Hooks functions are executed as well.

Step 2: Server to Browser

The browser receives the rendered page from the server with the generated HTML. The content is displayed and the Vue.js hydration kicks in, making it reactive. After this process, the page is interactive.

Step 3: Browser to Browser

Navigating between pages with `<NuxtLink>` is done on the client side so you don't hit the server again unless you hard refresh the browser.

Caveats

window or document undefined

This is due to the server-side rendering. If you need to specify that you want to import a resource only on the client-side, you need to use the `process.client` variable.

For example, in your `.vue` file:

```
if (process.client) {
  require('external_library')
}
```

iOS and phone numbers

Some mobile Safari versions will automatically transform phone numbers into links. This will trigger a `NodeMismatch` warning as the SSR content doesn't match the website content anymore. This can make your app unusable on these Safari versions.

If you include telephone numbers in your Nuxt page, you have two options.

Use a meta tag to stop the transformation

```
<meta name="format-detection" content="telephone=no" />
```

Wrap your phone numbers in links

```
<!-- Example phone number: +7 (982) 536-50-77 -->

<template>
  <a href="tel: +7 (982) 536-50-77">+7 (982) 536-50-77</a>
</template>
```

Go to TOC

Static Site Generation

With static site generation you can render your application during the build phase and deploy it to any static hosting services such as Netlify, GitHub pages, Vercel etc. This means that no server is needed in order to deploy your application.

Generating your site

When deploying your site in with `target:static` all your `.vue` pages will be generated into HTML and JavaScript files. All calls to APIs will be made and cached in a folder called static inside your generated content so that no calls to your API need to be made on client side navigation.

Step 1: Browser to CDN

When a browser sends the initial request, it will hit the CDN.

Step 2: CDN to Browser

The CDN will send the already generated HTML, JavaScript and static assets back to the browser. The content is displayed and the Vue.js hydration kicks in, making it reactive. After this process, the page is interactive.

Step 3: Browser to Browser

Navigating between pages with `<NuxtLink>` is done on the client side so you don't hit the CDN again and all API calls will be loaded from the already cached static folder even if you hard refresh the browser.

SPA Fallback

Pages that have been excluded from generation, by using the `generate.exclude` property will fallback to being a single page application. These pages will therefore not exist in the CDN and will be rendered on client side in the browser once the user navigates to that page.

Next

To learn more about the generate property

Updating your content

In order to get new content to your site from your API you will need to regenerate your site again. With most static sites hosting providers you can do this by pushing your changes to your main branch via the git command or via a pull request.

Preview Mode

The Preview mode will call your API or your CMS so you can see the changes live before deploying. See the preview mode on how to enable this feature.

Go to TOC

Nuxt Lifecycle

No matter which tool you use, you will always feel more confident when you understand how the tool works under the hood. The same applies to Nuxt.

NUXT.JS LIFECYCLE HOOKS

Nuxt >= 2.12 - April 2020

@KrutiePatel

*Available **only** on page components*
*Available in **any** Vue components*
Nuxt Context Available

sets default data on server-side

...Only hook where Nuxt context is received as 2nd argument - the 1st argument is Vuex-context !

nuxt ServerInit

First **lifecycle hook** called from server-side, ***if*** Vuex store is set

...use`em to set or check conditions or even redirect page

Called before rendering page components

Route Middleware

Three types of route middleware, **called in order from 1 to 3**:
1. Global - affects all routes, defined in **nuxt.config.js**
2. Layout - affects group of routes, defined in **layouts**
3. Page - affects single route, defined on **page components**

useful to validate dynamic route parameter

must return **true** or **false**

validate()

Called before rendering page components

Called every time before loading page components

...merges with component data

...fetches data before rendering page

asyncData()

fetch(context)

...kinda same as *asyncData*, but doesn't set component data

I'd mostly use with **Vuex store** to fetch data

...can still be used as of Nuxt2.12 -for backward compatibility reason- as long as **context** is passed as an argument

Beginning of Vue Lifecycle Hooks

beforeCreate hook is called when the Vue instance is initialised

Called when Vue instance is created

created()

fetch pending ⟶ `$fetchState.pending = true`

New fetch feature introduced from Nuxt 2.12

Called after Vue instance is created

...mutates local data

fetch()

`this` context available here!

Works as a method too...

Handles error at component level using **$fetchState.error**

beforeMount hook is called before initial DOM render

```
<button @click="$fetch">
<template v-else-if="$fetchState.error">
```

Called when DOM is rendered and reactive

mounted()

fetch complete ⟶ `$fetchState.pending = false`

...rest of the Vue Lifecycle Hooks

The goal of this chapter is to give you a high-level overview of the different parts of the framework, their order of execution and how they work together.

The Nuxt lifecycle describes what happens after the build phase, where your application is bundled, chunked and minified. What happens after this phase depends on whether you have server-side rendering enabled or not. And if you do, it further depends on the type of server-side rendering you have chosen:

Dynamic SSR (`nuxt start`)

or Static Site Generation (`nuxt generate`).

Lifecycle

Server

For SSR, these steps will be executed for every initial request to your app

- The server starts (`nuxt start`)

When using static site generation, the server steps are only executed on build time, but once for every page that will be generated

- The generation process starts (`nuxt generate`)
- Nuxt hooks
- serverMiddleware
- Server-side Nuxt plugins
  - in order as defined in nuxt.config.js
- nuxtServerInit
  - Vuex action that is called only on server-side to pre-populate the store
  - First argument is the **Vuex context**, second argument is the **Nuxt context**
    - Dispatch other actions from here → only "entry point" for subsequent store actions on server-side
  - can only be defined in `store/index.js`
- Middleware
  - Global middleware
  - Layout middleware
  - Route middleware
- asyncData
- beforeCreate (Vue lifecycle method)
- created (Vue lifecycle method)
- The new fetch (top to bottom, siblings = parallel)

- Serialization of state (`render:routeContext` Nuxt hook)
- the HTML rendering happens (`render:route` Nuxt hook)
- `render:routeDone` hook when HTML has been sent to the browser
- `generate:before` Nuxt hook
- HTML files are generated
  - **Full static generation**
    - e.g. static payloads are extracted
  - `generate:page` (HTML editable)
  - `generate:routeCreated` (Route generated)
- `generate:done` when all HTML files have been generated

Client

This part of the lifecycle is fully executed in the browser, no matter which Nuxt mode you've chosen.

- Receives the HTML
- Loading assets (e.g. JavaScript)
- client-side Nuxt plugin
  - in order as defined in nuxt.config.js
- Vue Hydration
- Middleware
  - Global middleware
  - Layout middleware
  - Route middleware
- asyncData (blocking)
- beforeCreate (Vue lifecycle method)
- created (Vue lifecycle method)
- The new fetch (top to bottom, siblings = parallel) (non-blocking)
- beforeMount (Vue lifecycle method)
- mounted (Vue lifecycle method)

Navigate using the NuxtLink component

Same as for the *client* part, everything is happening in the browser but only when navigating via `<Nuxt-Link>`. Furthermore, no page content is displayed until all *blocking* tasks are fulfilled.

Info

Check out the component chapter to see more info on `<NuxtLink>`

- middleware (blocking)
  - Global middleware
  - Layout middleware

    - Route middleware
- asyncData (blocking) [or full static payload loading]
- beforeCreate & created (Vue lifecycle methods)
- fetch (non-blocking)
- beforeMount & mounted

What's next

Next

Check out the Features book

Go to TOC

Rendering

Server Side Rendered Sites and Static Sites

Server-side rendered sites are rendered on the server each time the user requests a page, therefore a server is needed to be able to serve the page on each request.

Static sites are very similar to server-side rendered applications with the main difference being that static sites are rendered at build time, therefore no server is needed. Navigating from one page to another is then on the client-side.

See deployment targets for more info on static and server hosting.

```
export default {
  ssr: true // default value
}
```

Info

You do not need to add `ssr: true` to your nuxt config in order to enable server-side-rendering as it is enabled by default.

Client Side Rendering Only

With client side rendering only there is no server-side rendering. Client side rendering means rendering the content in the browser using JavaScript. Instead of getting all of the content from the HTML we just get a basic HTML document with a JavaScript file that will then render the rest of the site using the browser. For client side rendering set ssr to `false`.

```
export default {
  ssr: false
}
```

Next

The ssr property

Go to TOC

Transitions

Nuxt uses the transition component to let you create amazing transitions/animations between your routes.

To define a custom transition for a specific route add the `transition` key to the page component.

```
export default {
  // Can be a String
  transition: ''
  // Or an Object
  transition: {}
  // or a Function
  transition (to, from) {}
}
```

String

If the `transition` key is set as a string, it will be used as the `transition.name`.

```
export default {
  transition: 'home'
}
```

Nuxt will use these settings to set the component as follows:

```
<transition name="home"></transition>
```

Warning

This is automatically done for you and you do not need to add the `<transition>` component to your pages or layouts.

Now all you have to do is create the new class for your transitions.

```
<style>
  .home-enter-active, .home-leave-active { transition: opacity .5s; }
  .home-enter, .home-leave-active { opacity: 0; }
</style>
```

Object

If the `transition` key is set as an object:

```
export default {
  transition: {
    name: 'home',
    mode: 'out-in'
  }
}
```

Nuxt will use these settings to set the component as follows:

```
<transition name="home" mode="out-in"></transition>
```

The `transition` object can have many properties such as name, mode, css, duration and many more. Please see the vue docs for more info.

You can also define methods in the page `transition` property, for more information on the JavaScript hooks see the vue docs.

```
export default {
  transition: {
    afterLeave(el) {
      console.log('afterLeave', el)
    }
  }
}
```

Transition Mode

Warning

The default transition mode for pages differs from the default mode in Vue.js. The `transition` mode is by default set to `out-in`. If you want to run leaving and entering transitions simultaneously, you have to set the mode to the empty string `mode: ''`.

```
export default {
  transition: {
    name: 'home',
    mode: ''
  }
}
```

Function

If the `transition` key is set as a function:

```
export default {
  transition(to, from) {
    if (!from) {
      return 'slide-left'
    }
    return +to.query.page < +from.query.page ? 'slide-right' : 'slide-left'
  }
}
```

Transitions applied on navigation:

`/` to `/posts` => `slide-left`, `/posts` to `/posts?page=3` => `slide-left`, `/posts?page=3` to `/posts?page=2` => `slide-right`.

Global Settings

The Nuxt default transition name is `"page"`. To add a fade transition to every page of your application, all you need is a CSS file that is shared across all your routes.

Our global css in `assets/main.css`:

```
.page-enter-active,
.page-leave-active {
  transition: opacity 0.5s;
}
.page-enter,
.page-leave-to {
  opacity: 0;
}
```

Then we add its path to the `css` array in our `nuxt.config.js` file:

```
export default {
  css: ['~/assets/main.css']
}
```

Configuration Settings

The layoutTransition Property

The layout transition is used to set the default properties of the layout transitions.

The default settings for layout transitions are:

```
{
  name: 'layout',
  mode: 'out-in'
}
```

```
.layout-enter-active,
.layout-leave-active {
  transition: opacity 0.5s;
}
.layout-enter,
.layout-leave-active {
  opacity: 0;
}
```

If you want to change the default settings for your layout transitions you can do so in the nuxt.config.js file.

```
export default {
  layoutTransition: 'my-layouts'
  // or
  layoutTransition: {
    name: 'my-layouts',
    mode: 'out-in'
  }
}
```

```
.my-layouts-enter-active,
.my-layouts-leave-active {
  transition: opacity 0.5s;
}
.my-layouts-enter,
.my-layouts-leave-active {
  opacity: 0;
}
```

The pageTransition Property

The default settings for page transitions are:

```
{
  name: 'page',
  mode: 'out-in'
}
```

Should you wish to modify the default settings you can do so in the nuxt.config.js

```
export default {
  pageTransition: 'my-page'
  // or
  pageTransition: {
    name: 'my-page',
    mode: 'out-in',
    beforeEnter (el) {
      console.log('Before enter...');
    }
  }
}
```

If you do modify the page Transition name you will also have to rename the css class.

```
.my-page-enter-active,
.my-page-leave-active {
  transition: opacity 0.5s;
}
.my-page-enter,
.my-page-leave-to {
  opacity: 0;
}
```

Go to TOC

Preview mode

Live Preview for target static with the preview mode

With Nuxt and full static you can now use live preview out of the box which will call your API or your CMS so you can see the changes live before deploying.

Warning

Only available when using `target:static`

The preview mode will automatically refresh the page data as it uses `$nuxt.refresh` under the hood and therefore calls nuxtServerInit, asyncData and fetch on the client side.

In order to activate live preview you will need to add the following plugin:

```
export default function ({ query, enablePreview }) {
  if (query.preview) {
    enablePreview()
  }
}
```

Warning

`enablePreview` is only available in the context object of plugins. Previews are handled client-side and thus the plugin should be run on the client: preview.client.js

```
export default {
  plugins: ['~/plugins/preview.client.js']
}
```

Once you have added the plugin you can now generate your site and serve it.

```
yarn generate
yarn start
```

```
npx nuxt generate
npx nuxt start
```

Then you can see your preview page by adding the query param to the end of the page you want to see once:

```
?preview=true
```

Warning

enablePreview should be tested locally with yarn start and not yarn dev

Previewing pages that are not yet generated

For pages that are not yet generated, SPA fallback will still call the API before showing the 404 page as these pages exist on the API but are not generated yet.

If you have set a validate hook, you will probably need to modify it so that it doesn't redirect to the 404 page in preview mode.

```
validate({ params, query }) {
  if (query.preview) {
    return true
}
```

Passing data to enablePreview

You can pass data to the `enablePreview` function. That data will then be available on the `$preview` context helper and on `this.$preview`.

What's next

Next

Check out the Directory Structure book

Go to TOC

Deployment Targets

Static Hosting

Nuxt also works as a static site generator. Statically render your Nuxt application and get all of the benefits of a universal app without a server. The `nuxt generate` command will generate a static version of your website. It will generate HTML for every one of your routes and put it inside of its own file in the `dist/` directory. This improves performance as well as SEO and better offline support.

Info

Dynamic routes are also generated thanks to the Nuxt Crawler

For static sites the target of `static` needs to be added to your `nuxt.config` file.

```
export default {
  target: 'static' // default is 'server'
}
```

Running nuxt dev with the static target will improve the developer experience:

- Remove `req` & `res` from `context`
- Fallback to client-side rendering on 404, errors and redirects see SPA fallback
- `$route.query` will always be equal to `{}` on server-side rendering
- `process.static` is true

Info

We are also exposing `process.target` for module authors to add logic depending on the user target.

Server Hosting

Server hosting means running Nuxt on a Node.js server. When the user opens your page, their browser will request that page from the server. Nuxt will handle the request, render the page and send back the resulting page with all its content.

You might need server hosting if you want to render HTML on each request rather than in advance at generate-time, or if you need serverMiddleware.

Info

You can still run Nuxt with server hosting with `ssr: false` but Nuxt will not fully render the HTML for each page - leaving that task to the browser. You might choose this option if you need serverMiddleware but do not want fully server-side rendered HTML.

For server hosting, `target: 'server'` is used, which is the default value. You will use the `build` command to build your application.

```
export default {
  target: 'server'
}
```

Go to TOC

File system routing

Nuxt automatically generates the vue-router configuration based on your file tree of Vue files inside the pages directory. When you create a .vue file in your pages directory you will have basic routing working with no extra configuration needed.

Sometimes you might need to create dynamic routes or nested routes or you might need to further configure the router property. This chapter will go through everything you need to know in order to get the best out of your router.

Info

Nuxt gives you automatic code splitting for your routes, no configuration is needed

Info

Use the NuxtLink component to navigate between pages

```
<template>
  <NuxtLink to="/">Home page</NuxtLink>
</template>
```

Basic Routes

This file tree:

```
pages/
--| user/
-----| index.vue
-----| one.vue
--| index.vue
```

will automatically generate:

```
router: {
  routes: [
    {
      name: 'index',
      path: '/',
      component: 'pages/index.vue'
    },
    {
      name: 'user',
      path: '/user',
      component: 'pages/user/index.vue'
    },
    {
      name: 'user-one',
      path: '/user/one',
      component: 'pages/user/one.vue'
```

```
    }
  ]
}
```

Dynamic Routes

Sometimes it is not possible to know the name of the route such as when we make a call to an API to get a list of users or blog posts. We call these dynamic routes. To create a dynamic route you need to add an underscore (`_`) before the `.vue` file name or before the name of the directory. You can name the file or directory anything you want but you must prefix it with an underscore.

This file tree:

```
pages/
--| _slug/
-----| comments.vue
-----| index.vue
--| users/
-----| _id.vue
--| index.vue
```

will automatically generate:

```
router: {
  routes: [
    {
      name: 'index',
      path: '/',
      component: 'pages/index.vue'
    },
    {
      name: 'users-id',
      path: '/users/:id?',
      component: 'pages/users/_id.vue'
    },
    {
      name: 'slug',
      path: '/:slug',
      component: 'pages/_slug/index.vue'
    },
    {
      name: 'slug-comments',
      path: '/:slug/comments',
      component: 'pages/_slug/comments.vue'
    }
  ]
}
```

Info

As you can see the route named `users-id` has the path `:id?` which makes it optional, if you want to make it required, create an `index.vue` file in the `users/_id` directory instead.

Info

As of Nuxt >= v2.13 there is a crawler installed that will now crawl your link tags and generate your dynamic routes based on those links. However if you have pages that are not linked to such as a secret page, then you will need to manually generate those dynamic routes.

Next

Generate dynamic routes for static sites

Locally Accessing Route Params

You can access the current route parameters within your local page or component by referencing `this.$route.params.{parameterName}`. For example, if you had a dynamic users page (`users/_id.vue`) and wanted to access the `id` parameter to load the user or process information, you could access the variable like this: `this.$route.params.id`.

Nested Routes

Nuxt lets you create nested routes by using the children routes of vue-router. To define the parent component of a nested route, you need to create a Vue file with the same name as the directory which contains your children views.

Warning

Don't forget to include the NuxtChild component inside the parent component (`.vue` file).

This file tree:

```
pages/
--| users/
-----| _id.vue
-----| index.vue
--| users.vue
```

will automatically generate:

```
router: {
  routes: [
    {
      path: '/users',
      component: 'pages/users.vue',
      children: [
        {
          path: '',
          component: 'pages/users/index.vue',
          name: 'users'
        },
        {
          path: ':id',
          component: 'pages/users/_id.vue',
          name: 'users-id'
        }
      ]
```

```
    }
  ]
}
```

Dynamic Nested Routes

This is not a common scenario, but it is possible with Nuxt to have dynamic children inside dynamic parents.

This file tree:

```
pages/
--| _category/
-----| _subCategory/
| _id.vue
| index.vue
-----| _subCategory.vue
-----| index.vue
--| _category.vue
--| index.vue
```

will automatically generate:

```
router: {
  routes: [
    {
      path: '/',
      component: 'pages/index.vue',
      name: 'index'
    },
    {
      path: '/:category',
      component: 'pages/_category.vue',
      children: [
        {
          path: '',
          component: 'pages/_category/index.vue',
          name: 'category'
        },
        {
          path: ':subCategory',
          component: 'pages/_category/_subCategory.vue',
          children: [
            {
              path: '',
              component: 'pages/_category/_subCategory/index.vue',
              name: 'category-subCategory'
            },
            {
              path: ':id',
              component: 'pages/_category/_subCategory/_id.vue',
              name: 'category-subCategory-id'
            }
          ]
        }
      ]
    }
  ]
}
```

Unknown Dynamic Nested Routes

If you do not know the depth of your URL structure, you can use `_.vue` to dynamically match nested paths. This will handle requests that do not match a *more specific* route.

This file tree:

```
pages/
--| people/
-----| _id.vue
-----| index.vue
--| _.vue
--| index.vue
```

Will handle requests like this:

```
/ -> index.vue
/people -> people/index.vue
/people/123 -> people/_id.vue
/about -> _.vue
/about/careers -> _.vue
/about/careers/chicago -> _.vue
```

Info

Handling 404 pages is now up to the logic of the `_.vue` page.

Extending the router

There are multiple ways to extend the routing with Nuxt:

- router-extras-module to customize the route parameters in the page
- component@nuxtjs/router to overwrite the Nuxt router and write your own `router.js` file
- Use the router.extendRoutes property in your `nuxt.config.js`

The router Property

The router property lets you customize the Nuxt router (vue-router).

```
export default {
  router: {
    // customize the Nuxt router
  }
}
```

Base:

The base URL of the app. For example, if the entire single page application is served under `/app/`, then base should use the value `'/app/'`.

Next

Router Base Property

extendRoutes

You may want to extend the routes created by Nuxt. You can do so via the `extendRoutes` option.

Example of adding a custom route:

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        name: 'custom',
        path: '*',
        component: resolve(__dirname, 'pages/404.vue')
      })
    }
  }
}
```

If you want to sort your routes, you can use the `sortRoutes(routes)` function from `@nuxt/utils`:

```
import { sortRoutes } from '@nuxt/utils'
export default {
  router: {
    extendRoutes(routes, resolve) {
      // Add some routes here ...

      // and then sort them
      sortRoutes(routes)
    }
  }
}
```

Warning

The schema of the route should respect the vue-router schema.

Warning

When adding routes that use Named Views, don't forget to add the corresponding `chunkNames` of named `components`.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        path: '/users/:id',
        components: {
          default: resolve(__dirname, 'pages/users'), // or
routes[index].component
          modal: resolve(__dirname, 'components/modal.vue')
        },
        chunkNames: {
          modal: 'components/modal'
```

```
        }
      })
    }
  }
}
```

Next

extendRoutes Property

fallback

Controls whether the router should fallback to hash mode when the browser does not support history.push-State but mode is set to history.

Next

fallback Property

mode

Configure the router mode, it is not recommended to change it due to server-side rendering.

Next

mode Property

parseQuery / stringifyQuery

Provide custom query string parse / stringify functions.

Next

parseQuery / stringifyQuery Property

routeNameSplitter

You may want to change the separator between route names that Nuxt uses. You can do so via the `route-NameSplitter` option in your configuration file. Imagine we have the page file `pages/posts/_id.vue`. Nuxt will generate the route name programmatically, in this case `posts-id`. Changing the `routeNameS-plitter` config to `/` the name will therefore change to `posts/id`.

```
export default {
  router: {
    routeNameSplitter: '/'
  }
}
```

scrollBehavior

The `scrollBehavior` option lets you define a custom behavior for the scroll position between the routes. This method is called every time a page is rendered.

Next

To learn more about it, see vue-router scrollBehavior documentation.

Available since:v2.9.0:

In Nuxt you can use a file to overwrite the router scrollBehavior. This file should be placed in a folder called app.

`~/app/router.scrollBehavior.js`.

Example of forcing the scroll position to the top for every route:

```
export default function (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

Next

Nuxt default `router.scrollBehavior.js` file.

Next

scrollBehavior Property

trailingSlash

Available since: v2.10

If this option is set to true, trailing slashes will be appended to every route. If set to false, they'll be removed.

```
export default {
  router: {
    trailingSlash: true
  }
}
```

Warning

This option should not be set without preparation and has to be tested thoroughly. When setting `router.trailingSlash` to something else other than `undefined` (which is the default value), the opposite route will stop working. Thus 301 redirects should be in place and your *internal linking* has to be adapted correctly. If you set `trailingSlash` to `true`, then only `example.com/abc/` will work but not `example.com/abc`. On false, it's vice-versa.

Next

trailingSlash Property

Go to TOC

Data Fetching

In Nuxt we have 2 ways of getting data from an API. We can use the fetch method or the asyncData method.

Nuxt supports traditional Vue patterns for loading data in your client-side app, such as fetching data in a component's `mounted()` hook. Universal apps, however, need to use Nuxt-specific hooks to be able to render data during server-side rendering. This allows your page to render with all of its required data present.

Nuxt has two hooks for asynchronous data loading:

- **`asyncData`**. This hook can only be placed on *page* components. Unlike `fetch`, this hook does not display a loading placeholder during client-side rendering: instead, this hook blocks route navigation until it is resolved, displaying a page error if it fails.
- **`fetch`** (Nuxt 2.12+). This hook can be placed on any component, and provides shortcuts for rendering loading states (during client-side rendering) and errors.

These hooks can be used with *any data fetching library* you choose. We recommend using @nuxt/http or @nuxt/axios for making requests to HTTP APIs. More information about these libraries, such as guides for configuring authentication headers, can be found in their respective documentation.

Info

If you define `fetch` or `asyncData` inside a mixin and also have it defined in a component/page, the mixin function will be overwritten instead of called.

The fetch hook

Info

Before Nuxt 2.12, there was a different `fetch` hook that only worked for *page* components and didn't have access to the component instance.

If your `fetch()` accepts a `context` argument, it will be treated like a legacy fetch hook. This functionality is deprecated, and should be replaced with either `asyncData` or an anonymous middleware.

`fetch` is a hook called during server-side rendering after the component instance is created, and on the client when navigating. The fetch hook should return a promise (whether explicitly, or implicitly using `async/await`) that will be resolved:

- On the server, before the initial page is rendered
- On the client, some time after the component is mounted

Info

For static hosting, the fetch hook is only called during page generation, and the result is then cached for use on the client. To avoid cache conflicts, it may be necessary to specify a name for your component, or alternatively provide a unique fetchKey implementation.

Usage

Fetching data

Within the fetch hook, you will have access to the component instance via `this`.

Info

Make sure any properties you want to modify have already been declared in `data()`. Then the data that comes from the fetch can be assigned to these properties.

Changing fetch behavior

`fetchOnServer`: `Boolean` or `Function` (default: `true`), call `fetch()` when server-rendering the page

`fetchKey`: `String` or `Function` (defaults to the component scope ID or component name), a key (or a function that produces a unique key) that identifies the result of this component's fetch (available on Nuxt 2.15+). When hydrating a server-rendered page, this key is used to map the result of the server-side `fetch()` to the client-side component data. More information available in original PR.

`fetchDelay`: `Integer` (default: `200`), set the minimum executing time in milliseconds (to avoid quick flashes)

When `fetchOnServer` is falsy (`false` or returns `false`), `fetch` will be called only on client-side and `$fetchState.pending` will return `true` when server-rendering the component.

```
export default {
  data: () => ({
    posts: []
  }),
  async fetch() {
    this.posts = await this.$http.$get('https://api.nuxtjs.dev/posts')
  },
  fetchOnServer: false,
  // multiple components can return the same `fetchKey` and Nuxt will track them
both separately
  fetchKey: 'site-sidebar',
  // alternatively, for more control, a function can be passed with access to the
component instance
  // It will be called in `created` and must not depend on fetched data
  fetchKey(getCounter) {
    // getCounter is a method that can be called to get the next number in a
sequence
    // as part of generating a unique fetchKey.
    return this.someOtherData + getCounter('sidebar')
  }
}
```

Accessing the fetch state

The `fetch` hook exposes `this.$fetchState` at the component level with the following properties:

- `pending` is a `Boolean` that allows you to display a placeholder when `fetch` is being called *on client-side*.
- `error` is either `null` or an `Error` thrown by the fetch hook
- `timestamp` is a timestamp of the last fetch, useful for caching with `keep-alive`

In addition to fetch being called by Nuxt, you can manually call fetch in your component (to e.g. reload its async data) by calling `this.$fetch()`.

```
<template>
  <div>
    <p v-if="$fetchState.pending">Fetching mountains...</p>
    <p v-else-if="$fetchState.error">An error occurred :(</p>
    <div v-else>
      <h1>Nuxt Mountains</h1>
      <ul>
        <li v-for="mountain of mountains">{{ mountain.title }}</li>
      </ul>
      <button @click="$fetch">Refresh</button>
    </div>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        mountains: []
      }
    },
    async fetch() {
      this.mountains = await fetch(
        'https://api.nuxtjs.dev/mountains'
      ).then(res => res.json())
    }
  }
</script>
```

Info

You can access the Nuxt context within the fetch hook using `this.$nuxt.context`.

Listening to query string changes

The `fetch` hook is not called on query string changes by default. To watch for query changes you can add a watcher on `$route.query` and call `$fetch`:

```
export default {
  watch: {
    '$route.query': '$fetch'
  },
  async fetch() {
```

```
    // Called also on query changes
  }
}
```

Caching

You can use `keep-alive` directive in `<nuxt/>` and `<nuxt-child/>` component to save `fetch` calls on pages you already visited:

```
<template>
  <nuxt keep-alive />
</template>
```

You can also specify the props passed to `<keep-alive>` by passing a prop `keep-alive-props` to the `<nuxt>` component.

```
<nuxt keep-alive :keep-alive-props="{ max: 10 }" />
```

Keeps only 10 page components in memory.

Error handling

Warning

If there is an error when fetching data, the normal Nuxt error page won't be loaded - and you should not use the Nuxt `redirect` or `error` methods within `fetch()`. Instead, you will need to handle it within your component using `$fetchState.error`.

We can check `$fetchState.error` and show an error message if there is an error fetching the data.

```
<template>
  <div>
    <p v-if="$fetchState.pending">Loading....</p>
    <p v-else-if="$fetchState.error">Error while fetching mountains</p>
    <ul v-else>
      <li v-for="(mountain, index) in mountains" :key="index">
        {{ mountain.title }}
      </li>
    </ul>
  </div>
</template>
<script>
  export default {
    data() {
      return {
        mountains: []
      }
    },
    async fetch() {
      this.mountains = await fetch(
        'https://api.nuxtjs.dev/mountains'
      ).then(res => res.json())
    }
  }
</script>
```

Using `activated` hook

Nuxt will directly fill `this.$fetchState.timestamp` (timestamp) of the last `fetch` call (ssr included). You can use this property combined with `activated` hook to add a 30 seconds cache to `fetch`:

```
<template> ... </template>

<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    activated() {
      // Call fetch again if last fetch more than 30 sec ago
      if (this.$fetchState.timestamp <= Date.now() - 30000) {
        this.$fetch()
      }
    },
    async fetch() {
      this.posts = await fetch('https://api.nuxtjs.dev/posts').then(res =>
        res.json()
      )
    }
  }
</script>
```

The navigation to the same page will not call `fetch` if last `fetch` call was before 30 sec ago.

Async Data

Warning

`asyncData` is only available for pages and you don't have access to `this` inside the hook.

`asyncData` is another hook for universal data fetching. Unlike `fetch`, which requires you to set properties on the component instance (or dispatch Vuex actions) to save your async state, `asyncData` simply merges its return value into your component's local state. Here's an example using the @nuxt/http library:

```
<template>
  <div>
    <h1>{{ post.title }}</h1>
    <p>{{ post.description }}</p>
  </div>
</template>

<script>
  export default {
    async asyncData({ params, $http }) {
      const post = await $http.$get(`https://api.nuxtjs.dev/posts/${params.id}`)
      return { post }
    }
  }
</script>
```

Unlike `fetch`, the promise returned by the `asyncData` hook is resolved *during route transition*. This means that no "loading placeholder" is visible during client-side transitions (although the loading bar can be used to indicate a loading state to the user). Nuxt will instead wait for the `asyncData` hook to be finished before navigating to the next page or display the error page).

This hook can only be used for page-level components. Unlike `fetch`, `asyncData` cannot access the component instance (`this`). Instead, it receives the context as its argument. You can use it to fetch some data and Nuxt will automatically shallow merge the returned object with the component data.

In the upcoming examples, we are using @nuxt/http which we recommend for fetching data from an API.

Async data in components?

Because components do not have an `asyncData` method, you cannot directly fetch async data server side within a component. In order to get around this limitation you have three basic options:

1. Use the new `fetch` hook that is available in Nuxt 2.12 and later versions.
2. Make the API call in the `mounted` hook and set data properties when loaded. *Downside: Won't work for server side rendering.*
3. Make the API call in the `asyncData` method of the page component and pass the data as props to the sub components. Server rendering will work fine. *Downside: the `asyncData` of the page might be less readable because it's loading the data for other components*.

Listening to query changes

The `asyncData` method is not called on query string changes by default. If you want to change this behavior, for example when building a pagination component, you can set up parameters that should be listened to with the `watchQuery` property of your page component.

Next

Learn more about the watchQuery property and see the list of available keys in context.

Go to TOC

Meta Tags and SEO

Nuxt gives you 3 different ways to add meta data to your application:

div{.d-heading-description .leading-6}

- Globally using the nuxt.config.js
- Locally using the head as an object
- Locally using the head as a function so that you have access to data and computed properties.

Global Settings

Nuxt lets you define all default `<meta>` tags for your application inside the nuxt.config.js file using the head property. This is very useful for adding a default title and description tag for SEO purposes or for setting the viewport or adding the favicon.

```
export default {
  head: {
    title: 'my website title',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },
      {
        hid: 'description',
        name: 'description',
        content: 'my website description'
      }
    ],
    link: [{ rel: 'icon', type: 'image/x-icon', href: '/favicon.ico' }]
  }
}
```

Info

This will give you the same title and description on every page

Local Settings

You can also add titles and meta for each page by setting the `head` property inside your script tag on every page:

```
<script>
export default {
  head: {
    title: 'Home page',
    meta: [
      {
        hid: 'description',
        name: 'description',
        content: 'Home page description'
      }
    ],
```

```
  }
}
</script>
```

Info

Use `head` as an object to set a title and description only for the home page

```
<template>
  <h1>{{ title }}</h1>
</template>
<script>
  export default {
    data() {
      return {
        title: 'Home page'
      }
    },
    head() {
      return {
        title: this.title,
        meta: [
          {
            hid: 'description',
            name: 'description',
            content: 'Home page description'
          }
        ]
      }
    }
  }
</script>
```

Info

Use `head` as a function to set a title and description only for the home page. By using a function you have access to data and computed properties

Nuxt uses vue-meta to update the document head and meta attributes of your application.

Warning

To avoid any duplication when used in child components, please give a unique identifier with the `hid` key to the meta description. This way `vue-meta` will know that it has to overwrite the default tag.

Next

Learn more about the options available for `head`, in the vue-meta documentation.

External Resources

You can include external resources such as scripts and fonts by adding them globally to the `nuxt.config.js` or locally in the `head` object or function.

Info

You can also pass each resource an optional `body: true` to include the resource before the closing `</body>` tag.

Global Settings

```
export default {
  head: {
    script: [
      {
        src: 'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
      }
    ],
    link: [
      {
        rel: 'stylesheet',
        href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
      }
    ]
  }
}
```

Local Settings

```
<template>
  <h1>About page with jQuery and Roboto font</h1>
</template>

<script>
  export default {
    head() {
      return {
        script: [
          {
            src:
              'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
          }
        ],
        link: [
          {
            rel: 'stylesheet',
            href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
          }
        ]
      }
    }
  }
</script>

<style scoped>
  h1 {
    font-family: Roboto, sans-serif;
  }
</style>
```

Go to TOC

Configuration

By default, Nuxt is configured to cover most use cases. This default configuration can be overwritten with the nuxt.config.js file.

The css Property

Nuxt lets you define the CSS files/modules/libraries you want to set globally (included in every page).

Warning

In case you want to use `sass` make sure that you have installed the `sass` and `sass-loader` packages.

In `nuxt.config.js`, add the CSS resources:

```
export default {
  css: [
    // Load a Node.js module directly (here it's a Sass file)
    'bulma',
    // CSS file in the project
    '~/assets/css/main.css',
    // SCSS file in the project
    '~/assets/css/main.scss'
  ]
}
```

Warning

Nuxt will automatically guess the file type by its extension and use the appropriate pre-processor loader for webpack. You will still need to install the required loader if you need to use them.

Style Extensions

You can omit the file extension for CSS/SCSS/Postcss/Less/Stylus/... files listed in the css array in your nuxt config file.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

Warning

If you have two files with the same name, e.g. `main.scss` and `main.css`, and don't specify an extension in the css array entry, e.g. `css: ['~/assets/css/main']`, then only one file will be loaded depending on the order of `styleExtensions`. In this case only the `css` file will be loaded and the `scss` file will be ignored because `css` comes first in the default `styleExtension` array.

Default order: `['css', 'pcss', 'postcss', 'styl', 'stylus', 'scss', 'sass', 'less']`

Pre-processors

Thanks to Vue Loader, you can use any kind of pre-processor for your `<template>` or `<style>`: use the `lang` attribute.

Example of our `pages/index.vue` using Pug and Sass:

```
<template lang="pug">
  h1.red Hello {{ name }}!
</template>

<style lang="scss">
  .red {
    color: red;
  }
</style>
```

To use these pre-processors, we need to install their webpack loaders:

```
yarn add --dev pug pug-plain-loader
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev pug pug-plain-loader
npm install --save-dev sass sass-loader@10
```

External Resources

Global Settings

You can include your external resources in the head object or function. As described in the head API docs, the following examples shows the use of `head` as an object and as a function. If you want to use values from your Vue component like computed properties or data, you can use the `head()` function, returning the final head object. You can also pass each resource an optional `body: true` to include the resource before the closing `</body>` tag.

Include your resources in `nuxt.config.js` (here in the head object):

```
export default {
  head: {
    script: [
      {
        src: 'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
      }
    ],
    link: [
      {
        rel: 'stylesheet',
        href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
      }
    ]
  }
}
```

Local Settings

Include your resources in your `.vue` file inside the `pages/` directory (here in the head function):

```
<template>
  <h1>About page with jQuery and Roboto font</h1>
</template>

<script>
  export default {
    head() {
      return {
        script: [
          {
            src:
'https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js'
          }
        ],
        link: [
          {
            rel: 'stylesheet',
            href: 'https://fonts.googleapis.com/css?family=Roboto&display=swap'
          }
        ]
      }
    }
  }
</script>

<style scoped>
  h1 {
    font-family: Roboto, sans-serif;
  }
</style>
```

PostCSS plugins

If present, rename or delete the `postcss.config.js` in your project directory. Then, in your `nuxt.config.js` file add the following:

```
export default {
  build: {
    postcss: {
      // Add plugin names as key and arguments as value
      // Install them before as dependencies with npm or yarn
      plugins: {
        // Disable a plugin by passing false as value
        'postcss-url': false,
        'postcss-nested': {},
        'postcss-responsive-type': {},
        'postcss-hexrgba': {}
      },
      preset: {
        // Change the postcss-preset-env settings
        autoprefixer: {
          grid: true
        }
      }
```

```
    }
  }
}
```

JSX

Nuxt uses @nuxt/babel-preset-app, which is based on the official @vue/babel-preset-app for babel default configuration, so you can use JSX in your components.

You can also use JSX in the `render` method of your components:

```
export default {
  data () {
    return { name: 'World' }
  },
  render (h) {
    return <h1 class="red">{this.name}</h1>
  }
}
```

Aliasing `createElement` to `h` is a common convention you'll see in the Vue ecosystem but is actually optional for JSX since it automatically injects `const h = this.$createElement` in any method and getter (not functions or arrow functions) declared in ES2015 syntax that has JSX so you can drop the (h) parameter.

You can learn more about how to use it in the JSX section of the Vue.js documentation.

Ignoring files

.nuxtignore

You can use a `.nuxtignore` file to let Nuxt ignore `layout`, `page`, `store` and `middleware` files in your project's root directory (`rootDir`) during the build phase. The `.nuxtignore` file is subject to the same specification as `.gitignore` and `.eslintignore` files, in which each line is a glob pattern indicating which files should be ignored.

```
# ignore layout foo.vue
layouts/foo.vue
# ignore layout files whose name ends with -ignore.vue
layouts/*-ignore.vue
# ignore page bar.vue
pages/bar.vue
# ignore page inside ignore folder
pages/ignore/*.vue
# ignore store baz.js
```

```
store/baz.js

# ignore store files match _.test._

store/ignore/_.test._

# ignore middleware files under foo folder except foo/bar.js

middleware/foo/*.js !middleware/foo/bar.js
```

The ignorePrefix Property

Any file in pages/, layout/, middleware/ or store/ will be ignored during the build if its filename starts with the prefix specified by ignorePrefix.

By default all files which start with `-` will be ignored, such as `store/-foo.js` and `pages/-bar.vue`. This allows for co-locating tests, utilities, and components with their callers without themselves being converted into routes, stores, etc.

The ignore Property

More customizable than ignorePrefix: all files matching glob patterns specified inside ignore will be ignored in building.

```
export default {
  ignore: 'pages/bar.vue'
}
```

ignoreOptions

`nuxtignore` is using `node-ignore` under the hood, `ignoreOptions` can be configured as `options` of `node-ignore`.

```
export default {
  ignoreOptions: {
    ignorecase: false
  }
}
```

Extend webpack config

You can extend nuxt's webpack configuration via the `extend` option in your `nuxt.config.js`. The `extend` option of the `build` property is a method that accepts two arguments. The first argument is the webpack `config` object exported from nuxt's webpack config. The second parameter is a context object with the following boolean properties: `{ isDev, isClient, isServer, loaders }`.

```
export default {
  build: {
    extend(config, { isDev, isClient }) {
      // ...
      config.module.rules.push({
        test: /\.(ttf|eot|svg|woff(2)?)(\?[a-z0-9=&.]+)?$/,
        loader: 'file-loader'
```

```
      })
      // Sets webpack's mode to development if `isDev` is true.
      if (isDev) {
        config.mode = 'development'
      }
    }
  }
}
```

The `extend` method gets called twice - Once for the client bundle and the other for the server bundle.

Customize chunks configuration

You may want to tweak the optimization configuration a bit, avoiding a rewrite of the default object.

```
export default {
  build: {
    extend(config, { isClient }) {
      if (isClient) {
        config.optimization.splitChunks.maxSize = 200000
      }
    }
  }
}
```

Inspect webpack configuration

For complex projects and debugging it's sometimes useful to check what the final webpack configuration will look like. Luckily you can run `nuxt webpack` command from withing your project to output the configuration. Checkout this PR #7029 for more details.

Add webpack plugins

In your `nuxt.config.js` file, under the `build` option, you can pass webpack `plugins`, the same way you would do it in a `webpack.config.js` file.

In this example we add the webpack built-in ProvidePlugin for automatically loading JavaScript modules (*lodash* and *jQuery*) instead of having to `import` or `require` them everywhere.

```
import webpack from 'webpack'

export default {
  build: {
    plugins: [
      new webpack.ProvidePlugin({
        // global modules
        $: 'jquery',
        _: 'lodash'
      })
    ]
  }
}
```

> Note: You might not need jQuery in a Vue-based app.

With Nuxt, you can also control plugins execution context: if they are meant to be run on the `client` or in the `server` builds (or differentiating `dev` and `prod` builds) within `build.extend`, where you can manually pass webpack plugins too.

Extend Webpack to load audio files

Audio files should be processed by `file-loader`. This loader is already included in the default Webpack configuration, but it is not set up to handle audio files. You need to extend its default configuration in `nuxt.config.js`:

```
export default {
  build: {
    extend(config, ctx) {
      config.module.rules.push({
        test: /\.(ogg|mp3|wav|mpe?g)$/i,
        loader: 'file-loader',
        options: {
          name: '[path][name].[ext]'
        }
      })
    }
  }
}
```

You can now import audio files like this `<audio :src="require('@/assets/water.mp3')" controls></audio>`.

If you only want to write: `<audio src="@/assets/water.mp3" controls></audio>`, you need to tell `vue-loader` to automatically require your audio files when you reference them with the `src` attribute:

```
export default {
  build: {
    loaders: {
      vue: {
        transformAssetUrls: {
          audio: 'src'
        }
      }
    },

    extend(config, ctx) {
      config.module.rules.push({
        test: /\.(ogg|mp3|wav|mpe?g)$/i,
        loader: 'file-loader',
        options: {
          name: '[path][name].[ext]'
        }
      })
    }
  }
}
```

Edit host and port

By default, the Nuxt development server host is `localhost` which is only accessible from within the host machine. In order to view your app on another device you need to modify the host. You can modify the host in your nuxt.config.js file.

Host `'0.0.0.0'` is designated to tell Nuxt to resolve a host address, which is accessible to connections *outside* of the host machine (e.g. LAN). If the host is assigned the string value of `'0'` (not 0, which is falsy), or `'0.0.0.0'` your local IP address will be assigned to your Nuxt application.

```
export default {
  server: {
    host: '0' // default: localhost
  }
}
```

You can also change the port number from the default port of 3000.

```
export default {
  server: {
    port: 8000 // default: 3000
  }
}
```

Info

If the port is assigned the string value of `'0'` (not 0, which is falsy) a random port number will be assigned to your Nuxt application.

Although you can modify this in the nuxt.config.js file it is not advised to as it might cause you issues when hosting your site. It is much better to modify the host and port direct in the dev command.

```
HOST=0 PORT=8000 npm run dev
```

or create a script in your package.json

```
"scripts": {
  "dev:host": "nuxt --hostname '0' --port 8000"
}
```

Asynchronous Configuration

Although it is better to use the normal configuration `export default {}` you can have an async configuration by exporting an async function that return the config object.

```
import axios from 'axios'

export default async () => {
  const data = await axios.get('https://api.nuxtjs.dev/posts')
  return {
    head: {
      title: data.title
```

```
      //... rest of config
    }
  }
}
```

Warning

The axios-module cannot be used in `nuxt.config.js`. You will need to import axios and configure it again.

Further configuration

Next

The `nuxt.config.js` has way more customization and configuration options! Check out all its keys in the configuration glossary.

Go to TOC

Loading

Out of the box, Nuxt gives you its own loading progress bar component that's shown between routes. You can customize it, disable it or even create your own loading component.

Customizing the Progress Bar

Among other properties, the color, size, duration and direction of the progress bar can be customized to suit your application's needs. This is done by updating the `loading` property of the `nuxt.config.js` with the corresponding properties.

For example, to set a blue progress bar with a height of 5px, we update the `nuxt.config.js` to the following:

```
export default {
  loading: {
    color: 'blue',
    height: '5px'
  }
}
```

List of properties to customize the progress bar.

| Key | Type | Default | Description | | | | color | String | 'black' | CSS color of the progress bar | | | failedColor | String | 'red' | CSS color of the progress bar when an error appended while rendering the route (if data or fetch sent back an error for example). | | | height | String | '2px' | Height of the progress bar (used in the style property of the progress bar) | | | throttle | Number | 200 | In ms, wait for the specified time before displaying the progress bar. Useful for preventing the bar from flashing. | | | duration | Number | 5000 | In ms, the maximum duration of the progress bar, Nuxt assumes that the route will be rendered before 5 seconds. | | | continuous | Boolean | false | Keep animating progress bar when loading takes longer than duration. | | | css | Boolean | true | Set to false to remove default progress bar styles (and add your own). | | | rtl | Boolean | false | Set the direction of the progress bar from right to left. | |

Disable the Progress Bar

If you don't want to display the progress bar between the routes add `loading: false` in your `nuxt.config.js` file:

```
export default {
  loading: false
}
```

The loading property gives you the option to disable the default loading progress bar on a specific page.

```
<template>
  <h1>My page</h1>
</template>
```

```
<script>
  export default {
    loading: false
  }
</script>
```

Programmatically starting the loading bar

The loading bar can also be programmatically started in your components by calling `this.$nuxt.$loading.start()` to start the loading bar and `this.$nuxt.$loading.finish()` to finish it.

During your page component's mounting process, the `$loading` property may not be immediately available to access. To work around this, if you want to start the loader in the `mounted` method, make sure to wrap your `$loading` method calls inside `this.$nextTick` as shown below.

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Internals of the Progress Bar

Unfortunately, it is not possible for the Loading component to know in advance how long loading a new page will take. Therefore, it is not possible to accurately animate the progress bar to 100% of the loading time.

Nuxt's loading component partially solves this by letting you set the `duration`, this should be set to an estimate of how long the loading process will take. Unless you use a custom loading component, the progress bar will always move from 0% to 100% in `duration` time (regardless of actual progression). When the loading takes longer than `duration` time, the progress bar will stay at 100% until the loading finishes.

You can change the default behavior by setting `continuous` to true, then after reaching 100% the progress bar will start shrinking back to 0% again in `duration` time. When the loading is still not finished after reaching 0% it will start growing from 0% to 100% again, this repeats until the loading finishes.

```
export default {
  loading: {
    continuous: true
  }
}
```

Example of a continuous progress bar:

../../../../static/img/docs/api-continuous-loading.gif

Using a Custom Loading Component

You can also create your own component that Nuxt will call instead of the default loading progress bar component. To do so, you need to give a path to your component in the `loading` option. Then, your component will be called directly by Nuxt.

Your component has to expose some of these methods:

| Method | Required | Description | | | start() | Required | Called when a route changes, this is where you display your component. | | finish() | Required | Called when a route is loaded (and data fetched), this is where you hide your component. | | fail(error) | Optional | Called when a route couldn't be loaded (failed to fetch data for example). | | increase(num) | Optional | Called during loading the route component, num is an Integer < 100. |

You can create your custom component in `components/LoadingBar.vue`:

```
<template>
  <div v-if="loading" class="loading-page">
    <p>Loading...</p>
  </div>
</template>

<script>
  export default {
    data: () => ({
      loading: false
    }),
    methods: {
      start() {
        this.loading = true
      },
      finish() {
        this.loading = false
      }
    }
  }
</script>

<style scoped>
  .loading-page {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(255, 255, 255, 0.8);
    text-align: center;
    padding-top: 200px;
    font-size: 30px;
    font-family: sans-serif;
  }
</style>
```

Then, you update your `nuxt.config.js` to tell Nuxt to use your component:

```
export default {
  loading: '~/components/LoadingBar.vue'
}
```

The loading indicator Property

When running Nuxt in SPA mode, there is no content from the server side on the first page load. So, instead of showing a blank page while the page loads, Nuxt gives you a spinner which you can customize to add your own colors or background and even change the the indicator.

```
export default {
  loadingIndicator: {
    name: 'circle',
    color: '#3B8070',
    background: 'white'
  }
}
```

Built-in indicators

These indicators are imported from the awesome SpinKit project. You can check out its demo page to preview the spinners. In order to use one of these spinners all you have to do is add its name to the name property. No need to import or install anything. Here is a list of built in indicators you can use.

- circle
- cube-grid
- fading-circle
- folding-cube
- chasing-dots
- nuxt
- pulse
- rectangle-bounce
- rotating-plane
- three-bounce
- wandering-cubes

Built-in indicators support `color` and `background` options.

Custom indicators

If you need your own special indicator, a String value or Name key can also be a path to an HTML template of indicator source code! All of the options are passed to the template, too.

Nuxt's built-in source code is also available if you need a base!

Go to TOC

Nuxt Components

Nuxt comes with a few important components included out of the box, which will be helpful when building your application. The components are globally available, which means that you don't need to import them in order to use them.

In the following paragraphs, each of the included components is explained.

The Nuxt Component

The `<Nuxt>` component is the component you use to display your page components. Basically, this component gets replaced by what is inside your page components depending on the page that is being shown. Therefore it is important that you add the `<Nuxt>` component to your layouts.

```
<template>
  <div>
    <div>My nav bar</div>
    <Nuxt />
    <div>My footer</div>
  </div>
</template>
```

Warning

The `<Nuxt>` component can only be used inside layouts.

The `<Nuxt>` component can take the prop of `nuxt-child-key`. This prop will be passed to `<RouterView>` so that your transitions will work correctly inside dynamic pages.

There are 2 ways to handle the internal `key` prop of `<RouterView>`.

1. Use a `nuxtChildKey` prop on your `<Nuxt>` component

```
<template>
  <div>
    <Nuxt :nuxt-child-key="someKey" />
  </div>
</template>
```

2. Add the `key` option in *page* components as `string` or `function`

```
export default {
  key(route) {
    return route.fullPath
  }
}
```

The NuxtChild Component

This component is used for displaying the children components in a nested route.

Example:

```
-| pages/
---| parent/
| child.vue
---| parent.vue
```

This file tree will generate these routes:

```
;[
  {
    path: '/parent',
    component: '~/pages/parent.vue',
    name: 'parent',
    children: [
      {
        path: 'child',
        component: '~/pages/parent/child.vue',
        name: 'parent-child'
      }
    ]
  }
]
```

To display the `child.vue` component, you have to insert the `<NuxtChild>` component inside `pages/parent.vue`:

```
<template>
  <div>
    <h1>I am the parent view</h1>
    <NuxtChild :foobar="123" />
  </div>
</template>
```

keep-alive

Both, the `<Nuxt>` component and the `<NuxtChild/>` component, accept `keep-alive` and `keep-alive-props`.

Info

To learn more about keep-alive and keep-alive-props see the vue docs

```
<template>
  <div>
    <Nuxt keep-alive :keep-alive-props="{ exclude: ['modal'] }" />
  </div>
</template>

<!-- will be converted into something like this -->
<div>
```

```
  <KeepAlive :exclude="['modal']">
    <RouterView />
  </KeepAlive>
</div>
```

```
<template>
  <div>
    <NuxtChild keep-alive :keep-alive-props="{ exclude: ['modal'] }" />
  </div>
</template>

<!-- will be converted into something like this -->
<div>
  <KeepAlive :exclude="['modal']">
    <RouterView />
  </KeepAlive>
</div>
```

`<NuxtChild>` components can also receive properties like a regular Vue component.

```
<template>
  <div>
    <NuxtChild :key="$route.params.id" />
  </div>
</template>
```

To see an example, take a look at the nested-routes example.

:code-sandbox{src="csb_link_nuxt"}

The NuxtLink Component

To navigate between pages of your app, you should use the `<NuxtLink>` component. This component is included with Nuxt and therefore you don't have to import it like you do with other components. It is similar to the HTML `<a>` tag except that instead of using a `href="/about"` you use `to="/about"`. If you've used `vue-router` before, you can think of `<NuxtLink>` as a replacement of `<RouterLink>`

A simple link to the `index.vue` page in your `pages` folder:

```
<template>
  <NuxtLink to="/">Home page</NuxtLink>
</template>
```

The `<NuxtLink>` component should be used for all internal links. That means for all links to the pages within your site you should use `<NuxtLink>`. The `<a>` tag should be used for all external links. That means if you have links to other websites you should use the `<a>` tag for those.

```
<template>
  <div>
    <h1>Home page</h1>
    <NuxtLink to="/about"
      >About (internal link that belongs to the Nuxt App)</NuxtLink
    >
```

```
      <a href="https://nuxtjs.org">External Link to another page</a>
    </div>
  </template>
```

Info

If you want to know more about `<RouterLink>`, feel free to read the Vue Router documentation for more information.

Info

`<NuxtLink>` also comes with smart prefetching out of the box.

prefetchLinks

Nuxt automatically includes smart prefetching. That means it detects when a link is visible, either in the viewport or when scrolling and prefetches the JavaScript for those pages so that they are ready when the user clicks the link. Nuxt only loads the resources when the browser isn't busy and skips prefetching if your connection is offline or if you only have 2g connection.

Disable prefetching for specific links

However sometimes you may want to disable prefetching on some links if your page has a lot of JavaScript or you have a lot of different pages that would be prefetched or you have a lot of third party scripts that need to be loaded. To disable the prefetching on a specific link, you can use the `no-prefetch` prop. Since Nuxt v2.10.0, you can also use the `prefetch` prop set to `false`

```
<NuxtLink to="/about" no-prefetch>About page not pre-fetched</NuxtLink>
<NuxtLink to="/about" :prefetch="false">About page not pre-fetched</NuxtLink>
```

Disable prefetching globally

To disable the prefetching on all links, set the `prefetchLinks` to `false`:

```
export default {
  router: {
    prefetchLinks: false
  }
}
```

Since Nuxt v2.10.0, if you have set `prefetchLinks` to `false` but you want to prefetch a specific link, you can use the `prefetch` prop:

```
<NuxtLink to="/about" prefetch>About page pre-fetched</NuxtLink>
```

linkActiveClass

The `linkActiveClass` works the same as the `vue-router` class for active links. If we want to show which links are active all you have to do is create some css for the class `nuxt-link-active` .

```
.nuxt-link-active {
  color: red;
}
```

alert This css can be added to the navigation component or for a specific page or layout or in your main.css file.

If you want to you can also configure the class name to be something else. You can do this by modifying the `linkActiveClass` in the router property in your `nuxt.config.js` file.

```
export default {
  router: {
    linkActiveClass: 'my-custom-active-link'
  }
}
```

Info

This option is given directly to the `vue-router` linkActiveClass. See the vue-router docs for more info.

linkExactActiveClass

The `linkExactActiveClass` works the same as the `vue-router` class for exact active links. If we want to show which links are active with an exact match all you have to do is create some css for the class `nuxt-link-exact-active` .

```
.nuxt-link-exact-active {
  color: green;
}
```

Info

This css can be added to the navigation component or for a specific page or layout or in your main.css file.

If you want to you can also configure the class name to be something else. You con do this by modifying the `linkExactActiveClass` in the router property in your `nuxt.config.js` file.

```
export default {
  router: {
    linkExactActiveClass: 'my-custom-exact-active-link'
  }
}
```

Info

This option is given directly to the `vue-router` linkExactActiveClass. See the vue-router docs for more info

linkPrefetchedClass

The linkPrefetchedClass will allow you to add styles for all links that have been prefetched. This is great for testing which links are being prefetched after modifying the default behavior. The linkPrefetchedClass is disabled by default. If you want to enable it you need to add it to the router property in your `nuxt-config.js` file.

```
export default {
  router: {
    linkPrefetchedClass: 'nuxt-link-prefetched'
  }
}
```

Then you can add the styles for that class.

```
.nuxt-link-prefetched {
  color: orangeRed;
}
```

Info

In this example we have used the class `nuxt-link-prefetched` but you can name it anything you like

:code-sandbox{src="csb_link_nuxt_link"}

The client-only Component

This component is used to purposely render a component only on client-side. To import a component only on the client, register the component in a client-side only plugin.

```
<template>
  <div>
    <sidebar />
    <client-only placeholder="Loading...">
      <!-- this component will only be rendered on client-side -->
      <comments />
    </client-only>
  </div>
</template>
```

Use a slot as placeholder until `<client-only />` is mounted on client-side.

```
<template>
  <div>
    <sidebar />
    <client-only>
      <!-- this component will only be rendered on client-side -->
      <comments />

      <!-- loading indicator, rendered on server-side -->
      <template #placeholder>
        <comments-placeholder />
      </template>
```

```
    </client-only>
  </div>
</template>
```

Info

Sometimes in server rendered pages `$refs` inside `<client-only>` might not be ready even with `$nextTick`, the trick might be to call `$nextTick` a couple of times:

```
mounted(){
  this.initClientOnlyComp()
},
methods: {
  initClientOnlyComp(count = 10) {
    this.$nextTick(() => {
      if (this.$refs.myComp) {
        //...
      } else if (count > 0) {
        this.initClientOnlyComp(count - 1);
      }
    });
  },
}
```

alert If you are using a version of Nuxt < v2.9.0, use `<no-ssr>` instead of `<client-only>`

Go to TOC

Enabling Auto-Discovery

Starting from `v2.13`, Nuxt can auto import your components when used in your templates. To activate this feature, set `components: true` in your configuration:

```
export default {
  components: true
}
```

Info

Check out how to configure component auto-discovery.

Using Components

Once you create your components in the components directory they will then be available throughout your app without the need to import them.

```
| components/
--| TheHeader.vue
--| TheFooter.vue
```

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <TheFooter />
  </div>
</template>
```

Info

See live demo or video example.

Component Names

If you have components in nested directories such as:

```
| components/
--| base/
----| foo/
| Button.vue
```

The component name will be based on its own path directory and filename. Therefore, the component will be:

```
<BaseFooButton />
```

alert For clarity, it is recommend that the component file name matches its name. (So, in the example above, you could rename `Button.vue` to be `BaseFooButton.vue`.)

If you want to use a custom directory structure that should not be part of the component name, you can explicitly specify these directories:

```
components/
--| base/
----| foo/
| Button.vue
```

```
components: {
  dirs: [
    '~/components',
    '~/components/base'
  ]
}
```

And now in your template you can use `FooButton` instead of `BaseFooButton`.

```
<FooButton />
```

Info

Consider naming your components and directories following the Vue Style Guide.

Dynamic Imports

To dynamically import a component (also known as lazy-loading a component) all you need to do is add the `Lazy` prefix to the component name.

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <LazyTheFooter />
  </div>
</template>
```

This is particularly useful if the component is not always needed. By using the `Lazy` prefix you can delay loading the component code until the right moment, which can be helpful for optimizing your JavaScript bundle size.

```
<template>
  <div>
    <h1>Mountains</h1>
    <LazyMountainsList v-if="show" />
    <button v-if="!show" @click="show = true">Show List</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      show: false
    }
```

```
    }
  }
  </script>
```

Cheatsheet

:modal{src="img" alt="imgAlt"}

Go to TOC

Nuxt build directory

The `.nuxt` directory is the so-called *build directory*. It is dynamically generated and hidden by default. Inside the directory you can find automatically generated files when using `nuxt dev` or your build artifacts when using `nuxt build`. Modifying these files is great for debugging but remember that they are generated files and once you run the `dev` or `build` command again, anything that was saved here will be regenerated.

Warning

The `.nuxt` directory should not be committed to your version control system and should be ignored through your `.gitignore` as it will be generated automatically when executing `nuxt dev` or `nuxt build`.

The buildDir Property

By default, many tools assume that `.nuxt` is a hidden directory, because its name starts with a dot. You can use the buildDir option to prevent that. If you do change the name remember to add the new name to your `.gitignore` file.

```
export default {
  buildDir: 'nuxt-dist'
}
```

Inside the .nuxt folder:

- The router.js file is the generated router file that Nuxt generates for you when you put `.vue` files inside the pages folder. You can use this file for debugging for when you want to look up which routes are generated for vue-router and find out the names of a specific route.
- The router.scrollBehavior.js which is your Router ScrollBehavior
- The Components folder has all your Nuxt components such as NuxtChild and NuxtLink. It also contains the nuxt-build-indicator which is the page we see when your application is building and nuxt-loading which is your loading component that gets seen when we are waiting for your page to load. You will also find the nuxt-error page in here which contains the Nuxt default error page.
- The mixins folder has the files needed for the Nuxt `$fetch` method.
- The views folder contains your app template and your server error page.
- The app.js is your main application file.
- The client.js file is your client file needed for everything that happens client side.
- The empty file is intentionally left empty for no-op aliases
- The index.js file bootstraps your application.
- The loading.html is the file that is used when the page is loading.
- The middleware file is where your middleware is kept
- The server.js file is all the code that is ran on the server
- the utilities contains the utilities that Nuxt needs for it to work.

Deploying

The `.nuxt` folder is part of the files needed to deploy your SSR application. It is not needed for deploying your static Nuxt app though because we use the `dist` folder for that.

Go to TOC

Plugins directory

The `plugins` directory contains your Javascript plugins that you want to run before instantiating the root Vue.js Application.

Nuxt Plugins

Nuxt v2.14.1
August 2020

3 ways to extend functionalities of Nuxt app using plugins

$root refers to root Vue instance.

1 injected in $root & context

make functions or values available on client-side and server-side

```
//plugins/hello.js
export default (context, inject) =>
{
 inject('hello', msg => console.log(msg))
}
```

*access **hello** on:*

client-side: `this.$hello('mounted')`

server-side: `$hello('asyncData')`

context refers to Nuxt context. You can access keys like **app**, **store**, etc. And **inject** is a method: *`inject(key, value)`*

2 Vue plugin

use Vue plugins in your Nuxt application

```
//plugins/vue-tooltip.js
import Vue from 'vue'
import VTooltip from 'v-tooltip'
Vue.use(VTooltip)
```

3 External packages or modules

use external packages/modules in your Nuxt application

```
// plugins/axios.js
export default function ({ $axios, redirect }) {
  $axios.onError(error => {
    if(error.response.status === 404) {
      redirect('/sorry')
    }
  })
}
```

Include Nuxt plugin using **plugins** property

○ Only in client-side
● Only in server-side
◉ Both client & server

type: array

Array of objects

```
// nuxt.config.js
export default {
 plugins: [
  ○ {src:'~/plugins/vue-tooltip.js', mode:'client'},
  ● {src:'~/plugins/bar.js', mode:'server'},
  ◉ {src:'~/plugins/axios.js'}
 ]
}
```

Array of strings

```
// nuxt.config.js
export default {
 plugins: [
  ○ '~/plugins/vue-tooltip.client.js',
  ● '~/plugins/bar.server.js',
  ◉ '~/plugins/axios.js'
 ]
}
```

If plugin exports ES6 module,use transpile build option!

```
module.exports = {
build: {
  transpile:['vue-tooltip']
 }
}
```

This is the place to add Vue plugins and to inject functions or constants. Every time you need to use `Vue.use()`, you should create a file in `plugins/` and add its path to `plugins` in `nuxt.config.js`.

External Packages

You may want to use external packages/modules in your application (one great example is axios) for making HTTP requests for both server and client.

First, install it via npm or Yarn.

```
yarn add @nuxtjs/axios
```

```
npm install @nuxtjs/axios
```

You can configure for example the axios interceptors to react on possible errors from your API calls across the application. In this example we redirect the user to a custom error page called sorry when we get a 500 status error from our API.

```
export default function ({ $axios, redirect }) {
  $axios.onError(error => {
    if (error.response.status === 500) {
      redirect('/sorry')
    }
  })
}
```

Last but not least, add the module and the newly created plugin to the project configuration.

```
module.exports = {
  modules: ['@nuxtjs/axios'],
  plugins: ['~/plugins/axios.js']
}
```

Then we can use it directly in your page components:

```
<template>
  <h1>{{ post.title }}</h1>
</template>

<script>
export default {
    async asyncData ({ $axios, params }) {
      const  post  = await
$axios.$get(`https://api.nuxtjs.dev/posts/${params.id}`)
      return { post }
    }
}
</script>
```

Another way to use `axios` without installing the module is by importing `axios` direct in the `<script>` tag.

```
<script>
import axios from 'axios'

export default {
    async asyncData ({ params }) {
      const { data: post }  = await
axios.get(`https://api.nuxtjs.dev/posts/${params.id}`)
      return { post }
    }
}
</script>
```

Info

If you get an *Cannot use import statement outside a module* error, you may need to add your package to the `build` > `transpile` option in `nuxt.config.js` for webpack loader to make your plugin available.

```
build: {
  // You can extend webpack config here
  transpile: ['npm-package-name'],
},
```

Vue Plugins

If we want to use Vue plugins, like v-tooltip to display tooltips in your application, we need to setup the plugin before launching the app.

First we need to install it

```
yarn add v-tooltip
```

```
npm install v-tooltip
```

Then we create the file `plugins/vue-tooltip.js`

```
import Vue from 'vue'
import VTooltip from 'v-tooltip'

Vue.use(VTooltip)
```

The plugins Property

Then we add the file path inside the `plugins` key of our `nuxt.config.js`. The plugins property lets you add Vue.js plugins easily to your main application. All the paths defined in the `plugins` property will be imported before initializing the main application.

```
export default {
  plugins: ['~/plugins/vue-tooltip.js']
}
```

ES6 Plugins

If the plugin is located in `node_modules` and exports an ES6 module, you may need to add it to the `transpile` build option:

```
module.exports = {
  build: {
    transpile: ['vue-tooltip']
  }
}
```

You can refer to the configuration build docs for more build options.

Client or server side only

Some plugins might work only in the browser because they lack SSR support.

Name conventional plugin

If a plugin is assumed to be run only on client or server side, `.client.js` or `.server.js` can be applied as an extension of the plugin file. The file will be automatically included only on the respective (client or server) side.

```
export default {
  plugins: [
    '~/plugins/foo.client.js', // only in client side
    '~/plugins/bar.server.js', // only in server side
    '~/plugins/baz.js' // both client & server
  ]
}
```

Object syntax

You can also use the object syntax with the `mode` property (`'client'` or `'server'`) in `plugins`.

```
export default {
  plugins: [
    { src: '~/plugins/both-sides.js' },
    { src: '~/plugins/client-only.js', mode: 'client' }, // only on client side
    { src: '~/plugins/server-only.js', mode: 'server' } // only on server side
  ]
}
```

Inject in `$root` & context

Sometimes you want to make functions or values available across your app. You can inject those variables into Vue instances (client side), the context (server side) and even in the Vuex store. It is a convention to prefix those functions with a `$`.

Nuxt provides you with an `inject(key, value)` method to do this easily. Inject is given as the second parameter when exporting a function. The `$` will be prepended automatically to the key.

Info

It is important to know that in any Vue instance lifecycle, only `beforeCreate` and `created` hooks are called both, from client-side and server-side. All other hooks are called only from the client-side.

```
export default ({ app }, inject) => {
  // Inject $hello(msg) in Vue, context and store.
  inject('hello', msg => console.log(`Hello ${msg}!`))
}
```

```
export default {
  plugins: ['~/plugins/hello.js']
}
```

Now `$hello` service can be accessed from `context` and `this` in pages, components, plugins, and store actions.

```
export default {
  mounted() {
    this.$hello('mounted')
    // will console.log 'Hello mounted!'
  },
  asyncData({ app, $hello }) {
    $hello('asyncData')
    // If using Nuxt <= 2.12, use 👇
    app.$hello('asyncData')
  }
}
```

```
export const state = () => ({
  someValue: ''
})

export const actions = {
  setSomeValueToWhatever({ commit }) {
    this.$hello('store action')
    const newValue = 'whatever'
    commit('changeSomeValue', newValue)
  }
}
```

Warning

Don't use `Vue.use()`, `Vue.component()`, and globally, don't plug anything in Vue **inside** this function, dedicated to Nuxt injection. It will cause memory leak on server-side.

The extendPlugins Property

You may want to extend plugins or change the plugins order created by Nuxt. This function accepts an array of plugin objects and should return an array of plugin objects.

Example of changing plugins order:

```
export default {
  extendPlugins(plugins) {
    const pluginIndex = plugins.findIndex(
      ({ src }) => src === '~/plugins/shouldBeFirst.js'
    )
    const shouldBeFirstPlugin = plugins[pluginIndex]

    plugins.splice(pluginIndex, 1)
    plugins.unshift(shouldBeFirstPlugin)

    return plugins
  }
}
```

Global mixins

Global mixins can be easily added with Nuxt plugins but can cause trouble and memory leaks when not handled correctly. Whenever you add a global mixin to your application, you should use a flag to avoid registering it multiple times:

```
import Vue from "vue"

// Make sure to pick a unique name for the flag
// so it won't conflict with any other mixin.
if (!Vue.__my_mixin__) {
  Vue.__my_mixin__ = true
  Vue.mixin({ ... }) // Set up your mixin then
}
```

Go to TOC

Static directory

The `static` directory is directly mapped to the server root () and contains files that likely won't be changed. All included files will be automatically served by Nuxt and are accessible through your project root URL.

`/static/robots.txt` will be available at `http://localhost:3000/robots.txt`

`/static/favicon.ico` will be available at `http://localhost:3000/favicon.ico`

This option is helpful for files like `robots.txt`, `sitemap.xml` or `CNAME` (which is important for GitHub Pages deployment).

Warning

This directory cannot be renamed without extra configuration.

Static Assets

If you don't want to use Webpack assets from the `assets` directory, you can add the images to the static directory.

In your code, you can then reference these files relative to the root (`/`):

```
<!-- Static image from static directory -->
<img src="/my-image.png" />

<!-- webpacked image from assets directory -->
<img src="~/assets/my-image-2.png" />
```

Info

Nuxt doesn't change this path, so if you customize your `router.base` then you'll need to make sure to add that manually to your paths. For example:

```
<img :src="`${yourPrefix}/my-image.png`" />
```

Static Directory Config

Should you need to you can configure the `static/` directory behavior in the `nuxt.config.js` file.

Static asset Prefix

If you deploy Nuxt to a subfolder, e.g. `/blog/`, the router base will be added to the static asset path by default. If you want to disable this behavior, you can set `static.prefix` to false in the `nuxt.config.js`.

```
export default {
  static: {
    prefix: false
  }
}
```

Default: `/blog/my-image.png`

With `static.prefix` disabled: `/my-image.png`

Go to TOC

Store directory

The `store` directory contains your Vuex Store files. The Vuex Store comes with Nuxt out of the box but is disabled by default. Creating an `index.js` file in this directory enables the store.

Warning

This directory cannot be renamed without extra configuration.

Using a store to manage the state is important for every big application. That's why Nuxt implements Vuex in its core.

Activate the Store

Nuxt will look for the `store` directory. If it contains a file, that isn't a hidden file or a `README.md` file, then the store will be activated. This means that Nuxt will:

1. Import Vuex,
2. Add the `store` option to the root Vue instance.

Modules

Every `.js` file inside the `store` directory is transformed as a namespaced module (`index` being the root module). Your `state` value should always be a `function` to avoid unwanted *shared* state on the server side.

To get started, export the state as a function, and the mutations and actions as objects.

```
export const state = () => ({
  counter: 0
})

export const mutations = {
  increment(state) {
    state.counter++
  }
}
```

Then, you can have a `store/todos.js` file:

```
export const state = () => ({
  list: []
})

export const mutations = {
  add(state, text) {
    state.list.push({
      text,
      done: false
    })
```

```
    },
    remove(state, { todo }) {
      state.list.splice(state.list.indexOf(todo), 1)
    },
    toggle(state, todo) {
      todo.done = !todo.done
    }
  }
```

The store will be created as such:

```
new Vuex.Store({
  state: () => ({
    counter: 0
  }),
  mutations: {
    increment(state) {
      state.counter++
    }
  },
  modules: {
    todos: {
      namespaced: true,
      state: () => ({
        list: []
      }),
      mutations: {
        add(state, { text }) {
          state.list.push({
            text,
            done: false
          })
        },
        remove(state, { todo }) {
          state.list.splice(state.list.indexOf(todo), 1)
        },
        toggle(state, { todo }) {
          todo.done = !todo.done
        }
      }
    }
  }
})
```

And in your `pages/todos.vue`, using the `todos` module:

```
<template>
  <ul>
    <li v-for="todo in todos" :key="todo.text">
      <input :checked="todo.done" @change="toggle(todo)" type="checkbox">
      <span :class="{ done: todo.done }">{{ todo.text }}</span>
    </li>
    <li><input @keyup.enter="addTodo" placeholder="What needs to be done?"></li>
  </ul>
</template>

<script>
import { mapMutations } from 'vuex'

export default {
```

```
  computed: {
    todos () {
      return this.$store.state.todos.list
    }
  },
  methods: {
    addTodo (e) {
      this.$store.commit('todos/add', e.target.value)
      e.target.value = ''
    },
    ...mapMutations({
      toggle: 'todos/toggle'
    })
  }
}
</script>

<style>
.done {
  text-decoration: line-through;
}
</style>
```

The module method also works for top-level definitions without implementing a sub-directory in the store directory.

Example for state: you create a file `store/state.js` and add the following.

```
export default () => ({
  counter: 0
})
```

And the corresponding mutations can be in the file `store/mutations.js`

```
export default {
  increment(state) {
    state.counter++
  }
}
```

Example folder structure

A complex store setup file/folder structure might look like this:

```
 store/
--| index.js
--| ui.js
--| shop/
----| cart/
| actions.js
| getters.js
| mutations.js
| state.js
----| products/
| mutations.js
| state.js
| itemsGroup1/
| state.js
```

Plugins in the Store

You can add additional plugins to the store by putting them into the `store/index.js` file:

```
import myPlugin from 'myPlugin'

export const plugins = [myPlugin]

export const state = () => ({
  counter: 0
})

export const mutations = {
  increment(state) {
    state.counter++
  }
}
```

More information about the plugins: Vuex documentation.

The nuxtServerInit Action

If the action `nuxtServerInit` is defined in the store and the mode is `universal`, Nuxt will call it with the context (only from the server-side). It's useful when we have some data on the server we want to give directly to the client-side.

For example, let's say we have sessions on the server-side and we can access the connected user through `req.session.user`. To add the authenticated user to our store, we update our `store/index.js` to the following:

```
actions: {
  nuxtServerInit ({ commit }, { req }) {
    if (req.session.user) {
      commit('user', req.session.user)
    }
  }
}
```

Warning

Only the primary module (in `store/index.js`) will receive this action. You'll need to chain your module actions from there.

The context is given to `nuxtServerInit` as the 2nd argument in the `asyncData` method.

If `nuxt generate` is ran, `nuxtServerInit` will be executed for every dynamic route generated.

Info

Asynchronous `nuxtServerInit` actions must return a Promise or leverage async/await to allow the nuxt server to wait on them.

```
actions: {
  async nuxtServerInit({ dispatch }) {
    await dispatch('core/load')
  }
}
```

Vuex Strict Mode

Strict mode is enabled by default on dev mode and turned off in production mode. To disable strict mode in dev, follow the below example in `store/index.js`:

```
export const strict = false
```

Go to TOC

Nuxt configuration file

By default, Nuxt is configured to cover most use cases. This default configuration can be overwritten with the nuxt.config.js file.

---

nuxt.config.js

alias

This option lets you define aliases that will be available within your JavaScript and CSS.

```
import { resolve } from 'path'

export default {
  alias: {
    'style': resolve(__dirname, './assets/style')
  }
}
```

Next

See more on the alias property

build

This option lets you configure various settings for the `build` step, including `loaders`, `filenames`, the `webpack` config and `transpilation`.

```
export default {
  build: {
    /*
     ** You can extend webpack config here
     */
    extend(config, ctx) {}
  }
}
```

Next

See more on the build property

css

This option lets you define the CSS files, modules, and libraries you want to include globally (on every page).

```
export default {
  css: ['~/assets/css/main.css', '~/assets/css/animations.scss']
}
```

You can omit the file extension for CSS, SCSS, Postcss, Less, Stylus, ... files listed in the css array in your nuxt config file.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

By omitting the extension, if you have a css file and decide to change to use sass for example, you won't have to update your nuxt.config as it will use the new extension once the filename remains the same.

Next

See more on the css property

dev

This option lets you define the `development` or `production` mode of Nuxt (important when you use Nuxt programmatically)

```
export default {
  dev: process.env.NODE_ENV !== 'production'
}
```

Next

See more on the dev property

env

This option lets you define environment variables that are required at build time (rather than runtime) such as `NODE_ENV=staging` or `VERSION=1.2.3`. However, for runtime environment variables `runtimeConfig` is required.

```
export default {
  env: {
    baseURL: process.env.BASE_URL
  }
}
```

runtimeConfig

The runtime config has built-in dotenv support for better security and faster development. The runtime config is added to the Nuxt payload so there is no need to rebuild in order to update the runtime configuration when working in development or with server-side rendering or client-side only applications. (For static sites you will still need to regenerate your site to see changes.)

`.env` support

If you have a `.env` file in your project root directory, it will be automatically loaded into `process.env` and accessible within your `nuxt.config` / `serverMiddleware` and any other files they import.

You can customize the path by using `--dotenv <file>` or disable entirely with `--dotenv false`. For example, you might specify a different `.env` file in production, staging or development environments.

`publicRuntimeConfig`

- should hold all env variables that are public as these will be exposed on the frontend. This could include a reference to your public URL for example.
- is available using `$config` in both server and client.

```
export default {
  publicRuntimeConfig: {
    baseURL: process.env.BASE_URL || 'https://nuxtjs.org'
  }
}
```

`privateRuntimeConfig`

- should hold all env variables that are private and that should not be exposed on the frontend. This could include a reference to your API secret tokens for example.
- is only available on server using same `$config` (it overrides publicRuntimeConfig)

```
export default {
  privateRuntimeConfig: {
    apiSecret: process.env.API_SECRET
  }
}
```

Using your config values:

You can then access these values anywhere by using the context in your pages, store, components and plugins by using `this.$config` or `context.$config`.

```
<script>
  asyncData ({ $config: { baseURL } }) {
    const posts = await fetch(`${baseURL}/posts`)
      .then(res => res.json())
  }
</script>
```

Inside your templates you can access your runtimeConfigs directly using `$config.*`

```
<template>
  <p>Our Url is: {{ $config.baseURL}}</p>
</template>
```

Warning

Your private config could be exposed if you use `$config` outside of a server-only context (for example, if you use `$config` in `fetch`, `asyncData` or directly inside your template).

Next

See more on the runtimeConfig

Next

See our blog post on Moving from @nuxtjs/dotenv to runtime config

Next

See more on the env property

generate

This option lets you set up parameter values for every dynamic route in your application that will be transformed into HTML files by Nuxt.

```
export default {
  generate: {
    dir: 'gh_pages', // gh_pages/ instead of dist/
    subFolders: false // HTML files are generated according to the route path
  }
}
```

Next

See more on the generate property

head

```
export default {
    head: {
    title: 'my title',
    meta: [
      { charset: 'utf-8' },
           .....
       ]
    }
}
```

This option lets you define all default meta tags for your application.

Next

See more on head integration

loading

This option lets you customize the loading component that Nuxt uses by default.

```
export default {
  loading: {
    color: '#fff'
  }
}
```

Next

See more on loading integration

modules

With this option you can add Nuxt modules to your project.

```
export default {
  modules: ['@nuxtjs/axios']
}
```

Next

See more on the modules property

modulesDir

The modulesDir property is used to set the modules directories for path resolving. For example: Webpack's resolveLoading, nodeExternals and postcss. The configuration path is relative to `options.rootDir` (default: process.cwd()).

```
export default {
  modulesDir: ['../../node_modules']
}
```

Setting this field may be necessary if your project is organized as a Yarn workspace-styled mono-repository.

Next

See more on the modulesDir property

plugins

This option lets you define JavaScript plugins that should be run before instantiating the root Vue.js application.

```
export default {
  plugins: ['~/plugins/url-helpers.js']
}
```

Next

See more on the plugins property

router

With the `router` option you can overwrite the default Nuxt configuration of Vue Router.

```
export default {
  router: {
    linkExactActiveClass: 'text-primary'
  }
}
```

Next

See more on the router property

server

This option lets you configure the connection variables for the server instance of your Nuxt application.

```
import path from 'path'
import fs from 'fs'

export default {
  server: {
    https: {
      key: fs.readFileSync(path.resolve(__dirname, 'server.key')),
      cert: fs.readFileSync(path.resolve(__dirname, 'server.crt'))
    }
  }
}
```

Next

See more on the server property

srcDir

This option lets you define the source directory of your Nuxt application.

```
export default {
  srcDir: 'client/'
}
```

Project structure example with your Nuxt application in the `client` directory.

```
**-| app/
---| node_modules/
---| nuxt.config.js
---| package.json
---| client/
| assets/
| components/
| layouts/
| middleware/
| pages/
| plugins/
| static/
| store/**
```

dir

This option lets you define custom names of your Nuxt directories.

```
export default {
  dir: {
    pages: 'views' // Nuxt will look for the views/ instead of the pages/ folder
  }
}
```

Next

See more on the dir property

pageTransition

This option lets you define the default properties of the page transitions.

```
export default {
  pageTransition: 'page'
}
```

Next

See more on the transition property

Other configuration files

Besides the `nuxt.config.js` there might be other config files in your project root, such as .eslintrc, prettier.config.json or .gitignore. These are used to configure other tools such as your linter, code formatter or your git repository and detached from the `nuxt.config.js`.

.gitignore

In your .gitignore file you will need to add the following so that they are ignored and not added to version control. `node_modules` which is where all your installed modules are. The `nuxt` folder which is what gets created when running the dev or build commands. The `dist` folder is the folder that gets created when running the generate command.

```
node_modules .nuxt dist
```

What's next

Next

Check out the configuration-glossary

Go to TOC

Assets directory

The `assets` directory contains your un-compiled assets such as Stylus or Sass files, images, or fonts.

Images

Inside your `vue` templates, if you need to link to your `assets` directory use `~/assets/your_image.png` with a slash before assets.

```
<template>
  <img src="~/assets/your_image.png" />
</template>
```

Inside your `css` files, if you need to reference your `assets` directory, use `~assets/your_image.png` (without a slash)

```
background: url('~assets/banner.svg');
```

When working with dynamic images you will need to use require

```
<img :src="require(`~/assets/img/${image}.jpg`)" />
```

Next

Learn more about webpack Assets

Styles

Nuxt lets you define the CSS files/modules/libraries you want to set globally (included in every page). In the nuxt.config you can easily add your styles using the CSS Property.

```
export default {
  css: [
    // Load a Node.js module directly (here it's a Sass file)
    'bulma',
    // CSS file in the project
    '~/assets/css/main.css',
    // SCSS file in the project
    '~/assets/css/main.scss'
  ]
}
```

Sass

In case you want to use `sass` make sure that you have installed `sass` and `sass-loader` packages.

```
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev sass sass-loader@10
```

Nuxt will automatically guess the file type by its extension and use the appropriate pre-processor loader for webpack. You will still need to install the required loader if you need to use them.

Fonts

You can use local fonts by adding them to your assets folder. Once they have been added you can then access them through your css using the @font-face.

```
-| assets
----| fonts
| DMSans-Regular.ttf
| DMSans-Bold.ttf
```

```
@font-face {
  font-family: 'DM Sans';
  font-style: normal;
  font-weight: 400;
  font-display: swap;
  src: url('~assets/fonts/DMSans-Regular.ttf') format('truetype');
}

@font-face {
  font-family: 'DM Sans';
  font-style: normal;
  font-weight: 700;
  font-display: swap;
  src: url('~assets/fonts/DMSans-Bold.ttf') format('truetype');
}
```

Info

CSS files are not automatically loaded. Add them using the CSS config property.

Next

To add external fonts such as google fonts check out the Meta Tags and SEO chapter

Webpack Assets

By default, Nuxt uses webpack's vue-loader, file-loader and url-loader to serve your assets. You can also use the static directory for assets that should not run through webpack

Webpack

vue-loader automatically processes your style and template files with `css-loader` and the Vue template compiler out of the box. In this compilation process, all asset URLs such as `<img src="...">`, `background: url(...)` and CSS `@import` are resolved as module dependencies.

For example, we have this file tree:

```
-| assets/
----| image.png
-| pages/
----| index.vue
```

If you use `url('~assets/image.png')` in your CSS, it will be translated into `require('~/assets/image.png')`.

Warning

The `~/` alias won't be resolved correctly in your CSS files. You must use `~assets` (**without a slash**) in `url` CSS references, i.e. `background: url("~assets/banner.svg")`

If you reference that image in your `pages/index.vue`:

```
<template>
  <img src="~/assets/image.png" />
</template>
```

It will be compiled into:

```
createElement('img', { attrs: { src: require('~/assets/image.png') } })
```

Because `.png` is not a JavaScript file, Nuxt configures webpack to use file-loader and url-loader to handle them for you.

The benefits of these loaders are:

`file-loader` lets you designate where to copy and place the asset file, and how to name it using version hashes for better caching. In production, you will benefit from long-term caching by default!

`url-loader` allows you to conditionally inline files as base64 data URLs if they are smaller than a given threshold. This can reduce the number of HTTP requests for trivial files. If a file is larger than the threshold, it automatically falls back to file-loader.

For these two loaders, the default configuration is:

```
//
https://github.com/nuxt/nuxt.js/blob/dev/packages/webpack/src/config/base.js#L382-
L411
{
  test: /\.(png|jpe?g|gif|svg|webp|avif)$/i,
  use: [{
    loader: 'url-loader',
    options: {
      esModule: false,
      limit: 1000, // 1kB
      name: 'img/[name].[contenthash:7].[ext]'
    }
  }]
},
{
  test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/i,
  use: [{
    loader: 'url-loader',
```

```
      options: {
        esModule: false,
        limit: 1000, // 1kB
        name: 'fonts/[name].[contenthash:7].[ext]'
      }
    }]
  },
  {
    test: /\.(webm|mp4|ogv)$/i,
    use: [{
      loader: 'file-loader',
      options: {
        esModule: false,
        name: 'videos/[name].[contenthash:7].[ext]'
      }
    }]
  }
```

Which means that every file below 1 kB will be inlined as base64 data URL. Otherwise, the image/font will be copied in its corresponding folder (inside the `.nuxt` directory) with a name containing a version hash for better caching.

When launching your application with `nuxt`, your template in `pages/index.vue`:

```
<template>
  <img src="~/assets/your_image.png" />
</template>
```

Will be transformed into:

```
<img src="/_nuxt/img/your_image.0c61159.png" />
```

If you want to change the loader configurations, please use build.extend.

Aliases

By default the source directory (srcDir) and the root directory (rootDir) are the same. You can use the alias of `~` for the source directory. Instead of writing relative paths like `../assets/your_image.png` you can use `~/assets/your_image.png`.

Both will achieve the same results.

```
<template>
  <div>
    <img src="../assets/your_image.png" />
    <img src="~/assets/your_image.png" />
  </div>
</template>
```

We recommend using the `~` as an alias. `@` is still supported but will not work in all cases such as with background images in your css.

You can use the alias of `~~` or `@@` for the root directory.

Info

Tip: On Spanish keyboard you can access `~` with (`Option` + `ñ`) on Mac OS, or (`Alt gr` + `4`) on Windows

Go to TOC

Components directory

The `components` directory contains your Vue.js Components. Components are what makes up the different parts of your page and can be reused and imported into your pages, layouts and even other components.

Fetching Data

To access asynchronous data from an API in your components you can use Nuxt `fetch()`.

By checking `$fetchState.pending`, we can show a message when data is waiting to be loaded. We can also check `$fetchState.error` and show an error message if there is an error fetching the data. When using `fetch()`, we must declare the appropriate properties in `data()`. The data that comes from the fetch can then be assigned to these properties.

```
<template>
  <div>
    <p v-if="$fetchState.pending">Loading....</p>
    <p v-else-if="$fetchState.error">Error while fetching mountains</p>
    <ul v-else>
      <li v-for="(mountain, index) in mountains" :key="index">
        {{ mountain.title }}
      </li>
    </ul>
  </div>
</template>
<script>
  export default {
    data() {
      return {
        mountains: []
      }
    },
    async fetch() {
      this.mountains = await fetch(
        'https://api.nuxtjs.dev/mountains'
      ).then(res => res.json())
    }
  }
</script>
```

Next

See the chapter on fetch() for more details on how fetch works.

Components Discovery

:prose-img{src="../../../../static/img/docs/components.png"}

Starting from `v2.13`, Nuxt can auto-import the components you use. To activate this feature, set `components: true` in your configuration:

```
export default {
  components: true
}
```

Any components in the `~/components` directory can then be used throughout your pages, layouts (and other components) without needing to explicitly import them.

```
| components/
--| TheHeader.vue
--| TheFooter.vue
```

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <TheFooter />
  </div>
</template>
```

Next

Learn more about the components module in the component discovery documentation and in this announcement article.

Dynamic Imports

To dynamically import a component, also known as lazy loading a component, all you need to do is add the `Lazy` prefix in your templates.

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <LazyTheFooter />
  </div>
</template>
```

Using the lazy prefix you can also dynamically import a component when an event is triggered.

```
<template>
  <div>
    <h1>Mountains</h1>
    <LazyMountainsList v-if="show" />
    <button v-if="!show" @click="show = true">Show List</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      show: false
    }
  }
}
</script>
```

Nested Directories

If you have components in nested directories such as:

```
components/
  base/
      foo/
         CustomButton.vue
```

The component name will be based on its own path directory and filename. Therefore, the component will be:

```
<BaseFooCustomButton />
```

If we want to use it as `<CustomButton />` while keeping the directory structure, we can add the directory of `CustomButton.vue` into `nuxt.config.js`.

```
components: {
  dirs: [
    '~/components',
    '~/components/base/foo'
  ]
}
```

And now we can use `<CustomButton />` instead of `<BaseFooCustomButton />`.

```
<CustomButton />
```

Next

See the components property for other methods of controlling component name.

Info

Learn more about the components module.

Go to TOC

Content directory

Empower your Nuxt application with `@nuxt/content` module where you can write in a `content/` directory and fetch your Markdown, JSON, YAML and CSV files through a MongoDB like API, acting as a **Git-based Headless CMS**.

Nuxt Content Module

Nuxt 2.13+ | @nuxt/content 1.4.1
July 2020 - Page 1
Created by @KrutiePatel

1. WRITE

Getting started:

- Install module **npm install @nuxt/content**
- Create `content/` directory in your Nuxt project.

Module will parse `.md, .yaml, .yml, .csv, .json, .json5, .xml` and generate following default properties:

- **dir** - **extension (.md)**
- **path** - **createdAt**
- **slug** - **updatedAt**

CSV — csv rows are transformed into an array objects and assigned to body variable

XML — xml is parsed into body variable

YAML/YML — no body is generated

JSON/JSON5 — no body is generated

Markdown

1. **frontmatter**

```
---
title: Introduction
description: Content module
---
```

title and **description** get merged with default properties

2. **headings**
Adds **id** and **link** to each heading

3. **Links** - remark-external-link
Adds **target** and **rel**
Relative links are transformed to nuxt-link

1. Write continues...

4. **Footnotes** - remark-footnote
Add footnotes like, `[^1]`,
then define that footnote like,
`[^1]: This is my first footnote`

5. **Codeblocks**
Add codeblocks

6. **Syntax highlight** - Prism Js

data line | file name

```
```js{1, 3-5}[server.js]
/* ``` */
```

7. **HTML**
Add html inside markdown file

8. **Vue Components**
Add Vue component inside markdown file

✓ `<my-component>` - only kebab case
✗ `<my-component/>` - no self-closing

Define options in your frontmatter, and pass it as props in vue components

Use `<template #named-slot> <template/>` within vue components

9. **Global components**
Put components in `components/global/` and access them directly inside markdown file

10. **TOC**
- **id of titles** create links
- **h2 & h3** are used to create toc

2. FETCH

Fetch content on:
- Client-side using `this.$content`
- Server-side using `context.$content`

```
content(path, options).fetch()
```

```
Example:
$content('articles', params.slug).fetch()
== /articles/${params.slug}
```

Chainable methods:

1. **only(keys)**
Select a subset of fields

2. **without(keys)**
Remove a subset of fields

3. **where(query)**
Filter results by query

4. **sortBy(key, direction)**
Sort results by key

5. **limit(n)**
Limit number of results

6. **skip(n)**
Skip results

7. **search(field, value)**
Perform a full-text search on a field
Default fields:
`['title', 'description', 'slug', 'text']`

Nuxt Content Module

Nuxt 2.13+ | @nuxt/content 1.4
July 2020 - Page 2
Created by @KrutiePatel

2. Fetch continues...

8. **surround(slug, options)**
Get prev and next results arround a specific slug
`search, limit` and `skip` are ineffective when using this method

9. **fetch()**
Ends chain sequence and collects data

path is file → fetch returns → Promise<Object>
path is dir → fetch returns → Promise<Array>

Example:

```
const articles =
await this.$content('articles')
.only(['title', 'date', 'authors'])
.sortBy('date', 'asc')
.limit(5)
.skip(10)
.where({tags: 'testing',
        isArchived: false,
        date: { $gt: new Date(2020) },
        rating: { $gte: 3 }
      })
.search('welcome')
.fetch()
```

Example with options:

Fetch files from subdirectories | Returns markdown content in a **text** variable

```
const options = {deep:true, text:true}
await this.$content('articles', params.slug, options)
```

3. DISPLAY

Use **<nuxt-content>** component directly in template area to display the page body. **No import required.**

```
<nuxt-content :document="page" />
```

<nuxt-content> accepts page object using :document prop.

Live editing (>= v1.4.0): In development, double-click on the <nuxt-content> component to live edit the content directly in the browser!

Custom style: <nuxt-content> component adds .nuxt-content class. .nuxt-content class can be used to add custom styles for the content.

```
.nuxt-content h1 {
// add custom style for h1 here
}
```

Example:

```
<template>
  <article>
    <h1>{{ page.title }}</h1>
    <nuxt-content :document="page" />
  </article>
</template>
<script>
export default {
  async asyncData ({ $content }) {
    const page = await
$content('home').fetch()

    return {
      page
    }
  }
}
</script>
```

Default configuration

Configure **@nuxt/content** with the **content** property in nuxt.config.js.

1. Customise api url using **apiPrefix.** By default, $content api will be served on http://localhost:3000/**_content**/articles
2. Provide different directory for writing content.
3. Customise searchable fields.
4. Provide custom PrismJS theme.

Legends

```
// nuxt.config.js
export default {
  content: {
    apiPrefix: '_content',
    dir: 'content',
    fullTextSearchFields:
      ['title','description', 'slug','text'],
    nestedProperties: ['categories.slug'],
    markdown: {
      remarkPlugins: [
        'remark-squeeze-paragraphs',
        'remark-slug',
        'remark-autolink-headings',
        'remark-external-links',
        'remark-footnotes'
      ],
      rehypePlugins: [
        'rehype-minify-whitespace',
        'rehype-sort-attribute-values',
        'rehype-sort-attributes',
        'rehype-raw'
      ],
    prism: {
        theme: 'prismjs/themes/prism.css'
      }
    },
    yaml: {},
    csv: {},
    xml: {}
  }
}
```

Learn more about custom configuration:
https://content.nuxtjs.org/configuration
```

Hot reload in development

The content module is blazing fast when it comes to hot reloading in development due to not having to go through webpack when you make changes to your markdown files. You can also listen to the `content:update` event and create a plugin so that every time you update a file in your content directory it will dispatch a fetchCategories method for example.

Next

See the content module docs for more details

Displaying content

You can use `<nuxt-content>` component directly in your template to display the page body.

```
<template>
  <article>
    <nuxt-content :document="article" />
  </article>
</template>
```

Next

See the content module docs for more details

Styling your content

Depending on what you're using to design your app, you may need to write some style to properly display the markdown.

`<nuxt-content>` component will automatically add a `.nuxt-content` class, you can use it to customize your styles.

```
<style>
  .nuxt-content h2 {
    font-weight: bold;
    font-size: 28px;
  }
  .nuxt-content p {
    margin-bottom: 20px;
  }
</style>
```

Next

See the content module docs for more details

Handles Markdown, CSV, YAML, JSON(5)

This module converts your .md files into a JSON AST tree structure, stored in a body variable. You can also add a YAML front matter block to your markdown files or a .yaml file which will be injected into the document. You can also add a json/json5 file which can also be injected into the document. And you can use a .csv file where rows will be assigned to the body variable.

```
---
title: My first Blog Post
description: Learning how to use @nuxt/content to create a blog
---
```

Next

See the content module docs for more details

Vue components in Markdown

You can use Vue components directly in your markdown files. You will however need to use your components as kebab case and cannot use self-closing tags.

```
<template>
  <div class="p-4 mb-4 text-white bg-blue-500">
    <p><slot name="info-box">default</slot></p>
  </div>
</template>
```

```
<info-box>
  <template #info-box>
    This is a vue component inside markdown using slots
  </template>
</info-box>
```

Next

See the content module docs for more details

Fully Searchable API

You can use `$content()` to list, filter and search your content easily.

```
<script>
  export default {
    async asyncData({ $content, params }) {
      const articles = await $content('articles', params.slug)
        .only(['title', 'description', 'img', 'slug', 'author'])
        .sortBy('createdAt', 'asc')
        .fetch()

      return {
        articles
      }
    }
  }
</script>
```

Next

See the content module docs for more details

Previous and Next articles

The content module includes a `.surround(slug)` so that you get previous and next articles easily.

```
async asyncData({ $content, params }) {
    const article = await $content('articles', params.slug).fetch()

    const [prev, next] = await $content('articles')
      .only(['title', 'slug'])
      .sortBy('createdAt', 'asc')
      .surround(params.slug)
      .fetch()

    return {
      article,
      prev,
      next
    }
  },
```

```
<prev-next :prev="prev" :next="next" />
```

Next

See the content module docs for more details

Full-text search

The content module comes with a full text search so you can easily search across your markdown files without having to install anything.

```
<script>
  export default {
    data() {
      return {
        searchQuery: '',
        articles: []
      }
    },
    watch: {
      async searchQuery(searchQuery) {
        if (!searchQuery) {
          this.articles = []
          return
        }
        this.articles = await this.$content('articles')
          .limit(6)
          .search(searchQuery)
          .fetch()
      }
    }
  }
</script>
```

Next

See the content module docs for more details

Syntax highlighting

This module automatically wraps codeblocks and applies Prism classes. You can also add a different Prism theme or disable it altogether.

```
yarn add prism-themes
```

```
npm install prism-themes
```

```
content: {
  markdown: {
    prism: {
      theme: 'prism-themes/themes/prism-material-oceanic.css'
    }
  }
}
```

Next

See the content module docs for more details

Extend Markdown Parsing

Originally markdown does not support highlighting lines inside codeblock nor filenames. The content module allows it with its own custom syntax. Line numbers are added to the `pre` tag in data-line attributes and the filename will be converted to a `span` with a `filename` class, so you can style it.

Next

See the content module docs for more details

Table of contents generation

A toc(Table of Contents) array property will be injected into your document, listing all the headings with their titles and ids, so you can link to them.

```
<nav>
  <ul>
    <li v-for="link of article.toc" :key="link.id">
      <NuxtLink :to="`#${link.id}`">{{ link.text }}</NuxtLink>
    </li>
  </ul>
</nav>
```

Next

See the content module docs for more details

Powerful query builder API (MongoDB-like)

The content module comes with a powerful query builder API similar to MongoDB which allows you to easily see the JSON of each directory at `http://localhost:3000/_content/`. The endpoint is accessible on GET and POST request, so you can use query params.

`http://localhost:3000/_content/articles?only=title&only=description&limit=10`

Next

See the content module docs for more details

Extend with hooks

You can use hooks to extend the module so you can add data to a document before it is stored.

Next

See the content module docs for more details

Integration with @nuxtjs/feed

In the case of articles, the content can be used to generate news feeds using @nuxtjs/feed module.

Next

See the content module docs for more details

Support static site generation

The content module works with static site generation using the `nuxt generate`. All routes will be automatically generated thanks to the nuxt crawler feature.

Warning

If using Nuxt <2.13 and you need to specify the dynamic routes you can do so using the generate property and using @nuxt/content programmatically.

Next

See the content module docs for more details on programmatic usage

What's next

Next

Check out our tutorial on How to Create a Blog with Nuxt Content

Next

Check out the content module docs for more advanced usage and examples

Go to TOC

Dist directory

The `dist` folder, short for *distribution* folder, is dynamically generated when using the `nuxt generate` commands and includes the generated production ready HTML files and assets that are necessary to deploy and run your statically generated Nuxt application.

Deploying

This is the folder you need to **upload for static hosting** as it contains your generated production ready HTML files and assets

Warning

The `dist` directory should not be committed to your version control system and should be ignored through your `.gitignore` as it will be generated automatically every time you run `nuxt generate`.

The dir property

The dist folder is named dist by default but can be configured in your nuxt.config file.

```
generate: {
  dir: 'my-site'
}
```

Warning

If you do change your dist folder then you will need to add that to your version control so that git will ignore it.

The subFolders Property

Nuxt puts all your generated pages inside a folder by default, however you can change this if you want by modifying the nuxt.config and changing the subFolders to be false.

```
generate: {
  subFolders: false
}
```

The fallback Property

When deploying your site you will need to make sure the fallback HTML path is set correctly. It should be set as the error page so that unknown routes are rendered via Nuxt. If it is unset Nuxt will use the default value which is 200.html.

When running a single page application it makes more sense to use 200.html as it is the only file necessary as no other routes are generated.

When working with statically generated pages it is recommended to use a 404.html for error pages.

Warning

Depending on where you are hosting your site, you may have to use 200.html or 404.html. Please check with your hosting provider. Netlify, for example uses 404.html.

```
export default {
  generate: {
    fallback: '404.html'
  }
}
```

The excludes property

You can exclude pages from being generated by using the generate excludes property. Instead of being generated as a static page it will fallback to be a single page application page and will only be rendered on the client side.

```
generate: {
  exclude: [/admin/]
}
```

Info

You can also use a regex expression here to exclude pages starting or ending with a particular word

Go to TOC

Layouts directory

Layouts are a great help when you want to change the look and feel of your Nuxt app. Whether you want to include a sidebar or have distinct layouts for mobile and desktop.

Warning

This directory cannot be renamed without extra configuration.

Default Layout

You can extend the main layout by adding a `layouts/default.vue` file. It will be used for all pages that don't have a layout specified. Make sure to add the `<Nuxt>` component when creating a layout to actually include the page component.

All you need in your layout is three lines of code which will render the page component.

```
<template>
  <Nuxt />
</template>
```

You can add more components here such as Navigation, Header, Footer etc.

```
<template>
  <div>
    <TheHeader />
    <Nuxt />
    <TheFooter />
  </div>
</template>
```

Info

If you have components set to true then there is no need to add any import statements for your components.

Custom Layout

Every file (*top-level*) in the `layouts` directory will create a custom layout accessible with the `layout` property in the page components.

Let's say we want to create a blog layout and save it to `layouts/blog.vue`:

```
<template>
  <div>
    <div>My blog navigation bar here</div>
    <Nuxt />
  </div>
</template>
```

Then you have to tell the pages to use your custom layout

```
<script>
export default {
  layout: 'blog',
  // OR
  layout (context) {
    return 'blog'
  }
}
</script>
```

Error Page

The error page is a *page component* which is always displayed when an error occurs (that is not thrown on the server-side).

Warning

Though this file is placed in the `layouts` folder, it should be treated as a page.

As mentioned above, this layout is special and you should not include `<Nuxt>` inside its template. You must see this layout as a component displayed when an error occurs (`404`, `500`, etc.). Similar to other page components, you can set a custom layout for the error page as well in the usual way.

You can customize the error page by adding a `layouts/error.vue` file:

```
<template>
  <div class="container">
    <h1 v-if="error.statusCode === 404">Page not found</h1>
    <h1 v-else>An error occurred</h1>
    <NuxtLink to="/">Home page</NuxtLink>
  </div>
</template>

<script>
export default {
  props: ['error'],
  layout: 'blog' // you can set a custom layout for the error page
}
</script>
```

Info

The default error page source code is available on GitHub.

Go to TOC

Middleware directory

The `middleware` directory contains your application middleware. Middleware lets you define custom functions that can be run before rendering either a page or a group of pages (layout).

Shared middleware should be placed in the `middleware/` directory. The filename will be the name of the middleware (`middleware/auth.js` will be the `auth` middleware). You can also define page-specific middleware by using a function directly, see anonymous middleware.

A middleware receives the context as the first argument.

```
export default function (context) {
  // Add the userAgent property to the context
  context.userAgent = process.server
    ? context.req.headers['user-agent']
    : navigator.userAgent
}
```

In universal mode, middlewares will be called once on server-side (on the first request to the Nuxt app, e.g. when directly accessing the app or refreshing the page) and on the client-side when navigating to further routes. With `ssr: false`, middlewares will be called on the client-side in both situations.

The middleware will be executed in series in this order:

1. `nuxt.config.js` (in the order within the file)
2. Matched layouts
3. Matched pages

Router Middleware

A middleware can be asynchronous. To do this return a `Promise` or use async/await.

```
import http from 'http'

export default function ({ route }) {
  return http.post('http://my-stats-api.com', {
    url: route.fullPath
  })
}
```

Then, in your `nuxt.config.js`, use the `router.middleware` key.

```
export default {
  router: {
    middleware: 'stats'
  }
}
```

Now the `stats` middleware will be called for every route change.

You can add your middleware (even multiple) to a specific layout or page as well.

```
export default {
  middleware: ['auth', 'stats']
}
```

Named middleware

You can create named middleware by creating a file inside the `middleware/` directory, the file name will be the middleware name.

```
export default function ({ store, redirect }) {
  // If the user is not authenticated
  if (!store.state.authenticated) {
    return redirect('/login')
  }
}
```

```
<template>
  <h1>Secret page</h1>
</template>

<script>
  export default {
    middleware: 'authenticated'
  }
</script>
```

Anonymous middleware

If you need to use a middleware only for a specific page, you can directly use a function for it (or an array of functions):

```
<template>
  <h1>Secret page</h1>
</template>

<script>
  export default {
    middleware({ store, redirect }) {
      // If the user is not authenticated
      if (!store.state.authenticated) {
        return redirect('/login')
      }
    }
  }
</script>
```

Go to TOC

Modules directory

Nuxt provides a higher-order module system that makes it possible to extend the core. Modules are functions that are called sequentially when booting Nuxt.

Exploring Nuxt Modules

Discover our list of modules to supercharge your Nuxt project, created by the Nuxt team and community.

- 165+ Modules
- 105+ Maintainers

Next

Check out modules.nuxtjs.org

Nuxt Modules

Nuxt v2.14
September 2020

modules are Nuxt extensions that **extend Nuxt's core functionalities**

property type: **array**

modules

Called once, in sequence, while booting Nuxt app
i.e. npm run dev

```
// nuxt.config.js
export default {
  modules: [
    // Using package name
   '@nuxtjs/axios',

    // Relative to project srcDir
    '~/modules/simple',

    // Providing options
    ['~/modules/simple', { token: '123' }]

    // Inline definition
    function () {}
  ]
}
```

array item can be string, array with options or function

Write your own module

Combine multiple options sources

Top-level options

```
// nuxt.config.js
export default {
 modules: [
  [ '@nuxtjs/axios', { anotherOption: true }] ],
 //axios module is aware of this by using`this.options.axios`
 axios: { option1, option2 }
}
```

Access Nuxt options, including module's options
i.e. add a CSS library or emit assets

`this.options`

```
// modules/module.js
export default function (moduleOptions) {
 // `options` will contain option1,option2 & anotherOption
 const options = Object.assign({},
                               this.options.axios,
                               moduleOptions
                              )
}
```

Refers to the current Nuxt instance, allowing us to register hooks on certain life-cycle events
i.e. provide plugins or register custom webpack loaders

`this.nuxt`

```
// modules/module.js
export default function (moduleOptions) {
  this.nuxt.hook('ready', async nuxt => {})
  this.nuxt.hook('error', async error => {})
  this.nuxt.hook('close', async nuxt => {})
  this.nuxt.hook('listen', async(server, {host, port})) => {})
}
```

`this` refers to the `moduleContainer` instance

create async modules that return a promise or call a callback

async module

```
// modules/mymodule.js
// Use async/await
export default async function asyncModule() {
 const pages = await ...
}
// Return a promise
export default function asyncModule() {
 return fetch().then(response => {...})
}
```

Use `modules`

If module:
- provides serverMiddleware
- registers Node.js runtime hook
- affects vue-renderer behaviour
- or anything that's outside of webpack scope

```
// nuxt.config.js
export default {
 modules: ['@nuxtjs/sitemap']
}
```

Use `buildModules`

If module is imported during dev and build time.

simply, add dependency in devDependencies and use `buildModules`:

```
// nuxt.config.js
export default {
 buildModules: ['@nuxtjs/eslint-module']
}
```

Benefits:
- ✅ Decrease `node_module` size
- ✅ Make production startup faster

While developing production-grade applications with Nuxt you might find that the framework's core functionality is not enough. Nuxt can be extended with configuration options and plugins, but maintaining these customizations across multiple projects is tedious, repetitive and time-consuming. On the other hand, supporting every project's needs out of the box would make Nuxt very complex and hard to use.

This is one of the reasons why Nuxt provides a higher-order module system that makes it possible to extend the core. Modules are functions that are called sequentially when booting Nuxt. The framework waits for each module to finish before continuing. In this way, modules can customize almost any aspect of your project. Thanks to Nuxt's modular design (based on webpack's Tapable), modules can easily register hooks for certain entry points like the builder initialization. Modules can also override templates, configure webpack loaders, add CSS libraries, and perform many other useful tasks.

Best of all, Nuxt modules can be incorporated into npm packages. This makes it possible to reuse across projects and to share with the community, helping create an ecosystem of high-quality add-ons.

The modules Property

Modules are Nuxt extensions which can extend the framework's core functionality and add endless integrations. Once you have installed the modules you can then add them to your nuxt.config.js file under the modules property.

```
export default {
  modules: [
    // Using package name
    '@nuxtjs/axios',

    // Relative to your project srcDir
    '~/modules/awesome.js',

    // Providing options
    ['@nuxtjs/google-analytics', { ua: 'X1234567' }],

    // Inline definition
    function () {}
  ]
}
```

Info

Module developers usually provide additionally needed steps and details for usage.

Nuxt tries to resolve each item in the modules array using node require path (in the `node_modules`) and then will resolve from the project `srcDir` if `@` alias is used.

Warning

Modules are executed sequentially so the order is important.

Modules should export a function to enhance build/runtime and optionally return a promise until their job is finished. Note that they are imported at runtime so they should be already transpiled if using modern ES6 features.

Write your own Module

Modules are functions. They can be packaged as npm modules or directly included in your project source code.

```
export default {
  exampleMsg: 'hello',
  modules: [
    // Simple usage
    '~/modules/example',
    // Passing options directly
    ['~/modules/example', { token: '123' }]
  ]
}
```

```
export default function ExampleModule(moduleOptions) {
  console.log(moduleOptions.token) // '123'
  console.log(this.options.exampleMsg) // 'hello'

  this.nuxt.hook('ready', async nuxt => {
    console.log('Nuxt is ready')
  })
}

// REQUIRED if publishing the module as npm package
module.exports.meta = require('./package.json')
```

1) ModuleOptions

`moduleOptions`: This is the object passed using the `modules` array by the user. We can use it to customize its behavior.

Top level options

Sometimes it is more convenient if we can use top level options while registering modules in `nuxt.config.js`. This allows us to combine multiple option sources.

```
export default {
  modules: [['@nuxtjs/axios', { anotherOption: true }]],

  // axios module is aware of this by using `this.options.axios`
  axios: {
    option1,
    option2
  }
}
```

2) this.options

`this.options`: You can directly access the Nuxt options using this reference. This is the content of the user's `nuxt.config.js` with all default options assigned to it. It can be used for shared options between modules.

```
export default function (moduleOptions) {
  // `options` will contain option1, option2 and anotherOption
  const options = Object.assign({}, this.options.axios, moduleOptions)

  // ...
}
```

Add a CSS Library

If your module will provide a CSS library, make sure to perform a check if the user already included the library to avoid duplicates, and add an option to disable the CSS library in the module.

```
export default function (moduleOptions) {
  if (moduleOptions.fontAwesome !== false) {
    // Add Font Awesome
    this.options.css.push('font-awesome/css/font-awesome.css')
  }
}
```

Emit assets

We can register webpack plugins to emit assets during build.

```
export default function (moduleOptions) {
  const info = 'Built by awesome module - 1.3 alpha on ' + Date.now()

  this.options.build.plugins.push({
    apply(compiler) {
      compiler.plugin('emit', (compilation, cb) => {
        // This will generate `.nuxt/dist/info.txt' with contents of info
variable.
        // Source can be buffer too
        compilation.assets['info.txt'] = {
          source: () => info,
          size: () => info.length
        }

        cb()
      })
    }
  })
}
```

3) this.nuxt

`this.nuxt`: This is a reference to the current Nuxt instance. We can register hooks on certain life cycle events.

- **Ready** : Nuxt is ready to work (ModuleContainer and Renderer ready).

```
nuxt.hook('ready', async nuxt => {
  // Your custom code here
})
```

- **Error**: An unhandled error when calling hooks.

```
nuxt.hook('error', async error => {
  // Your custom code here
})
```

- **Close**: Nuxt instance is gracefully closing.

```
nuxt.hook('close', async nuxt => {
  // Your custom code here
})
```

- **Listen**: Nuxt internal server starts listening. (Using nuxt start or nuxt dev)

```
nuxt.hook('listen', async (server, { host, port }) => {
  // Your custom code here
})
```

`this`: Context of modules. All modules will be called within context of the ModuleContainer instance.

Please look into the ModuleContainer class docs for available methods.

Run Tasks on Specific hooks

Your module may need to do things only on specific conditions and not just during Nuxt initialization. We can use the powerful Nuxt hooks to do tasks on specific events (based on Hookable). Nuxt will wait for your function if it returns a Promise or is defined as `async`.

Here are some basic examples:

```
export default function myModule() {
  this.nuxt.hook('modules:done', moduleContainer => {
    // This will be called when all modules finished loading
  })

  this.nuxt.hook('render:before', renderer => {
    // Called after the renderer was created
  })

  this.nuxt.hook('build:compile', async ({ name, compiler }) => {
    // Called before the compiler (default: webpack) starts
  })

  this.nuxt.hook('generate:before', async generator => {
    // This will be called before Nuxt generates your pages
  })
}
```

Provide plugins

It is common that modules provide one or more plugins when added. For example bootstrap-vue module would require to register itself into Vue. In such situations we can use the `this.addPlugin` helper.

```
import Vue from 'vue'
import BootstrapVue from 'bootstrap-vue/dist/bootstrap-vue.esm'

Vue.use(BootstrapVue)
```

```
import path from 'path'

export default function nuxtBootstrapVue(moduleOptions) {
  // Register `plugin.js` template
  this.addPlugin(path.resolve(__dirname, 'plugin.js'))
}
```

Note: Any plugins injected by modules are added to the *beginning* of the plugins list. Your options are to:

- Manually add your plugin to the end of the list of plugins (`this.nuxt.options.plugins.push(...`)
- Reverse the order of the modules if it depends on another

Template plugins

Registered templates and plugins can leverage lodash templates to conditionally change registered plugins output.

```
// Set Google Analytics UA
ga('create', '<%= options.ua %>', 'auto')

<% if (options.debug) { %>
// Dev only code
<% } %>
```

```
import path from 'path'

export default function nuxtGoogleAnalytics(moduleOptions) {
  // Register `plugin.js` template
  this.addPlugin({
    src: path.resolve(__dirname, 'plugin.js'),
    options: {
      // Nuxt will replace `options.ua` with `123` when copying plugin to project
      ua: 123,

      // conditional parts with dev will be stripped from plugin code on
production builds
      debug: this.options.dev
    }
  })
}
```

Register custom webpack loaders

We can do the same as `build.extend` in `nuxt.config.js` using `this.extendBuild`.

```
export default function (moduleOptions) {
    this.extendBuild((config, { isClient, isServer }) => {
      // `.foo` Loader
      config.module.rules.push({
        test: /\.foo$/,
        use: [...]
      })

      // Customize existing loaders
      // Refer to source code for Nuxt internals:
      //
https://github.com/nuxt/nuxt.js/blob/dev/packages/webpack/src/config/base.js
      const barLoader = config.module.rules.find(rule => rule.loader === 'bar-
```

```
loader')
  })
}
```

Async Modules

Not all modules will do everything synchronous. For example you may want to develop a module which needs fetching some API or doing asynchronous Operation. For this, Nuxt supports async modules which can return a Promise or call a callback.

Use async/await

```
import fse from 'fs-extra'

export default async function asyncModule() {
  // You can do async work here using `async`/`await`
  const pages = await fse.readJson('./pages.json')
}
```

Return a Promise

```
export default function asyncModule($http) {
  return $http
    .get('https://jsonplaceholder.typicode.com/users')
    .then(res => res.data.map(user => '/users/' + user.username))
    .then(routes => {
      // Do something by extending Nuxt routes
    })
}
```

Info

There are way more hooks and possibilities for modules. Please read the Nuxt Internals to find out more about the nuxt-internal API.

Publishing your module

`module.exports.meta`: This line is required if you are publishing the module as an npm package. Nuxt internally uses meta to work better with your package.

```
module.exports.meta = require('./package.json')
```

buildModules

Some modules are only imported during development and build time. Using `buildModules` helps to make production startup faster and also significantly decrease the size of your `node_modules` for production deployments. Please refer to the docs for each module to see if it is recommended to use `modules` or `buildModules`.

The usage difference is:

- Instead of adding to `modules` inside `nuxt.config.js`, use `buildModules`

```
export default {
  buildModules: ['@nuxtjs/eslint-module']
}
```

- Instead of adding to `dependencies` inside `package.json`, use `devDependencies`

```
yarn add --dev @nuxtjs/eslint-module
```

```
npm install --save-dev @nuxtjs/eslint-module
```

Info

If you are a module author, It is highly recommended to suggest to users to install your package as a `dev-Dependency` and use `buildModules` instead of `modules` for `nuxt.config.js`.

Your module is a `buildModule` unless:

- It is providing a serverMiddleware
- It has to register a Node.js runtime hook (Like sentry)
- It is affecting vue-renderer behavior or using a hook from `server:` or `vue-renderer:` namespace
- Anything else that is outside of webpack scope (Hint: plugins and templates are compiled and are in webpack scope)

Warning

If you are going to offer using `buildModules` please mention that this feature is only available since Nuxt v2.9. Older users should upgrade Nuxt or use the `modules` section.

Go to TOC

Pages directory

The `pages` directory contains your Application Views and Routes. Nuxt reads all the `.vue` files inside this directory and automatically creates the router configuration for you.

Info

You can also create routes with .js files and .ts files

Every Page component is a Vue component but Nuxt adds special attributes and functions to make the development of your universal application as easy as possible.

```
<template>
  <h1 class="red">Hello {{ name }}!</h1>
</template>

<script>
  export default {
    // page properties go here
  }
</script>

<style>
  .red {
    color: red;
  }
</style>
```

Dynamic Pages

Dynamic pages can be created when you don't know the name of the page due to it coming from an API or you don't want to have to create the same page over and over again. To create a dynamic page you need to add an underscore before the .vue file name or before the name of the directory, if you want the directory to be dynamic. You can name the file or directory anything you want but you must prefix it with an underscore.

If you've defined a file named `_slug.vue` in your pages folder, you can access the value using the context with params.slug

```
<template>
  <h1>{{ this.slug }}</h1>
</template>

<script>
  export default {
    async asyncData({ params }) {
      const slug = params.slug // When calling /abc the slug will be "abc"
      return { slug }
    }
  }
</script>
```

If you've defined a file named `_slug.vue` inside a folder called `_book` you can access the value using the context with `params.slug` and `params.book`

```
<template>
  <h1>{{ this.book }} / {{ this.slug }}</h1>
</template>

<script>
  export default {
    async asyncData({ params }) {
      const book = params.book
      const slug = params.slug
      return { book, slug }
    }
  }
</script>
```

Properties

asyncData

asyncData is called every time before loading the component. It can be asynchronous and receives the context as an argument. The returned object will be merged with your data object.

```
export default {
  asyncData(context) {
    return { name: 'World' }
  }
}
```

Next

See more on how asyncData works in our Data Fetching chapter

fetch

Every time you need to get asynchronous data you can use fetch. Fetch is called on server-side when rendering the route, and on client-side when navigating.

```
<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    async fetch() {
      this.posts = await fetch('https://api.nuxtjs.dev/posts').then(res =>
        res.json()
      )
    }
  }
</script>
```

Next

See more on how fetch works in our Data Fetching chapter

head

Set specific tags for the current page. Nuxt uses `vue-meta` to update the document head and meta attributes of your application.

```
export default {
  head() {
    // Set Meta Tags for this Page
  }
}
```

Next

See more in our Meta Tags and SEO chapter

layout

Specify a layout defined in the layouts directory.

```
export default {
  layout: 'blog'
}
```

Next

See more on layouts in our Views chapter.

loading

If set to false, prevents a page from automatically calling `this.$nuxt.$loading.finish()` as you enter it and `this.$nuxt.$loading.start()` as you leave it, allowing you to manually control the behavior, as this example shows.

```
export default {
  loading: false
}
```

Info

Only applies if loading is also set in nuxt.config.js.

Next

See more in our Loading chapter.

transition

Defines a specific transition for the page.

```
export default {
  transition: 'fade'
}
```

Next

See more on transitions in our Transitions chapter

scrollToTop

The `scrollToTop` property lets you tell Nuxt to scroll to the top before rendering the page. By default, Nuxt scrolls to the top when you go to another page, but with child routes, Nuxt keeps the scroll position. If you want to tell Nuxt to scroll to the top when rendering your child route, set `scrollToTop` to `true`

```
export default {
  scrollToTop: true
}
```

Conversely, you can manually set `scrollToTop` to `false` on parent routes as well.

If you want to overwrite the default scroll behavior of Nuxt, take a look at the scrollBehavior option.

middleware

Defines middleware for this page. The middleware will be called before rendering the page.

```
export default {
  middleware: 'auth'
}
```

Next

See more on middleware in our Middleware chapter

The watchQuery Property

Use the `watchQuery` key to set up a watcher for query strings. If the defined strings change, all component methods (asyncData, fetch(context), validate, layout, ...) will be called. Watching is disabled by default to improve performance.

```
export default {
  watchQuery: ['page']
}
```

Warning

Warning: The new `fetch` hook introduced in 2.12 is not affected by `watchQuery`. For more information see listening to query string changes.

```
export default {
  watchQuery: true
}
```

You can also use the function `watchQuery(newQuery, oldQuery)` to have more refined watchers.

```
export default {
  watchQuery(newQuery, oldQuery) {
    // Only execute component methods if the old query string contained `bar`
    // and the new query string contains `foo`
    return newQuery.foo && oldQuery.bar
  }
}
```

Next

See more on the watch query property in our Data Fetching chapter

key

Same as the `key` property that can be used on Vue components in templates as a hint for the virtual DOM, this property allows the key value to be defined from the page itself (rather than the parent component).

By default in Nuxt, this value will be the `$route.path`, meaning that navigating to a different route will ensure a clean page component is created. Logically equivalent to:

```
<router-view :key="$route.path" />
```

The property can be a `String` or a `Function` which takes the route as the first argument.

Ignoring pages

If you want to ignore pages so that they are not included in the generated `router.js` file then you can ignore them by prefixing them with a `-`.

For example, `pages/-about.vue` will be ignored.

Next

Checkout the ignore option to learn more about it.

Configuration

You can rename the `pages/` directory to something different by setting `dir.pages` option:

```
export default {
  dir: {
    // Rename `pages` directory to `routes`
    pages: 'routes'
  }
}
```

Next

Checkout the dir option to learn more about it.

Go to TOC

The alias property

Nuxt allows you to use aliases to access custom directories within your JavaScript and CSS

- Type: `Object`
- Default:

```
{
  '~~': `<rootDir>`,
  '@@': `<rootDir>`,
  '~': `<srcDir>`,
  '@': `<srcDir>`,
  'assets': `<srcDir>/assets`, // (unless you have set a custom `dir.assets`)
  'static': `<srcDir>/static`, // (unless you have set a custom `dir.static`)
}
```

This option lets you define aliases to directories within your project (in addition to the ones above). These aliases can be used within your JavaScript and CSS.

```
import { resolve } from 'path'
export default {
  alias: {
    'images': resolve(__dirname, './assets/images'),
    'style': resolve(__dirname, './assets/style'),
    'data': resolve(__dirname, './assets/other/data')
  }
}
```

```
<template>
  <img src="~images/main-bg.jpg">
</template>

<script>
import data from 'data/test.json'

// etc.
</script>

<style>
@import '~style/variables.scss';
@import '~style/utils.scss';
@import '~style/base.scss';

body {
  background-image: url('~images/main-bg.jpg');
}
</style>
```

Warning

Within a Webpack context (image sources, CSS - but *not* JavaScript) you must prefix your alias with `~` (as in the example above).

Info

If you are using TypeScript and want to use the alias you define within your TypeScript files, you will need to add the aliases to your `paths` object within `tsconfig.json`.

Go to TOC

The extendPlugins Property

The extendPlugins property lets you customize Nuxt plugins. (options.plugins).

- Type: `Function`
- Default: `undefined`

You may want to extend plugins or change plugins order created by Nuxt. This function accepts an array of plugin objects and should return array of plugin objects.

Example of changing plugins order:

```
export default {
  extendPlugins(plugins) {
    const pluginIndex = plugins.findIndex(
      plugin => (typeof plugin === 'string' ? plugin : plugin.src) ===
'~/plugins/shouldBeFirst.js'
    )
    const shouldBeFirstPlugin = plugins[pluginIndex]

    plugins.splice(pluginIndex, 1)
    plugins.unshift(shouldBeFirstPlugin)

    return plugins
  }
}
```

Go to TOC

The generate property

Configure the generation of your universal web application to a static web application.

- Type: `Object`

When calling `nuxt.generate()`, Nuxt will use the configuration defined in the `generate` property.

```
export default {
  generate: {
    ...
  }
}
```

cache

> Introduced in v2.14.0

- Type: `Object` or `false`

This option is used by `nuxt generate` with static target to avoid re-building when no tracked file has been changed.

Defaults:

```
{
  ignore: [
    '.nuxt', // buildDir
    'static', // dir.static
    'dist', // generate.dir
    'node_modules',
    '.**/*',
    '.*',
    'README.md'
  ]
}
```

If you want to avoid re-building when changing a configuration file, just add it to the list by providing the `cache.ignore` option:

```
export default {
  generate: {
    cache: {
      ignore: ['renovate.json'] // ignore changes applied on this file
    }
  }
}
```

concurrency

- Type: `Number`
- Default: `500`

The generation of routes are concurrent, `generate.concurrency` specifies the amount of routes that run in one thread.

crawler

- Type: `boolean`
- Default: true

As of Nuxt >= v2.13 Nuxt comes with a crawler installed that will crawl your relative links and generate your dynamic links based on these links. If you want to disable this feature you can set the value to `false`

```
export default {
  generate: {
    crawler: false
  }
}
```

dir

- Type: `String`
- Default: `'dist'`

Directory name created when building web applications using the `nuxt generate` command.

devtools

- Type: `boolean`
- Default: `false`

Configure whether to allow vue-devtools inspection.

If you already activated through nuxt.config.js or otherwise, devtools enable regardless of the flag.

exclude

- Type: `Array`
  - Items: `String` or `RegExp`

It accepts an array of string or regular expressions and will prevent generation of routes matching them. The routes will still be accessible when `generate.fallback` is used.

Taking this examples of structure:

```
-| pages/
---| index.vue
---| admin/
-----| about.vue
-----| index.vue
```

By default, running `nuxt generate` a file will be created for each route.

```
-| dist/
---| index.html
---| admin/
-----| about.html
-----| index.html
```

When adding a regular expression which matches all routes with "ignore", it will prevent the generation of these routes.

```
export default {
  generate: {
    exclude: [
      /^\/admin/ // path starts with /admin
    ]
  }
}
```

```
-| dist/
---| index.html
```

You can also exclude a specific route by giving a string:

```
export default {
  generate: {
    exclude: ['/my-secret-page']
  }
}
```

fallback

- Type: `String` or `Boolean`
- Default: `200.html`

```
export default {
  generate: {
    fallback: '404.html'
  }
}
```

The path to the fallback HTML file. It should be set as the error page, so that also unknown routes are rendered via Nuxt. If unset or set to a falsy value, the name of the fallback HTML file will be `200.html`. If set to `true`, the filename will be `404.html`. If you provide a string as a value, it will be used instead.

```
fallback: false;
```

If working with statically generated pages then it is recommended to use a `404.html` for error pages and for those covered by excludes (the files that you do not want generated as static pages).

```
fallback: true
```

However, Nuxt allows you to configure any page you like so if you don't want to use the `200.html` or `404.html` you can add a string and then you just have to make sure you redirect to that page instead. This is of course not necessary and is best to redirect to `200.html` / `404.html`.

```
fallback: 'fallbackPage.html'
```

Note: Multiple services (e.g. Netlify) detect a `404.html` automatically. If you configure your web server on your own, please consult its documentation to find out how to set up an error page (and set it to the `404.html` file)

interval

- Type: `Number`
- Default: `0`

Interval in milliseconds between two render cycles to avoid flooding a potential API with calls from the web application.

minify

- **Deprecated!**
- Use build.html.minify instead

routes

- Type: `Array`

Info

As of Nuxt v2.13 there is a crawler installed that will crawl your link tags and generate your routes when running `nuxt generate`.

If have unlinked pages (such as secret pages) and you would like these to also be generated then you can use the `generate.routes` property.

Warning

Dynamic routes are ignored by the `generate` command when using `Nuxt <= v2.12`

Example:

```
-| pages/
---| index.vue
---| users/
-----| _id.vue
```

Only the route `/` will be generated by Nuxt.

If you want Nuxt to generate routes with dynamic params, you need to set the `generate.routes` property to an array of dynamic routes.

We add routes for `/users/:id`:

```
export default {
  generate: {
    routes: ['/users/1', '/users/2', '/users/3']
  }
}
```

Then when we launch `nuxt generate`:

```
[nuxt] Generating...
[...]
nuxt:render Rendering url / +154ms
nuxt:render Rendering url /users/1 +12ms
nuxt:render Rendering url /users/2 +33ms
nuxt:render Rendering url /users/3 +7ms
nuxt:generate Generate file: /index.html +21ms
nuxt:generate Generate file: /users/1/index.html +31ms
nuxt:generate Generate file: /users/2/index.html +15ms
nuxt:generate Generate file: /users/3/index.html +23ms
nuxt:generate HTML Files generated in 7.6s +6ms
[nuxt] Generate done
```

Great, but what if we have **dynamic params**?

1. Use a `Function` which returns a `Promise`.
2. Use a `Function` with a `callback(err, params)`.

Function which returns a Promise

```
import axios from 'axios'

export default {
  generate: {
    routes() {
      return axios.get('https://my-api/users').then(res => {
        return res.data.map(user => {
          return '/users/' + user.id
        })
      })
    }
  }
}
```

Function with a callback

```
import axios from 'axios'

export default {
  generate: {
    routes(callback) {
      axios
        .get('https://my-api/users')
        .then(res => {
          const routes = res.data.map(user => {
            return '/users/' + user.id
          })
          callback(null, routes)
        })
        .catch(callback)
    }
  }
}
```

Speeding up dynamic route generation with `payload`

In the example above, we're using the `user.id` from the server to generate the routes but tossing out the rest of the data. Typically, we need to fetch it again from inside the `/users/_id.vue`. While we can do that, we'll probably need to set the `generate.interval` to something like `100` in order not to flood the server with calls. Because this will increase the run time of the generate script, it would be preferable to pass along the entire `user` object to the context in `_id.vue`. We do that by modifying the code above to this:

```
import axios from 'axios'

export default {
  generate: {
    routes() {
      return axios.get('https://my-api/users').then(res => {
        return res.data.map(user => {
          return {
            route: '/users/' + user.id,
            payload: user
          }
        })
      })
    }
  }
}
```

Now we can access the `payload` from `/users/_id.vue` like so:

```
async asyncData ({ params, error, payload }) {
  if (payload) return { user: payload }
  else return { user: await backend.fetchUser(params.id) }
}
```

subFolders

- Type: `Boolean`
- Default: `true`

By default, when running `nuxt generate`, Nuxt will create a directory for each route & serve an `index.html` file.

Example:

```
-| dist/
---| index.html
---| about/
-----| index.html
---| products/
-----| item/
| index.html
```

When set to false, HTML files are generated according to the route path:

```
export default {
  generate: {
    subFolders: false
  }
}
```

```
-| dist/
---| index.html
---| about.html
---| products/
-----| item.html
```

Go to TOC

The globalName property

Nuxt lets you customize the global ID used in the main HTML template as well as the main Vue instance name and other options.

- Type: `String`
- Default: `nuxt`

```
{
  globalName: 'myCustomName'
}
```

Warning

The `globalName` needs to be a valid JavaScript identifier, and changing it may break support for certain plugins that rely on Nuxt-named functions. If you're looking to just change the visible `__nuxt` HTML ID, then use the `globals` property.

The globals property

Customizes specific global names which are based on `globalName` by default.

- Type: `Object`
- Default:

```
globals: {
  id: globalName => `__${globalName}`,
  nuxt: globalName => `$${globalName}`,
  context: globalName => `__${globalName.toUpperCase()}__`,
  pluginPrefix: globalName => globalName,
  readyCallback: globalName => `on${_.capitalize(globalName)}Ready`,
  loadedCallback: globalName => `_on${_.capitalize(globalName)}Loaded`
},
```

Go to TOC

The head property

Nuxt let you define all default meta for your application inside `nuxt.config.js`, use the same `head` property

---

- Type: `Object` or `Function`

```
export default {
  head: {
    titleTemplate: '%s - Nuxt',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' },

      // hid is used as unique identifier. Do not use `vmid` for it as it will not
work
      { hid: 'description', name: 'description', content: 'Meta description' }
    ]
  }
}
```

To know the list of options you can give to `head`, take a look at vue-meta documentation.

You can also use `head` as a function in your components to access the component data through `this` (read more).

Info

To avoid duplicated meta tags when used in child component, set up a unique identifier with the `hid` key for your meta elements (read more).

Go to TOC

The hooks property

Hooks are listeners to Nuxt events that are typically used in Nuxt modules, but are also available in `nuxt.config.js`. menu: hooks

---

- Type: `Object`

```
import fs from 'fs'
import path from 'path'

export default {
  hooks: {
    build: {
      done(builder) {
        const extraFilePath = path.join(
          builder.nuxt.options.buildDir,
          'extra-file'
        )
        fs.writeFileSync(extraFilePath, 'Something extra')
      }
    }
  }
}
```

Internally, hooks follow a naming pattern using colons (e.g., `build:done`). For ease of configuration, you can structure them as an hierarchical object when using `nuxt.config.js` (as exemplified above) to set your own hooks. See Nuxt Internals for more detailed information on how they work.

List of hooks

- Nuxt hooks
- Renderer hooks
- ModulesContainer hooks
- Builder hooks
- Generator hooks

Examples

Redirect to router.base when not on root

Let´s say you want to serve pages as `/portal` instead of `/`.

This is maybe an edge-case, and the point of *nuxt.config.js*' `router.base` is for when a web server will serve Nuxt elsewhere than the domain root.

But when in local development, hitting *localhost*, when router.base is not / returns a 404. In order to prevent this, you can setup a Hook.

Maybe redirecting is not the best use-case for a production Web site, but this will help you leverage Hooks.

To begin, you can change `router.base`; Update your `nuxt.config.js`:

```
import hooks from './hooks'
export default {
  router: {
    base: '/portal'
  }
  hooks: hooks(this)
}
```

Then, create a few files;

1. `hooks/index.js`, Hooks module

```
import render from './render'

export default nuxtConfig => ({
  render: render(nuxtConfig)
})
```

2. `hooks/render.js`, Render hook

```
import redirectRootToPortal from './route-redirect-portal'

export default nuxtConfig => {
  const router = Reflect.has(nuxtConfig, 'router') ? nuxtConfig.router : {}
  const base = Reflect.has(router, 'base') ? router.base : '/portal'

  return {
    /**
     * 'render:setupMiddleware'
     * {@link node_modules/nuxt/lib/core/renderer.js}
     */
    setupMiddleware(app) {
      app.use('/', redirectRootToPortal(base))
    }
  }
}
```

3. `hooks/route-redirect-portal.js`, The Middleware itself

```
/**
 * Nuxt middleware hook to redirect from / to /portal (or whatever we set in
nuxt.config.js router.base)
 *
 * Should be the same version as connect
 * {@link node_modules/connect/package.json}
 */
import parseurl from 'parseurl'

/**
 * Connect middleware to handle redirecting to desired Web Application Context
Root.
 *
 * Notice that Nuxt docs lacks explaining how to use hooks.
 * This is a sample router to help explain.
 *
 * See nice implementation for inspiration:
```

```
 * - https://github.com/nuxt/nuxt.js/blob/dev/examples/with-
cookies/plugins/cookies.js
 * - https://github.com/yyx990803/launch-editor/blob/master/packages/launch-
editor-middleware/index.js
 *
 * [http_class_http_clientrequest]:
https://nodejs.org/api/http.html#http_class_http_clientrequest
 * [http_class_http_serverresponse]:
https://nodejs.org/api/http.html#http_class_http_serverresponse
 *
 * @param {http.ClientRequest} req Node.js internal client request object
[http_class_http_clientrequest]
 * @param {http.ServerResponse} res Node.js internal response
[http_class_http_serverresponse]
 * @param {Function} next middleware callback
 */
export default desiredContextRoot =>
  function projectHooksRouteRedirectPortal(req, res, next) {
    const desiredContextRootRegExp = new RegExp(`^${desiredContextRoot}`)
    const _parsedUrl = Reflect.has(req, '_parsedUrl') ? req._parsedUrl : null
    const url = _parsedUrl !== null ? _parsedUrl : parseurl(req)
    const startsWithDesired = desiredContextRootRegExp.test(url.pathname)
    const isNotProperContextRoot = desiredContextRoot !== url.pathname
    if (isNotProperContextRoot && startsWithDesired === false) {
      const pathname = url.pathname === null ? '' : url.pathname
      const search = url.search === null ? '' : url.search
      const Location = desiredContextRoot + pathname + search
      res.writeHead(302, {
        Location
      })
      res.end()
    }
    next()
  }
```

Then, whenever a colleague in development accidentally hits `/` to reach the development web development service, Nuxt will automatically redirect to `/portal`

Go to TOC

The ignore property

Define the ignore files for your Nuxt application

.nuxtignore

You can use a `.nuxtignore` file to let Nuxt ignore `layout`, `page`, `store` and `middleware` files in your project's root directory (`rootDir`) during the build phase. The `.nuxtignore` file is subject to the same specification as `.gitignore` and `.eslintignore` files, in which each line is a glob pattern indicating which files should be ignored.

For example:

```
# ignore layout foo.vue
layouts/foo.vue
# ignore layout files whose name ends with -ignore.vue
layouts/*-ignore.vue

# ignore page bar.vue
pages/bar.vue
# ignore page inside ignore folder
pages/ignore/*.vue

# ignore store baz.js
store/baz.js
# ignore store files match *.test.*
store/ignore/*.test.*

# ignore middleware files under foo folder except foo/bar.js
middleware/foo/*.js
!middleware/foo/bar.js
```

More details about the spec are in gitignore doc

The ignorePrefix Property

- Type: `String`
- Default: `'-'`

Any file in pages/, layouts/, middleware/ or store/ will be ignored during building if its filename starts with the prefix specified by `ignorePrefix`.

By default all files which start with `-` will be ignored, such as `store/-foo.js` and `pages/-bar.vue`. This allows for co-locating tests, utilities, and components with their callers without themselves being converted into routes, stores, etc.

The ignore Property

- Type: `Array`
- Default: `['**/*.test.*']`

> More customizable than `ignorePrefix`: all files matching glob patterns specified inside `ignore` will be ignored in building.

ignoreOptions

`nuxtignore` is using `node-ignore` under the hood, `ignoreOptions` can be configured as `options` of `node-ignore`.

```
export default {
  ignoreOptions: {
    ignorecase: false
  }
}
```

Go to TOC

The loading property

Nuxt uses its own component to show a progress bar between the routes. You can customize it, disable it or create your own component.

- Type: `Boolean` or `Object` or `String`

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()

      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Disable the Progress Bar

- Type: `Boolean`

```
export default {
  loading: false
}
```

Customizing the Progress Bar

- Type: `Object`

```
export default {
  loading: {
    color: 'blue',
    height: '5px'
  }
}
```

List of properties to customize the progress bar. | Key | Type | Default | Description | | | | `color` | String | `'black'` | CSS color of the progress bar | | `failedColor` | String | `'red'` | CSS color of the progress bar when an error appended while rendering the route (if `data` or `fetch` sent back an error for example). | | `height` | String | `'2px'` | Height of the progress bar (used in the `style` property of the progress bar) | | `throttle` | Number | `200` | In ms, wait for the specified time before displaying the progress bar. Useful for preventing the bar from flashing. | | `duration` | Number | `5000` | In ms, the maximum duration of the progress bar, Nuxt assumes that the route will be rendered before 5 seconds. | | `continuous` | Boolean | `false` | Keep animating progress bar when loading takes longer than `duration`. | | `css` | Boolean | `true` | Set to false to remove default progress bar styles (and add your own). | | `rtl` | Boolean | `false` | Set the direction of the progress bar from right to left. |

Using a Custom Loading Component

- Type: `String`

Your component has to expose some of these methods:

| Method | Required | Description | | | `start()` | Required | Called when a route changes, this is where you display your component. | | `finish()` | Required | Called when a route is loaded (and data fetched), this is where you hide your component. | | `fail(error)` | *Optional* | Called when a route couldn't be loaded (failed to fetch data for example). | | `increase(num)` | *Optional* | Called during loading the route component, `num` is an Integer < 100. |

```
<template lang="html">
  <div class="loading-page" v-if="loading">
    <p>Loading...</p>
  </div>
</template>

<script>
  export default {
    data: () => ({
      loading: false
    }),
    methods: {
      start() {
        this.loading = true
      },
      finish() {
        this.loading = false
      }
    }
  }
</script>

<style scoped>
  .loading-page {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(255, 255, 255, 0.8);
    text-align: center;
    padding-top: 200px;
    font-size: 30px;
    font-family: sans-serif;
  }
</style>
```

```
export default {
  loading: '~/components/loading.vue'
}
```

Go to TOC

The loading indicator Property

Show fancy loading indicator while page is loading!

Without Server Side Rendering (when `ssr` option is `false`), there is no content from the server side on the first page load. So, instead of showing a blank page while the page loads, we may show a spinner.

This property can have 3 different types: `string` or `false` or `object`. If a string value is provided it is converted to object style.

Default value is:

```
loadingIndicator: {
  name: 'circle',
  color: '#3B8070',
  background: 'white'
}
```

Built-in indicators

These indicators are imported from the awesome SpinKit project. You can use its demo page to preview spinners.

- circle
- cube-grid
- fading-circle
- folding-cube
- chasing-dots
- nuxt
- pulse
- rectangle-bounce
- rotating-plane
- three-bounce
- wandering-cubes

Built-in indicators support `color` and `background` options.

Custom indicators

If you need your own special indicator, a String value or Name key can also be a path to an HTML template of indicator source code! All of the options are passed to the template, too.

Nuxt's built-in source code is also available if you need a base!

Go to TOC

The mode property

Change default nuxt mode

- Type: `string`
  - Default: `universal`
  - Possible values:
    - `'spa'`: No server-side rendering (only client-side navigation)
    - `'universal'`: Isomorphic application (server-side rendering + client-side navigation)

> You can use this option to change default nuxt mode for your project using `nuxt.config.js`

Warning

Deprecated: please use `ssr: false` instead of `mode: spa`

Next

To learn more about the `ssr` option, checkout the ssr property.

Next

To learn more about the `mode` option, checkout the rendering modes section.

Go to TOC

The modern property

Build and serve a modern bundle

This feature is inspired by vue-cli modern mode

- Type: `String` or `Boolean`
  - Default: false
  - Possible values:
    - `'client'`: Serve both, the modern bundle `<script type="module">` and the legacy bundle `<script nomodule>` scripts, also provide a `<link rel="modulepreload">` for the modern bundle. Every browser that understands the `module` type will load the modern bundle while older browsers fall back to the legacy (transpiled) one.
    - `'server'` or `true`: The Node.js server will check browser version based on the user agent and serve the corresponding modern or legacy bundle.
    - `false`: Disable modern build

The two versions of bundles are:

1. Modern bundle: targeting modern browsers that support ES modules
2. Legacy bundle: targeting older browsers based on babel config (IE9 compatible by default).

Info:

- Use command option `[--modern | -m]=[mode]` to build/start modern bundles:

```
{
  "scripts": {
    "build:modern": "nuxt build --modern=server",
    "start:modern": "nuxt start --modern=server"
  }
}
```

Note about *nuxt generate*: The `modern` property also works with the `nuxt generate` command, but in this case only the `client` option is honored and will be selected automatically when launching the `nuxt generate --modern` command without providing any values.

- Nuxt will automatically detect `modern` build in `nuxt start` when `modern` is not specified, auto-detected mode is:

ssr
true
false

- Modern mode for `nuxt generate` can only be `client`
- Use `render.crossorigin` to set `crossorigin` attribute in `<link>` and `<script>`

> Please refer Phillip Walton's excellent post for more knowledge regarding modern builds.

Go to TOC

The build property

Nuxt lets you customize the webpack configuration for building your web application as you want.

---

analyze

> Nuxt use webpack-bundle-analyzer to let you visualize your bundles and how to optimize them.

- Type: `Boolean` or `Object`
- Default: `false`

If an object, see available properties here.

```
export default {
  build: {
    analyze: true,
    // or
    analyze: {
      analyzerMode: 'static'
    }
  }
}
```

Info

Info: you can use the command `yarn nuxt build --analyze` or `yarn nuxt build -a` to build your application and launch the bundle analyzer on http://localhost:8888. If you are not using `yarn` you can run the command with `npx`.

corejs

> As of Nuxt@2.14 Nuxt automatically detects the current version of `core-js` in your project, also you can specify which version you want to use.

- Type: `number` | `string` (Valid values are `'auto'`, `2` and `3`)
- Default: `'auto'`

babel

> Customize Babel configuration for JavaScript and Vue files. `.babelrc` is ignored by default.

- Type: `Object`
- See `babel-loader` options and `babel` options
- Default:

```
{
  babelrc: false,
  cacheDirectory: undefined,
  presets: ['@nuxt/babel-preset-app']
}
```

The default targets of @nuxt/babel-preset-app are `ie: '9'` in the `client` build, and `node: 'current'` in the `server` build.

presets

- Type: `Function`
- Argument:
  1. `Object` : isServer[false]: true
  2. `Array` :
     - preset name `@nuxt/babel-preset-app`
     - `options` of `@nuxt/babel-preset-app`

Note: The presets configured in `build.babel.presets` will be applied to both, the client and the server build. The target will be set by Nuxt accordingly (client/server). If you want configure the preset differently for the client or the server build, please use `presets` as a function:

> We **highly recommend** to use the default preset instead of below customization

```
export default {
  build: {
    babel: {
      presets({ isServer }, [ preset, options ]) {
        // change options directly
        options.targets = isServer ? ... :  ...
        options.corejs = ...
        // return nothing
      }
    }
  }
}
```

Or override default value by returning whole presets list:

```
export default {
  build: {
    babel: {
      presets({ isServer }, [preset, options]) {
        return [
          [
```

```
            preset,
            {
              targets: isServer ? ... :  ...,
              ...options
            }
          ],
          [
            // Other presets
          ]
        ]
      }
    }
  }
}
```

cache

- Type: `Boolean`
- Default: `false`
- ⚠ Experimental

Enable cache of terser-webpack-plugin and cache-loader

cssSourceMap

- Type: `boolean`
- Default: `true` for dev and `false` for production.

Enables CSS Source Map support

devMiddleware

- Type: `Object`

See webpack-dev-middleware for available options.

devtools

- Type: `boolean`
- Default: `false`

Configure whether to allow vue-devtools inspection.

If you already activated through nuxt.config.js or otherwise, devtools enable regardless of the flag.

extend

> Extend the webpack configuration manually for the client & server bundles.

- Type: `Function`

The extend is called twice, one time for the server bundle, and one time for the client bundle. The arguments of the method are:

1. The Webpack config object,
2. An object with the following keys (all boolean except `loaders`): `isDev`, `isClient`, `isServer`, `loaders`.

Warning

Warning: The `isClient` and `isServer` keys provided in are separate from the keys available in `context`. They are **not** deprecated. Do not use `process.client` and `process.server` here as they are `undefined` at this point.

```
export default {
  build: {
    extend(config, { isClient }) {
      // Extend only webpack config for client-bundle
      if (isClient) {
        config.devtool = 'source-map'
      }
    }
  }
}
```

If you want to see more about our default webpack configuration, take a look at our webpack directory.

loaders in extend

`loaders` has the same object structure as build.loaders, so you can change the options of loaders inside `extend`.

```
export default {
  build: {
    extend(config, { isClient, loaders: { vue } }) {
      // Extend only webpack config for client-bundle
      if (isClient) {
        vue.transformAssetUrls.video = ['src', 'poster']
      }
    }
  }
}
```

extractCSS

> Enables Common CSS Extraction using Vue Server Renderer guidelines.

- Type: `Boolean` or `Object`
- Default: `false`

Using `extract-css-chunks-webpack-plugin` under the hood, all your CSS will be extracted into separate files, usually one per component. This allows caching your CSS and JavaScript separately and is worth a try in case you have a lot of global or shared CSS.

Example (`nuxt.config.js`):

```
export default {
  build: {
    extractCSS: true,
    // or
    extractCSS: {
      ignoreOrder: true
    }
  }
}
```

Info

Note: There was a bug prior to Vue 2.5.18 that removed critical CSS imports when using this options.

You may want to extract all your CSS to a single file. There is a workaround for this:

Warning

It is not recommended to extract everything into a single file. Extracting into multiple CSS files is better for caching and preload isolation. It can also improve page performance by downloading and resolving only those resources that are needed.

```
export default {
  build: {
    extractCSS: true,
    optimization: {
      splitChunks: {
        cacheGroups: {
          styles: {
            name: 'styles',
            test: /\.(css|vue)$/,
            chunks: 'all',
            enforce: true
          }
        }
      }
    }
  }
}
```

filenames

Customize bundle filenames.

- Type: `Object`
- Default:

```
{
  app: ({ isDev, isModern }) => isDev ? `[name]${isModern ? '.modern' : ''}.js`
: `[contenthash:7]${isModern ? '.modern' : ''}.js`,
  chunk: ({ isDev, isModern }) => isDev ? `[name]${isModern ? '.modern' :
''}.js` : `[contenthash:7]${isModern ? '.modern' : ''}.js`,
  css: ({ isDev }) => isDev ? '[name].css' : 'css/[contenthash:7].css',
  img: ({ isDev }) => isDev ? '[path][name].[ext]' : 'img/[name].
[contenthash:7].[ext]',
  font: ({ isDev }) => isDev ? '[path][name].[ext]' : 'fonts/[name].
[contenthash:7].[ext]',
  video: ({ isDev }) => isDev ? '[path][name].[ext]' : 'videos/[name].
[contenthash:7].[ext]'
}
```

This example changes fancy chunk names to numerical ids:

```
export default {
  build: {
    filenames: {
      chunk: ({ isDev }) => (isDev ? '[name].js' : '[id].[contenthash].js')
    }
  }
}
```

To understand a bit more about the use of manifests, take a look at this webpack documentation.

Warning

Be careful when using non-hashed based filenames in production as most browsers will cache the asset and not detect the changes on first load.

friendlyErrors

- Type: `Boolean`
- Default: `true` (Overlay enabled)

Enables or disables the overlay provided by FriendlyErrorsWebpackPlugin

hardSource

- Type: `Boolean`
- Default: `false`

- ⚠ Experimental

Enables the HardSourceWebpackPlugin for improved caching

hotMiddleware

- Type: `Object`

See webpack-hot-middleware for available options.

html.minify

- Type: `Object`
- Default:

```
{
  collapseBooleanAttributes: true,
  decodeEntities: true,
  minifyCSS: true,
  minifyJS: true,
  processConditionalComments: true,
  removeEmptyAttributes: true,
  removeRedundantAttributes: true,
  trimCustomFragments: true,
  useShortDoctype: true
}
```

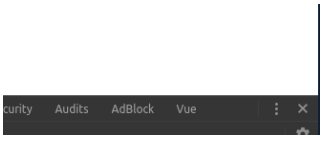
Attention: If you make changes to `html.minify`, they won't be merged with the defaults!

Configuration for the html-minifier plugin used to minify HTML files created during the build process (will be applied for *all modes*).

indicator

Display build indicator for hot module replacement in development (available in `v2.8.0+`)

- Type: `Boolean`
- Default: `true`



nuxt-build-indicator

loaders

Customize options of Nuxt integrated webpack loaders.

- Type: `Object`
- Default:

```
{
  file: {},
  fontUrl: { limit: 1000 },
  imgUrl: { limit: 1000 },
  pugPlain: {},
  vue: {
    transformAssetUrls: {
      video: 'src',
      source: 'src',
      object: 'src',
      embed: 'src'
    }
  },
  css: {},
  cssModules: {
    localIdentName: '[local]_[hash:base64:5]'
  },
  less: {},
  sass: {
    indentedSyntax: true
  },
  scss: {},
  stylus: {},
  vueStyle: {}
}
```

Note: In addition to specifying the configurations in `nuxt.config.js`, it can also be modified by build.extend

loaders.file

More details are in file-loader options.

loaders.fontUrl and loaders.imgUrl

More details are in url-loader options.

loaders.pugPlain

More details are in pug-plain-loader or Pug compiler options.

loaders.vue

More details are in vue-loader options.

loaders.css and loaders.cssModules

More details are in css-loader options. Note: cssModules is loader options for usage of CSS Modules

loaders.less

You can pass any Less specific options to the `less-loader` via `loaders.less`. See the Less documentation for all available options in dash-case.

loaders.sass and loaders.scss

See the Sass documentation for all available Sass options. Note: `loaders.sass` is for Sass Indented Syntax

loaders.vueStyle

More details are in vue-style-loader options.

optimization

- Type: `Object`
- Default:

```
{
  minimize: true,
  minimizer: [
    // terser-webpack-plugin
    // optimize-css-assets-webpack-plugin
```

```
    ],
    splitChunks: {
      chunks: 'all',
      automaticNameDelimiter: '.',
      name: undefined,
      cacheGroups: {}
    }
  }
```

The default value of `splitChunks.name` is `true` in `dev` or `analyze` mode.

You can set `minimizer` to a customized Array of plugins or set `minimize` to `false` to disable all minimizers. (`minimize` is being disabled for development by default)

See Webpack Optimization.

optimizeCSS

- Type: `Object` or `Boolean`
- Default:
  - `false`
  - `{}` when extractCSS is enabled

OptimizeCSSAssets plugin options.

See NMFR/optimize-css-assets-webpack-plugin.

parallel

- Type: `Boolean`
- Default: `false`
- ⚠ Experimental

Enable thread-loader in webpack building

plugins

Add webpack plugins

- Type: `Array`
- Default: `[]`

```
import webpack from 'webpack'
import { version } from './package.json'
export default {
  build: {
```

```
    plugins: [
      new webpack.DefinePlugin({
        'process.VERSION': version
      })
    ]
  }
}
```

postcss

Customize PostCSS Loader plugins.

- Type: `Array` (legacy, will override defaults), `Object` (recommended), `Function` or `Boolean`

  Note: Nuxt has applied PostCSS Preset Env. By default it enables Stage 2 features and Autoprefixer, you can use `build.postcss.preset` to configure it.

- Default:

```
{
  plugins: {
    'postcss-import': {},
    'postcss-url': {},
    'postcss-preset-env': this.preset,
    'cssnano': { preset: 'default' } // disabled in dev mode
  },
  order: 'presetEnvAndCssnanoLast',
  preset: {
    stage: 2
  }
}
```

Your custom plugin settings will be merged with the default plugins (unless you are using an `Array` instead of an `Object`).

```
export default {
  build: {
    postcss: {
      plugins: {
        // Disable `postcss-url`
        'postcss-url': false,
        // Add some plugins
        'postcss-nested': {},
        'postcss-responsive-type': {},
        'postcss-hexrgba': {}
      },
      preset: {
        autoprefixer: {
          grid: true
        }
      }
    }
  }
}
```

If the postcss configuration is an `Object`, `order` can be used for defining the plugin order:

- Type: `Array` (ordered plugin names), `String` (order preset name), `Function`
- Default: `cssnanoLast` (put `cssnano` in last)

```
export default {
  build: {
    postcss: {
      // preset name
      order: 'cssnanoLast',
      // ordered plugin names
      order: ['postcss-import', 'postcss-preset-env', 'cssnano']
      // Function to calculate plugin order
      order: (names, presets) => presets.cssnanoLast(names)
    }
  }
}
```

postcss plugins & @nuxtjs/tailwindcss

If you want to apply a postcss plugin (e.g. postcss-pxtorem) on the @nuxtjs/tailwindcss configuration, you have to change order and load tailwindcss first.

This setup has no impact on nuxt-purgecss.

```
import { join } from 'path'

export default {
  // ...
  build: {
    postcss: {
      plugins: {
        tailwindcss: join(__dirname, 'tailwind.config.js'),
        'postcss-pxtorem': {
          propList: ['*', '!border*']
        }
      }
    }
  }
}
```

profile

- Type: `Boolean`
- Default: enabled by command line argument `--profile`

Enable the profiler in WebpackBar

publicPath

Nuxt lets you upload your dist files to your CDN for maximum performances, simply set the `publicPath` to your CDN.

- Type: `String`
- Default: `'/_nuxt/'`

```
export default {
  build: {
    publicPath: 'https://cdn.nuxtjs.org'
  }
}
```

Then, when launching `nuxt build`, upload the content of `.nuxt/dist/client` directory to your CDN and voilà!

In Nuxt 2.15+, changing the value of this property at runtime will override the configuration of an app that has already been built.

quiet

Suppresses most of the build output log

- Type: `Boolean`
- Default: Enabled when a `CI` or `test` environment is detected by std-env

splitChunks

- Type: `Object`
- Default:

```
export default {
  build: {
    splitChunks: {
      layouts: false,
      pages: true,
      commons: true
    }
  }
}
```

Whether or not to create separate chunks for `layout`, `pages` and `commons` (common libs: vue|vue-loader|vue-router|vuex...). For more information, see webpack docs.

ssr

> Creates special webpack bundle for SSR renderer.

- Type: `Boolean`
- Default: `true` for universal mode and `false` for spa mode

This option is automatically set based on `mode` value if not provided.

standalone

> Inline server bundle dependencies (advanced)

- Type: `Boolean`
- Default: `false`

This mode bundles `node_modules` that are normally preserved as externals in the server build (more information).

Warning

Runtime dependencies (modules, `nuxt.config`, server middleware and static directory) are not bundled. This feature only disables use of webpack-externals for server-bundle.

Info

you can use the command `yarn nuxt build --standalone` to enable this mode on the command line. (If you are not using `yarn` you can run the command with `npx`.)

styleResources

- Type: `Object`
- Default: `{}`

Warning

This property is deprecated. Please use the style-resources-module instead for improved performance and better DX!

This is useful when you need to inject some variables and mixins in your pages without having to import them every time.

Nuxt uses https://github.com/yenshih/style-resources-loader to achieve this behavior.

You need to specify the patterns/path you want to include for the given pre-processors: `less`, `sass`, `scss` or `stylus`

You cannot use path aliases here (`~` and `@`), you need to use relative or absolute paths.

```
{
  build: {
    styleResources: {
      scss: './assets/variables.scss',
      less: './assets/*.less',
      // sass: ...,
      // scss: ...
      options: {
        // See https://github.com/yenshih/style-resources-loader#options
        // Except `patterns` property
      }
    }
  }
}
```

templates

Nuxt allows you provide your own templates which will be rendered based on Nuxt configuration. This feature is specially useful for using with modules.

- Type: `Array`

```
export default {
  build: {
    templates: [
      {
        src: '~/modules/support/plugin.js', // `src` can be absolute or relative
        dst: 'support.js', // `dst` is relative to project `.nuxt` dir
        options: {
          // Options are provided to template as `options` key
          live_chat: false
        }
      }
    ]
  }
}
```

Templates are rendered using `lodash.template` you can learn more about using them here.

terser

- Type: `Object` or `Boolean`
- Default:

```
{
  parallel: true,
  cache: false,
```

```
    sourceMap: false,
    extractComments: {
      filename: 'LICENSES'
    },
    terserOptions: {
      output: {
        comments: /^\**!|@preserve|@license|@cc_on/
      }
    }
  }
```

Terser plugin options. Set to `false` to disable this plugin.

Enabling `sourceMap` will leave `//# sourceMappingURL` linking comment at the end of each output file if webpack `config.devtool` is set to `source-map`.

See webpack-contrib/terser-webpack-plugin.

transpile

- Type: `Array<String | RegExp | Function>`
- Default: `[]`

If you want to transpile specific dependencies with Babel, you can add them in `build.transpile`. Each item in transpile can be a package name, a string or regex object matching the dependency's file name.

Starting with `v2.9.0`, you can also use a function to conditionally transpile, the function will receive a object (`{ isDev, isServer, isClient, isModern, isLegacy }`):

```
{
  build: {
    transpile: [({ isLegacy }) => isLegacy && 'ky']
  }
}
```

In this example, `ky` will be transpiled by Babel if Nuxt is not in modern mode.

vueLoader

Note: This config has been removed since Nuxt 2.0, please use `build.loaders.vue` instead.

- Type: `Object`
- Default:

```
{
  productionMode: !this.options.dev,
  transformAssetUrls: {
    video: 'src',
    source: 'src',
    object: 'src',
```

```
      embed: 'src'
    }
  }
```

Specify the Vue Loader Options.

watch

You can provide your custom files to watch and regenerate after changes. This feature is specially useful for using with modules.

- Type: `Array<String>`

```
export default {
  build: {
    watch: ['~/.nuxt/support.js']
  }
}
```

followSymlinks

By default, the build process does not scan files inside symlinks. This boolean includes them, thus allowing usage of symlinks inside folders such as the "pages" folder, for example.

- Type: `Boolean`

```
export default {
  build: {
    followSymlinks: true
  }
}
```

Go to TOC

The modules property

Modules are Nuxt extensions which can extend its core functionality and add endless integrations. Learn More

---

- Type: `Array`

Example (`nuxt.config.js`):

```
export default {
  modules: [
    // Using package name
    '@nuxtjs/axios',

    // Relative to your project srcDir
    '~/modules/awesome.js',

    // Providing options
    ['@nuxtjs/google-analytics', { ua: 'X1234567' }],

    // Inline definition
    function () {}
  ]
}
```

Module developers usually provide additionally needed steps and details for usage.

Nuxt tries to resolve each item in the modules array using node require path (in the `node_modules`) and then will be resolved from project `srcDir` if `~` alias is used. Modules are executed sequentially so the order is important.

Note: Any plugins injected by modules are added to the *beginning* of the plugins list. Your options are to:

- Manually add your plugin to the end of the list of plugins (`this.nuxt.options.plugins.push(...)`)
- Reverse the order of the modules if it depends on another

Modules should export a function to enhance nuxt build/runtime and optionally return a promise until their job is finished. Note that they are required at runtime so should be already transpiled if depending on modern ES6 features.

Please see Modules Guide for more detailed information on how they work or if interested developing your own module. Also we have provided an official Modules Section listing dozens of production ready modules made by Nuxt Community.

`buildModules`

Info

This feature is available since Nuxt v2.9

Some modules are only required during development and build time. Using `buildModules` helps to make production startup faster and also significantly decreasing `node_modules` size for production deployments. Please refer to each module docs to see if it is recommended to use `modules` or `buildModules`.

The usage difference is:

- Instead of adding to `modules` inside `nuxt.config.js`, use `buildModules`
- Instead of adding to `dependencies` inside `package.json`, use `devDependencies` (`yarn add --dev` or `npm install --save-dev`)

Go to TOC

The modulesDir property

Define the modules directory for your Nuxt application

- Type: `Array`
- Default: `['node_modules']`

> Used to set the modules directories for path resolving, for example: Webpack's `resolveLoading`, `nodeExternals` and `postcss`. Configuration path is relative to options.rootDir (default: `process.cwd()`).

```
export default {
  modulesDir: ['../../node_modules']
}
```

Setting this field may be necessary if your project is organized as a Yarn workspace-styled mono-repository.

Go to TOC

The plugins property

Use vue.js plugins with the plugins option of Nuxt.

Note: Since Nuxt 2.4, `mode` has been introduced as option of `plugins` to specify plugin type, possible value are: `client` or `server`. `ssr: false` will be adapted to `mode: 'client'` and deprecated in next major release.

- Type: `Array`
  - Items: `String` or `Object`

If the item is an object, the properties are:

- src: `String` (path of the file)
- mode: `String` (can be `client` or `server`) *If defined, the file will be included only on the respective (client or server) side.*

Note: Old version

- Type: `Array`
  - Items: `String` or `Object`

If the item is an object, the properties are:

- src: `String` (path of the file)
- ssr: `Boolean` (default to `true`) *If false, the file will be included only on the client-side.*

> The plugins property lets you add Vue.js plugins easily to your main application.

```
export default {
  plugins: [
    { src: '~/plugins/both-sides.js' },
    { src: '~/plugins/client-only.js', mode: 'client' },
    { src: '~/plugins/server-only.js', mode: 'server' }
  ]
}
```

```
export default {
  plugins: ['@/plugins/ant-design-vue']
}
```

```
import Vue from 'vue'
import Antd from 'ant-design-vue'
import 'ant-design-vue/dist/antd.css' // Per Ant Design's docs

Vue.use(Antd)
```

Note that the css was imported as per Ant Design Documentation

All the paths defined in the `plugins` property will be **imported** before initializing the main application.

Go to TOC

The render property

Nuxt lets you customize runtime options for rendering pages

---

bundleRenderer

- Type: `Object`

> Use this option to customize vue SSR bundle renderer. This option is skipped if `ssr: false`.

```
export default {
  render: {
    bundleRenderer: {
      directives: {
        custom1(el, dir) {
          // something ...
        }
      }
    }
  }
}
```

Learn more about available options on Vue SSR API Reference. It is recommended to not use this option as Nuxt is already providing best SSR defaults and misconfiguration might lead to SSR problems.

etag

- Type: `Object`
  - Default: `{ weak: true }`

To disable etag for pages set `etag: false`

See etag docs for possible options.

You can use your own hash function by specifying `etag.hash`:

```
import { murmurHash128 } from 'murmurhash-native'

export default {
  render: {
    etag: {
      hash: html => murmurHash128(html)
    }
  }
}
```

In this case we use murmurhash-native, which is faster for larger HTML body sizes. Note that the `weak` option is ignored, when specifying your own hash function.

compressor

- Type `Object`
  - Default: `{ threshold: 0 }`

When providing an object, the compression middleware will be used (with respective options).

If you want to use your own compression middleware, you can reference it directly (e.g. `otherComp({ myOptions: 'example' })`).

To disable compression, use `compressor: false`.

fallback

- Type `Object`
  - Default: `{ dist: {}, static: { skipUnknown: true } }`
  - `dist` key is for routes matching the publicPath (ie: `/_nuxt/*`)
  - `static` key is for routes matching `/*`

> `dist` and `static` values are forwarded to serve-placeholder middleware.

If you want to disable one of them or both, you can pass a falsy value.

Example of allowing `.js` extension for routing (ex: `/repos/nuxt.js`):

```
export default {
  render: {
    fallback: {
      static: {
        // Avoid sending 404 for these extensions
        handlers: {
          '.js': false
        }
      }
    }
  }
}
```

http2

- Type `Object`
  - Default: `{ push: false, pushAssets: null }`

> Activate HTTP2 push headers.

You can control what links to push using `pushAssets` function.

Example:

```
pushAssets: (req, res, publicPath, preloadFiles) =>
  preloadFiles
    .filter(f => f.asType === 'script' && f.file === 'runtime.js')
    .map(f => `<${publicPath}${f.file}>; rel=preload; as=${f.asType}`)
```

You can add your own assets to the array as well. Using `req` and `res` you can decide what links to push based on the request headers, for example using the cookie with application version.

The assets will be joined together with `,` and passed as a single `Link` header.

asyncScripts

- Type: `Boolean`
  - Default: `false`

Adds an `async` attribute to `<script>` tags for Nuxt bundles, enabling them to be fetched in parallel to parsing (available with `2.14.8+`). More information.

injectScripts

- Type: `Boolean`
  - Default: `true`

Adds the `<script>` for Nuxt bundles, set it to `false` to render pure HTML without JS (available with `2.8.0+`)

resourceHints

- Type: `Boolean`
  - Default: `true`

Adds `prefetch` and `preload` links for faster initial page load time.

You may want to only disable this option if you have many pages and routes.

ssr

- Type: `Boolean`
  - Default: `true`

- `false` only client side rendering

> Enable SSR rendering

This option is automatically set based on global `ssr` value if not provided. This can be useful to dynamically enable/disable SSR on runtime after image builds (with docker for example).

crossorigin

- Type: `String`
- Default: `undefined`

  Configure the `crossorigin` attribute on `<link rel="stylesheet">` and `<script>` tags in generated HTML.

  More Info: CORS settings attributes

ssrLog

- Type: `Boolean` | `String`
  - Default: `true` in dev mode and `false` in production

> Forward server-side logs to the browser for better debugging (only available in development)

To collapse the logs, use `'collapsed'` value.

static

- Type: `Object`
  - Default: `{}`

> Configure the `static/` directory behavior

See serve-static docs for possible options.

Additional to them, we introduced a `prefix` option which defaults to `true`. It will add the router base to your static assets.

Example:

- Assets: `favicon.ico`

- Router base: `/t`
- With `prefix: true` (default): `/t/favicon.ico`
- With `prefix: false`: `/favicon.ico`

Caveats:

Some URL rewrites might not respect the prefix.

dist

- Type: `Object`
  - Default: `{ maxAge: '1y', index: false }`

> Options used for serving distribution files. Only applicable in production.

See serve-static docs for possible options.

csp

- Type: `Boolean` or `Object`
  - Default: `false`

> Use this to configure Content-Security-Policy to load external resources

Prerequisites:

These CSP settings are only effective when using Nuxt with `target: 'server'` to serve your SSR application. The Policies defined under `csp.policies` are added to the response `Content-Security-Policy` HTTP header.

Updating settings:

These settings are read by the Nuxt server directly from `nuxt.config.js`. This means changes to these settings take effect when the server is restarted. There is no need to rebuild the application to update the CSP settings.

HTML meta tag:

In order to add `<meta http-equiv="Content-Security-Policy"/>` to the `<head>` you need to set `csp.addMeta` to `true`. Please note that this feature is independent of the `csp.policies` configuration:

- it only adds a `script-src` type policy, and
- the `script-src` policy only contains the hashes of the inline `<script>` tags.

When `csp.addMeta` is set to `true`, the complete set of the defined policies are still added to the HTTP response header.

Note that CSP hashes will not be added as `<meta>` if `script-src` policy contains `'unsafe-inline'`. This is due to browser ignoring `'unsafe-inline'` if hashes are present. Set option `unsafeInlineCompatibility` to `true` if you want both hashes and `'unsafe-inline'` for CSPv1 compatibility. In that case the `<meta>` tag will still only contain the hashes of the inline `<script>` tags, and the policies defined under `csp.policies` will be used in the `Content-Security-Policy` HTTP response header.

```
export default {
  render: {
    csp: true
  }
}

// OR

export default {
  render: {
    csp: {
      hashAlgorithm: 'sha256',
      policies: {
        'script-src': [
          'https://www.google-analytics.com',
          'https://name.example.com'
        ],
        'report-uri': ['https://report.example.com/report-csp-violations']
      },
      addMeta: true
    }
  }
}

// OR
/*
  The following example allows Google Analytics, LogRocket.io, and Sentry.io
  for logging and analytic tracking.

  Review to this blog on Sentry.io
  https://blog.sentry.io/2018/09/04/how-sentry-captures-csp-violations

  To learn what tracking link you should use.
*/
const PRIMARY_HOSTS = `loc.example-website.com`
export default {
  render: {
    csp: {
      reportOnly: true,
      hashAlgorithm: 'sha256',
      policies: {
        'default-src': ["'self'"],
        'img-src': ['https:', '*.google-analytics.com'],
        'worker-src': ["'self'", `blob:`, PRIMARY_HOSTS, '*.logrocket.io'],
        'style-src': ["'self'", "'unsafe-inline'", PRIMARY_HOSTS],
        'script-src': [
          "'self'",
          "'unsafe-inline'",
          PRIMARY_HOSTS,
          'sentry.io',
```

```
            '*.sentry-cdn.com',
            '*.google-analytics.com',
            '*.logrocket.io'
          ],
          'connect-src': [PRIMARY_HOSTS, 'sentry.io', '*.google-analytics.com'],
          'form-action': ["'self'"],
          'frame-ancestors': ["'none'"],
          'object-src': ["'none'"],
          'base-uri': [PRIMARY_HOSTS],
          'report-uri': [
            `https://sentry.io/api/<project>/security/?sentry_key=<key>`
          ]
        }
      }
    }
}
```

Go to TOC

The rootDir property

Define the workspace of Nuxt application

- Type: `String`
- Default: `process.cwd()`

This property will be overwritten by the nuxt commands(nuxt start, nuxt build etc) if an argument is passed to them. Eg running `nuxt ./my-app/` will set the `rootDir` to the absolute path of `./my-app/` from the current/working directory.

Because of that its normally not needed to configure this option unless you will use Nuxt programmatically.

Info

Both `rootDir` as the package root containing the `node_modules` directory need to be within the same directory tree to be able to resolve dependencies. See the `srcDir` option for examples of directory structure when that is not the case.

Go to TOC

The router property

The router property lets you customize Nuxt router. (vue-router).

---

base

- Type: `String`
- Default: `'/'`

The base URL of the app. For example, if the entire single page application is served under `/app/`, then base should use the value `'/app/'`.

This can be useful if you need to serve Nuxt as a different context root, from within a bigger Web site. Notice that you may, or may not set up a Front Proxy Web Server.

If you want to have a redirect to `router.base`, you can do so using a Hook, see *Redirect to router.base when not on root*.

In Nuxt 2.15+, changing the value of this property at runtime will override the configuration of an app that has already been built.

```
export default {
  router: {
    base: '/app/'
  }
}
```

Info

When `base` is set, Nuxt will also add in the document header `<base href="{{ router.base }}"/>`.

This option is given directly to the vue-router base.

routeNameSplitter

- Type: `String`
- Default: `'-'`

You may want to change the separator between route names that Nuxt uses. You can do so via the `routeNameSplitter` option in your configuration file. Imagine we have the page file `pages/posts/_id.vue`. Nuxt will generate the route name programmatically, in this case `posts-id`. Changing the `routeNameSplitter` config to `/` the name will therefore change to `posts/id`.

```
export default {
  router: {
    routeNameSplitter: '/'
  }
}
```

extendRoutes

- Type: `Function`

You may want to extend the routes created by Nuxt. You can do so via the `extendRoutes` option.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        name: 'custom',
        path: '*',
        component: resolve(__dirname, 'pages/404.vue')
      })
    }
  }
}
```

If you want to sort your routes, you can use the `sortRoutes(routes)` function from `@nuxt/utils`:

```
import { sortRoutes } from '@nuxt/utils'
export default {
  router: {
    extendRoutes(routes, resolve) {
      // Add some routes here ...

      // and then sort them
      sortRoutes(routes)
    }
  }
}
```

The schema of the route should respect the vue-router schema.

Warning

When adding routes that use Named Views, don't forget to add the corresponding `chunkNames` of named `components`.

```
export default {
  router: {
    extendRoutes(routes, resolve) {
      routes.push({
        path: '/users/:id',
        components: {
          default: resolve(__dirname, 'pages/users'), // or
routes[index].component
          modal: resolve(__dirname, 'components/modal.vue')
        },
        chunkNames: {
          modal: 'components/modal'
```

```
        }
      })
    }
  }
}
```

fallback

- Type: `boolean`
- Default: `false`

Controls whether the router should fallback to hash mode when the browser does not support history.push-State but mode is set to history.

Setting this to false essentially makes every router-link navigation a full page refresh in IE9. This is useful when the app is server-rendered and needs to work in IE9, because a hash mode URL does not work with SSR.

This option is given directly to the vue-router fallback.

linkActiveClass

- Type: `String`
- Default: `'nuxt-link-active'`

Globally configure `<nuxt-link>` default active class.

```
export default {
  router: {
    linkActiveClass: 'active-link'
  }
}
```

This option is given directly to the vue-router linkActiveClass.

linkExactActiveClass

- Type: `String`
- Default: `'nuxt-link-exact-active'`

Globally configure `<nuxt-link>` default exact active class.

```
export default {
  router: {
    linkExactActiveClass: 'exact-active-link'
  }
}
```

This option is given directly to the vue-router linkExactActiveClass.

linkPrefetchedClass

- Type: `String`
- Default: `false`

Globally configure `<nuxt-link>` default prefetch class (feature disabled by default)

```
export default {
  router: {
    linkPrefetchedClass: 'nuxt-link-prefetched'
  }
}
```

middleware

- Type: `String` or `Array`
  - Items: `String`

Set the default(s) middleware for every page of the application.

```
export default {
  router: {
    // Run the middleware/user-agent.js on every page
    middleware: 'user-agent'
  }
}
```

```
export default function (context) {
  // Add the userAgent property in the context (available in `asyncData` and
`fetch`)
  context.userAgent = process.server
    ? context.req.headers['user-agent']
    : navigator.userAgent
}
```

To learn more about the middleware, see the middleware guide.

mode

- Type: `String`
- Default: `'history'`

Configure the router mode, this is not recommended to change it due to server-side rendering.

```
export default {
  router: {
    mode: 'hash'
  }
}
```

> This option is given directly to the vue-router mode.

parseQuery / stringifyQuery

- Type: `Function`

Provide custom query string parse / stringify functions. Overrides the default.

> This option is given directly to the vue-router parseQuery / stringifyQuery.

prefetchLinks

> Added with Nuxt v2.4.0

- Type: `Boolean`
- Default: `true`

Configure `<nuxt-link>` to prefetch the *code-splitted* page when detected within the viewport. Requires IntersectionObserver to be supported (see Caniuse).

We recommend conditionally polyfilling this feature with a service like Polyfill.io:

```
export default {
  head: {
    script: [
      {
        src:
          'https://polyfill.io/v3/polyfill.min.js?features=IntersectionObserver',
        body: true
      }
    ]
  }
}
```

To disable the prefetching on a specific link, you can use the `no-prefetch` prop. Since Nuxt v2.10.0, you can also use the `prefetch` prop set to `false`:

```
<nuxt-link to="/about" no-prefetch>About page not prefetched</nuxt-link>
<nuxt-link to="/about" :prefetch="false">About page not prefetched</nuxt-link>
```

To disable the prefetching on all links, set the `prefetchLinks` to `false`:

```
export default {
  router: {
    prefetchLinks: false
  }
}
```

Since Nuxt v2.10.0, if you have set `prefetchLinks` to `false` but you want to prefetch a specific link, you can use the `prefetch` prop:

```
<nuxt-link to="/about" prefetch>About page prefetched</nuxt-link>
```

prefetchPayloads

> Added with v2.13.0, only available for static target.

- Type: `Boolean`
- Default: `true`

When using `nuxt generate` with `target: 'static'`, Nuxt will generate a `payload.js` for each page.

With this option enabled, Nuxt will automatically prefetch the payload of the linked page when the `<nuxt-link>` is visible in the viewport, making **instant navigation**.

Info

This option depends of the prefetchLinks option to be enabled.

You can disable this behavior by setting `prefetchPaylods` to `false`:

```
export default {
  router: {
    prefetchPayloads: false
  }
}
```

scrollBehavior

- Type: `Function`

The `scrollBehavior` option lets you define a custom behavior for the scroll position between the routes. This method is called every time a page is rendered. To learn more about it, see vue-router scrollBehavior documentation.

Starting from v2.9.0, you can use a file to overwrite the router scrollBehavior, this file should be placed in `~/app/router.scrollBehavior.js` (note: filename is case-sensitive if running on Windows).

Warning

The `router.scrollBehavior.js` file must be in the `app` folder, which in turn is in the project's root.

You can see Nuxt default `router.scrollBehavior.js` file here: packages/vue-app/template/router.scrollBehavior.js.

Example of forcing the scroll position to the top for every routes:

`app/router.scrollBehavior.js`

```
export default function (to, from, savedPosition) {
  return { x: 0, y: 0 }
}
```

trailingSlash

- Type: `Boolean` or `undefined`
- Default: `undefined`
- Available since: v2.10

If this option is set to true, trailing slashes will be appended to every route. If set to false, they'll be removed.

Attention: This option should not be set without preparation and has to be tested thoroughly. When setting `router.trailingSlash` to something else than `undefined`, the opposite route will stop working. Thus 301 redirects should be in place and your *internal linking* has to be adapted correctly. If you set `trailingSlash` to `true`, then only `example.com/abc/` will work but not `example.com/abc`. On false, it's vice-versa

Example behavior (with child routes)

For a directory with this structure:

```
-| pages/
---| index.vue
---| posts.vue
---| posts/
-----| _slug.vue
-----| index.vue
```

This is the behavior for each possible setting of `trailingSlash`:

```
Route           Page
/               ~/pages/index.vue
/posts          ~/pages/posts.vue (parent) + ~/pages/posts.vue (child route)
/posts/         ~/pages/posts.vue (parent) + ~/pages/posts.vue (child route)
/posts/foo      ~/pages/posts.vue (parent) + ~/pages/_slug.vue (child route)
/posts/foo/     ~/pages/posts.vue (parent) + ~/pages/_slug.vue (child route)
```

```
Route           Page
/               ~/pages/index.vue
/posts          404
/posts/         ~/pages/posts.vue (parent) + ~/pages/index.vue (child route)
/posts/foo      404
/posts/foo/     ~/pages/posts.vue (parent) + ~/pages/_slug.vue (child route)
```

```
Route           Page
/               ~/pages/index.vue
/posts          ~/pages/posts.vue
/posts/         ~/pages/posts.vue (parent) + ~/pages/index.vue (child route)
/posts/foo      ~/pages/posts.vue (parent) + ~/pages/_slug.vue (child route)
/posts/foo/     404
```

Go to TOC

Runtime config properties

Runtime config allows passing dynamic config and environment variables to the nuxt context. For more information of usage, please see runtime config guide

---

`publicRuntimeConfig`

- Type: `Object`

Value of this object is **accessible from both client and server** using `$config`.

`privateRuntimeConfig`

- Type: `Object`

Value of this object is accessible from **server only** using `$config`. Overrides `publicRuntimeConfig` for server.

Go to TOC

The server property

Nuxt let you define the server connection variables for your application inside `nuxt.config.js`.

- Type: `Object`

Basic example:

```
export default {
  server: {
    port: 8000, // default: 3000
    host: '0.0.0.0', // default: localhost,
    timing: false
  }
}
```

This lets you specify the host and port for your Nuxt server instance.

Example using HTTPS configuration

```
import path from 'path'
import fs from 'fs'

export default {
  server: {
    https: {
      key: fs.readFileSync(path.resolve(__dirname, 'server.key')),
      cert: fs.readFileSync(path.resolve(__dirname, 'server.crt'))
    }
  }
}
```

You can find additional information on creating server keys and certificates on `localhost` on certificates for localhost article.

Example using sockets configuration

```
export default {
  server: {
    socket: '/tmp/nuxt.socket'
  }
}
```

timing

- Type: `Object` or `Boolean`
- Default: `false`

Enabling the `server.timing` option adds a middleware to measure the time elapsed during server-side rendering and adds it to the headers as 'Server-Timing'

Example using timing configuration

`server.timing` can be an object for providing options. Currently, only `total` is supported (which directly tracks the whole time spent on server-side rendering)

```
export default {
  server: {
    timing: {
      total: true
    }
  }
}
```

Using timing API

The `timing` API is also injected into the `response` on server-side when `server.time` is enabled.

Syntax

```
res.timing.start(name, description)
res.timing.end(name)
```

Example using timing in serverMiddleware

```
export default function (req, res, next) {
  res.timing.start('midd', 'Middleware timing description')
  // server side operation..
  // ...
  res.timing.end('midd')
  next()
}
```

Then `server-timing` head will be included in response header like:

```
Server-Timing: midd;desc="Middleware timing description";dur=2.4
```

Please refer to Server-Timing MDN for more details.

Go to TOC

The serverMiddleware property

Define server-side middleware.

- Type: `Array`
  - Items: `String` or `Object` or `Function`

Nuxt internally creates a connect instance that you can add your own custom middleware to. This allows us to register additional routes (typically `/api` routes) **without need for an external server**.

Because connect itself is a middleware, registered middleware will work with both `nuxt start` and also when used as a middleware with programmatic usages like express-template. Nuxt Modules can also provide `serverMiddleware` using this.addServerMiddleware()

Additional to them, we introduced a `prefix` option which defaults to `true`. It will add the router base to your server middlewares.

Example:

- Server middleware path: `/server-middleware`
- Router base: `/admin`
- With `prefix: true` (default): `/admin/server-middleware`
- With `prefix: false`: `/server-middleware`

serverMiddleware vs middleware!

Don't confuse it with routes middleware which are called before each route by Vue in Client Side or SSR. Middleware listed in the `serverMiddleware` property runs server-side **before** `vue-server-renderer` and can be used for server specific tasks like handling API requests or serving assets.

Warning

Do not add serverMiddleware to the middleware/ directory.

Middleware, are bundled by webpack into your production bundle and run on beforeRouteEnter. If you add serverMiddleware to the middleware/ directory it will be wrongly picked up by Nuxt as middleware and will add wrong dependencies to your bundle or generate errors.

Usage

If middleware is String Nuxt will try to automatically resolve and require it.

```
import serveStatic from 'serve-static'

export default {
  serverMiddleware: [
    // Will register redirect-ssl npm package
```

```
    'redirect-ssl',

    // Will register file from project server-middleware directory to handle
/server-middleware/* requires
    { path: '/server-middleware', handler: '~/server-middleware/index.js' },

    // We can create custom instances too
    { path: '/static2', handler: serveStatic(__dirname + '/static2') }
  ]
}
```

Warning

If you don't want middleware to register for all routes you have to use Object form with specific path, otherwise nuxt default handler won't work!

Custom Server Middleware

It is also possible to write custom middleware. For more information See Connect Docs.

Middleware (`server-middleware/logger.js`):

```
export default function (req, res, next) {
  // req is the Node.js http request object
  console.log(req.url)

  // res is the Node.js http response object

  // next is a function to call to invoke the next middleware
  // Don't forget to call next at the end if your middleware is not an endpoint!
  next()
}
```

```
serverMiddleware: ['~/server-middleware/logger']
```

Custom API endpoint

A server middleware can also extend Express. This allows the creation of REST endpoints.

```
const bodyParser = require('body-parser')
const app = require('express')()

app.use(bodyParser.json())
app.all('/getJSON', (req, res) => {
  res.json({ data: 'data' })
})

module.exports = app
```

```
serverMiddleware: [
  { path: "/server-middleware", handler: "~/server-middleware/rest.js" },
],
```

Object Syntax

If your server middleware consists of a list of functions mapped to paths:

```
export default {
  serverMiddleware: [
    { path: '/a', handler: '~/server-middleware/a.js' },
    { path: '/b', handler: '~/server-middleware/b.js' },
    { path: '/c', handler: '~/server-middleware/c.js' }
  ]
}
```

You can alternatively pass an object to define them, as follows:

```
export default {
  serverMiddleware: {
    '/a': '~/server-middleware/a.js',
    '/b': '~/server-middleware/b.js',
    '/c': '~/server-middleware/c.js'
  }
}
```

Go to TOC

The srcDir property

Define the source directory of your Nuxt application.

- Type: `String`
- Default: rootDir value

If a relative path is specified it will be relative to the `rootDir`.

Example 1: Prerequisites:

```
export default {
  srcDir: 'client/'
}
```

```
  "script": {
    "dev": "yarn nuxt"
  }
```

works with the following folder structure (note that nuxt.config is listed in the app directory)

```
-| app/
---| node_modules/
---| nuxt.config.js
---| package.json
---| client/
| assets/
| components/
| layouts/
| middleware/
| pages/
| plugins/
| static/
| store/
```

Example 2:

Instead of example 1 you can also move the nuxt.config into your `client` folder. In this case you only need to specify `client` as the `rootDir` and you can leave `srcDir` empty:

Prerequisites:

```
export default {
  srcDir: '' // or just remove it
}
```

```
  "script": {
    "dev": "yarn nuxt client" // this sets client as the rootDir
  }
```

works with the following folder structure (note that nuxt.config is listed in the client directory)

```
-| app/
---| node_modules/
---| package.json
---| client/
| nuxt.config.js
| assets/
| components/
| layouts/
| middleware/
| pages/
| plugins/
| static/
| store/
```

Go to TOC

The buildDir property

Define the dist directory for your Nuxt application

- Type: `String`
- Default: `.nuxt`

```
export default {
  buildDir: 'nuxt-dist'
}
```

By default, many tools assume that `.nuxt` is a hidden directory, because its name starts with a dot. You can use this option to prevent that.

Go to TOC

The ssr property

Change default nuxt ssr value

- Type: `boolean`
- Default: `true`
- Possible values:
  - `true`: Server-side rendering enabled
  - `false`: No server-side rendering (only client-side rendering)

> You can set this option to `false` when you want **only client side rendering**

```
export default {
  ssr: false // Disable Server Side rendering
}
```

Next

Previously, `mode` was used to disable or enable server-side rendering. Here is the `mode` documentation.

Go to TOC

The target property

Change default nuxt target

Deployment targets for Nuxt >= v2.13:

- Type: `string`
  - Default: `server`
  - Possible values:
    - `'server'`: For server side rendering
    - `'static'`: For static sites

> You can use this option to change default nuxt target for your project using `nuxt.config.js`

To learn more about the target option check out the deployment targets section.

Go to TOC

The telemetry property

Nuxt collects anonymous telemetry data about general usage. This helps us to accurately gauge Nuxt feature usage and customization across all our users.

The telemetry Property

> Nuxt v2.13.0 introduces Nuxt Telemetry to collect anonymous telemetry data about general usage. This helps us to accurately gauge Nuxt feature usage and customization across all our users.

- Type: `Boolean`
- Default is based on your user preferences

Why collect Telemetry

Nuxt has grown a lot from its initial release (7 Nov 2016) and we are keep listening to community feedback to improve it.

However, this manual process only collects feedback from a subset of users that takes the time to fill the issue template and it may have different needs or use-case than you.

Nuxt Telemetry collects **anonymous telemetry data about general usage**. This helps us to accurately gauge feature usage and customization across all our users. This data will let us better understand how Nuxt is used globally, measuring improvements made (DX and performances) and their relevance.

We collect multiple events:

- Command invoked (nuxt dev, nuxt build, etc)
- Versions of Nuxt and Node.js
- General machine information (MacOS/Linux/Windows and if command is run within CI, the CI name)
- Duration of the Webpack build and average size of the application, as well as the generation stats (when using nuxt generate)
- What are the public dependency of your project (Nuxt modules)

The code is open source and available at https://github.com/nuxt/telemetry.

Opting-out

You can disable Nuxt Telemetry for your project with several ways:

1. Using `npx nuxt telemetry disable`

```
npx nuxt telemetry [status|enable|disable] [-g,--global] [dir]
```

2. Using an environment variable

```
NUXT_TELEMETRY_DISABLED=1
```

3. Setting `telemetry: false` in your `nuxt.config.js`:

```
export default {
  telemetry: false
}
```

You can learn more about Nuxt Telemetry and the events sent on https://github.com/nuxt/telemetry

Go to TOC

Transition properties

Set the default properties of the page and layout transitions.

The pageTransition Property

> Nuxt v2.7.0 introduces key "pageTransition" in favor of the "transition" key to consolidate the naming with layout transition keys.

- Type: `String` or `Object`

> Used to set the default properties of the page transitions.

Default:

```
{
  name: 'page',
  mode: 'out-in'
}
```

```
export default {
  pageTransition: 'page'
  // or
  pageTransition: {
    name: 'page',
    mode: 'out-in',
    beforeEnter (el) {
      console.log('Before enter...');
    }
  }
}
```

The transition key in `nuxt.config.js` is used to set the default properties for the page transitions. To learn more about the available keys when the `transition` key is an object, see the pages transition property.

The layoutTransition Property

- Type: `String` or `Object`

Used to set the default properties of the layout transitions. The value provided in the `name` option is configured to work with the name provided in `layout` from your `layouts` folder.

Default:

```
{
  name: 'layout',
  mode: 'out-in'
}
```

```
export default {
  layoutTransition: 'layout'
  // or
  layoutTransition: {
    name: 'layout',
    mode: 'out-in'
  }
}
```

```
.layout-enter-active,
.layout-leave-active {
  transition: opacity 0.5s;
}
.layout-enter,
.layout-leave-active {
  opacity: 0;
}
```

Go to TOC

The vue.config property

A config object for Vue.config

- Type: `Object`
- Default: `{ silent: !isDev, performance: isDev }`

> The vue.config property provides a direct configuration bridge for the `Vue.config`

Example

```
export default {
  vue: {
    config: {
      productionTip: true,
      devtools: false
    }
  }
}
```

This configuration will lead to the following Vue.config:

```
Vue.config.productionTip // true
Vue.config.devtools // false
Vue.config.silent // !isDev [default value]
Vue.config.performance // isDev [default value]
```

To learn more about the `Vue.config` API, check out the official Vue documentation

---

Go to TOC

The watch property

The watch property lets you watch custom files for restarting the server.

- Type: `Object`
- Default: `[]`

```
watch: ['~/custom/*.js']
```

chokidar is used to set up the watchers. To learn more about chokidar's pattern options, see the chokidar API.

Go to TOC

The watchers property

The watchers property lets you overwrite watchers configuration in your nuxt.config.js.

- Type: `Object`
- Default: `{}`

chokidar

- Type: `Object`
- Default: `{}`

To learn more about chokidar options, see the chokidar API.

webpack

- Type: `Object`
- Default:

```
watchers: {
  webpack: {
    aggregateTimeout: 300,
    poll: 1000
  }
}
```

To learn more about webpack watchOptions, see the webpack documentation.

What's next

Next

Check out the Internals Glossary book

Go to TOC

The cli property

Nuxt lets you customize the CLI configuration.

---

badgeMessages

- Type `Array`

Add a message to the CLI banner.

```
cli: {
  badgeMessages: ['Hello World!']
}
```

bannerColor

- Type: `String`
  - Default: `'green'`

Change the color of the 'Nuxt' title in the CLI banner.

Available colors:

`black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`, `gray`, `redBright`, `greenBright`, `yellowBright`, `blueBright`, `magentaBright`, `cyanBright`, `whiteBright`

```
export default {
  cli: {
    bannerColor: 'yellow'
  }
}
```

Go to TOC

The css property

Nuxt lets you define the CSS files/modules/libraries you want to set globally (included in every page).

In case you want to use `sass` make sure that you have installed `sass` and `sass-loader` packages. If you didn't just

```
yarn add --dev sass sass-loader@10
```

```
npm install --save-dev sass sass-loader@10
```

- Type: `Array`
  - Items: `string`

```
export default {
  css: [
    // Load a Node.js module directly (here it's a Sass file)
    'bulma',
    // CSS file in the project
    '@/assets/css/main.css',
    // SCSS file in the project
    '@/assets/css/main.scss'
  ]
}
```

Nuxt will automatically guess the file type by its extension and use the appropriate pre-processor loader for webpack. You will still need to install the required loader if you need to use them.

Style Extensions

You can omit the file extension for CSS/SCSS/Postcss/Less/Stylus/... files listed in the css array in your nuxt config file.

```
export default {
  css: ['~/assets/css/main', '~/assets/css/animations']
}
```

Warning

If you have two files with the same name e.g. `main.scss` and `main.css`, and don't specify an extension in the css array entry, e.g. `css: ['~/assets/css/main']`, then only one file will be loaded depending on the order of `styleExtensions`. In this case only the `css` file will be loaded and the `scss` file will be ignored because `css` comes first in the default `styleExtension` array.

Default order: `['css', 'pcss', 'postcss', 'styl', 'stylus', 'scss', 'sass', 'less']`

Go to TOC

The components property

Nuxt 2.13+ can scan and auto import your components using @nuxt/components module

---

alert It is possible to use this feature with Nuxt 2.10 - 2.12. Just manually install and add `@nuxt/components` to `buildModules` inside `nuxt.config`.

- Type: `Boolean` or `Array`
- Default: `false`

When set to `true` or an options object, Nuxt will include @nuxt/components and auto-import your components wherever you use them within your pages, layouts (and other components).

Info

For more information on how to use, please refer to component auto-discovery documentation for more information.

Configuration

```
export default {
  // This will automatically load components from `~/components`
  components: true
}
```

With `components: true`, by default the `~/components` directory will be included.

However you can customize auto-discovery behaviour by providing additional directories to scan:

```
export default {
  components: [
    // Equivalent to { path: '~/components' }
    '~/components',
    { path: '~/components/other', extensions: ['vue'] }
  ]
}
```

path

Each item can be either string or object. A string value is a shortcut for `{ path }`.

Info

Don't worry about ordering or overlapping directories! The components module will take care of it. (Each file will be only matched once with longest path.)

path

- Required

- Type: `String`

Path (absolute or relative) to the directory containing your components.

You can use Nuxt aliases (`~` or `@`) to refer to directories inside the project or directly use a npm package path (similar to using `require` within your project).

extensions

- Type: `Array<string>`
- Default:
  - Extensions supported by Nuxt builder (`builder.supportedExtensions`)
  - Default supported extensions `['vue', 'js']` or `['vue', 'js', 'ts', 'tsx']` depending on your environment

Example: Support multi-file component structure

If you prefer to split your SFCs into `.js`, `.vue` and `.css`, you could choose only to scan for `.vue` files:

```
| components
---| componentC
| componentC.vue
| componentC.js
| componentC.scss
```

```
// nuxt.config.js
export default {
  components: [{ path: '~/components', extensions: ['vue'] }]
}
```

pattern

- Type: `string` (glob pattern)
- Default: `**/*.${extensions.join(',')}`

Within the specified `path`, only files that match this pattern will be included.

ignore

- Type: `Array`
- Items: `string` (glob pattern)
- Default: `[]`

Patterns to exclude files within the specified `path`.

prefix

- Type: `String`
- Default: `''` (no prefix)

Prefix all matched components.

The example below adds the `awesome-`/`Awesome` prefix to the name of components in the `awesome/` directory.

```
// nuxt.config.js
export default {
  components: [
    '~/components',
    { path: '~/components/awesome/', prefix: 'awesome' }
  ]
}
```

```
| components/
---| awesome/
| Button.vue
---| Button.vue
```

```
<template>
  <div>
    <AwesomeButton>Click on me 🤘</AwesomeButton>
    <button>Click on me</button>
  </div>
</template>
```

pathPrefix

- Type: `Boolean`
- Default: `true`

Prefix component name by its path.

watch

- Type: `Boolean`
- Default: `true`

Watch the specified `path` for changes, including file additions and file deletions.

transpile

- Type: `Boolean`
- Default: `'auto'`

Transpile specified `path` using `build.transpile`. By default (`'auto'`) it will set `transpile: true` if `node_modules/` is in `path`.

level

- Type: `Number`
- Default: `0`

Levels are used to define allow overwriting components that have the same name in two different directories. This can be useful for library authors who want to allow users to override their components, or for custom themes.

```
export default {
  components: [
    '~/components', // default level is 0
    { path: 'my-theme/components', level: 1 }
  ]
}
```

A component in `~/components` will then overwrite one with the same name in `my-theme/components`. The lowest value takes priority.

Advanced

Overwriting Components

It is possible to have a way to overwrite components using the level option. This is very useful for modules and theme authors.

Considering this structure:

```
| node_modules/
---| my-theme/
| components/
| Header.vue
| components/
---| Header.vue
```

Then defining in the `nuxt.config`:

```
components: [
  '~/components', // default level is 0
  { path: 'node_modules/my-theme/components', level: 1 }
]
```

Our `components/Header.vue` will overwrite our theme component since the lowest level takes priority.

Library Authors

Making Vue Component libraries with automatic tree-shaking and component registration is now super easy ✨

This module exposes a hook named `components:dirs` so you can easily extend the directory list without requiring user configuration in your Nuxt module.

Imagine a directory structure like this:

```
| node_modules/
---| awesome-ui/
| components/
| Alert.vue
| Button.vue
| nuxt.js
| pages/
---| index.vue
| nuxt.config.js
```

Then in `awesome-ui/nuxt.js` you can use the `components:dir` hook:

```
import { join } from 'path'

export default function () {
  this.nuxt.hook('components:dirs', dirs => {
    // Add ./components dir to the list
    dirs.push({
      path: join(__dirname, 'components'),
      prefix: 'awesome'
    })
  })
}
```

That's it! Now in your project, you can import your ui library as a Nuxt module in your `nuxt.config.js`:

```
export default {
  buildModules: ['@nuxt/components', 'awesome-ui/nuxt']
}
```

And directly use the module components (prefixed with `awesome-`), our `pages/index.vue`:

```
<template>
  <div>
    My <AwesomeButton>UI button</AwesomeButton>!
    <awesome-alert>Here's an alert!</awesome-alert>
  </div>
</template>
```

It will automatically import the components only if used and also support HMR when updating your components in `node_modules/awesome-ui/components/`.

Next: publish your `awesome-ui` module to npm and share it with the other Nuxters ✨

Go to TOC

The dev property

Define the development or production mode.

- Type: `Boolean`
- Default: `true`

This property is overwritten by the nuxt commands:

- `dev` is forced to `true` with `nuxt`
- `dev` is forced to `false` with `nuxt build`, `nuxt start` and `nuxt generate`

This property should be used when using Nuxt programmatically:

```
export default {
  dev: process.env.NODE_ENV !== 'production'
}
```

```
const { Nuxt, Builder } = require('nuxt')
const app = require('express')()
const port = process.env.PORT || 3000

// We instantiate Nuxt with the options
const config = require('./nuxt.config.js')
const nuxt = new Nuxt(config)
app.use(nuxt.render)

// Build only in dev mode
if (config.dev) {
  new Builder(nuxt).build()
}

// Listen the server
app.listen(port, '0.0.0.0').then(() => {
  console.log(`Server is listening on port: ${port}`)
})
```

```
{
  "scripts": {
    "dev": "node server.js",
    "build": "nuxt build",
    "start": "NODE_ENV=production node server.js"
  }
}
```

Go to TOC

The dir property

Define the custom directories for your Nuxt application

- Type: `Object`
- Default:

```
{
  assets: 'assets',
  app: 'app',
  layouts: 'layouts',
  middleware: 'middleware',
  pages: 'pages',
  static: 'static',
  store: 'store'
}
```

```
export default {
  dir: {
    assets: 'custom-assets',
    app: 'custom-app',
    layouts: 'custom-layouts',
    middleware: 'custom-middleware',
    pages: 'custom-pages',
    static: 'custom-static',
    store: 'custom-store'
  }
}
```

Go to TOC

The env property

Share environment variables between client and server.

- Type: `Object`

> Nuxt lets you create environment variables client side, also to be shared from server side.

The env property defines environment variables that should be available on the client side. They can be assigned using server side environment variables, the dotenv module ones or similar.

alert For nuxt versions > 2.12+, in cases where environment variables are required at runtime (not build time) it is recommended to subsitute the `env` property with runtimeConfig properties: `publicRuntimeOptions` and `privateRuntimeOptions`.
Learn more with our tutorial about moving from @nuxtjs/dotenv to runtime config.

Make sure to read about `process.env` and `process.env == {}` below for better troubleshooting.

```
export default {
  env: {
    baseUrl: process.env.BASE_URL || 'http://localhost:3000'
  }
}
```

This lets you create a `baseUrl` property that will be equal to the `BASE_URL` server side environment variable if available or defined. If not, `baseUrl` in client side will be equal to `'http://localhost:3000'`. The server side variable BASE_URL is therefore copied to the client side via the `env` property in the `nuxt.config.js`. Alternatively, the other value is defined (http://localhost:3000).

Then, I can access my `baseUrl` variable in 2 ways:

1. Via `process.env.baseUrl`.
2. Via `context.env.baseUrl`, see context API.

You can use the `env` property for giving a public token for example.

For the example above, we can use it to configure axios.

```
import axios from 'axios'

export default axios.create({
  baseURL: process.env.baseUrl
})
```

Then, in your pages, you can import axios like this: `import axios from '~/plugins/axios'`

Automatic injection of environment variables

If you define environment variables starting with `NUXT_ENV_` in the build phase (e.g. `NUXT_ENV_COOL_WORD=freezing nuxt build` or `SET NUXT_ENV_COOL_WORD=freezing & nuxt build` for the Windows console, they'll be automatically injected into the process environment. Be aware that they'll potentially take precedence over defined variables in your `nuxt.config.js` with the same name.

process.env ==

Note that Nuxt uses webpack's `definePlugin` to define the environmental variable. This means that the actual `process` or `process.env` from Node.js is neither available nor defined. Each of the `env` properties defined in nuxt.config.js is individually mapped to `process.env.xxxx` and converted during compilation.

Meaning, `console.log(process.env)` will output `{}` but `console.log(process.env.your_var)` will still output your value. When webpack compiles your code, it replaces all instances of `process.env.your_var` with the value you've set it to, e.g.: `env.test = 'testing123'`. If you use `process.env.test` in your code somewhere, it is actually translated to 'testing123'.

before

```
if (process.env.test == 'testing123')
```

after

```
if ('testing123' == 'testing123')
```

serverMiddleware

As serverMiddleware is decoupled from the main Nuxt build, `env` variables defined in `nuxt.config.js` are not available there.

Go to TOC

The Context

The `context` provides additional objects/params from Nuxt to Vue components and is available in special nuxt lifecycle areas like `asyncData`, `fetch`, `plugins`, `middleware` and `nuxtServerInit`.

> *Note: "The Context" we refer to here is not to be confused with the `context` object available in `Vuex Actions`. The two are unrelated.*

```
function (context) {
  // Universal keys
  const {
    app,
    store,
    route,
    params,
    query,
    env,
    isDev,
    isHMR,
    redirect,
    error,
    $config
  } = context
  // Server-side
  if (process.server) {
    const { req, res, beforeNuxtRender } = context
  }
  // Client-side
  if (process.client) {
    const { from, nuxtState } = context
  }
}
```

Universal keys

These keys are available both on client-side and server-side.

app

`app` (*NuxtAppOptions*)

The root Vue instance options that includes all your plugins. For example, when using `i18n`, you can get access to `$i18n` through `context.app.i18n`.

store

`store` (*Vuex Store*)

Vuex Store instance. **Available only if the vuex store is set**.

route

`route` (*Vue Router Route*)

Vue Router route instance.

params

`params` (*Object*)

Alias of `route.params`.

query

`query` (*Object*)

Alias of `route.query`.

env

`env` (*Object*)

Environment variables set in `nuxt.config.js`, see env API.

isDev

`isDev` (*Boolean*)

Boolean to let you know if you're in dev mode, can be useful for caching some data in production.

isHMR

`isHMR` (*Boolean*)

Boolean to let you know if the method/middleware is called from webpack hot module replacement (*true only on client-side in dev mode*).

redirect

`redirect` (*Function*)

Use this method to redirect the user to another route, the status code is used on the server-side, defaults to `302`. `redirect([status,] path [, query])`.

Examples:

```
redirect(302, '/login')
redirect({ name: 'slug', params: { slug: mySlug } })
redirect('https://vuejs.org')
```

See the Vue Router docs more info on the Location property.

Info

It's not possible to use `redirect` or `error` in client-side Nuxt plugin due to hydration errors (client content would be different from what it'd expect from the server).

A valid workaround would be using `window.onNuxtReady(() => { window.$nuxt.$router.push('/your-route') })`

error

`error` (*Function*)

Use this method to show the error page: `error(params)`. The `params` should have the properties `statusCode` and `message`.

`$config`

`$config` (*Object*)

The actual runtime config.

Server-side keys

These keys are available only on the server-side.

req

`req` (*http.Request*)

Request from the Node.js server. If Nuxt is used as a middleware, the request object might be different depending on the framework you're using.
Not available via `nuxt generate`.

Res

`res` (*http.Response*)

Response from the Node.js server. If Nuxt is used as a middleware, the res object might be different depending on the framework you're using.
Not available via `nuxt generate`.

beforeNuxtRender

`beforeNuxtRender(fn)` (*Function*)

Use this method to update `__NUXT__` variable rendered on client-side, the `fn` (can be asynchronous) is called with `{ Components, nuxtState }`, see example.

Client-side keys

These keys are available only on client-side.

from

`from` (*Vue Router Route*)

The route navigated from.

nuxtState

`nuxtState` *(Object)*

Nuxt state, useful for plugins which uses `beforeNuxtRender` to get the nuxt state on client-side before hydration. **Available only in `universal` mode**.

Go to TOC

nuxt.render(req, res)

You can use Nuxt as a middleware with `nuxt.render` for your Node.js server.

- Type: `Function`
- Arguments:
  - Request
  - Response
- Returns: `Promise`

Example with Express:

```
const { loadNuxt, build } = require('nuxt')

const app = require('express')()
const isDev = process.env.NODE_ENV !== 'production'
const port = process.env.PORT || 3000

async function start() {
  // We get Nuxt instance
  const nuxt = await loadNuxt(isDev ? 'dev' : 'start')

  // Render every route with Nuxt
  app.use(nuxt.render)

  // Build only in dev mode with hot-reloading
  if (isDev) {
    build(nuxt)
  }
  // Listen the server
  app.listen(port, '0.0.0.0')
  console.log('Server listening on `localhost:' + port + '`.')
}

start()
```

Warning

It's recommended to call `nuxt.render` at the end of your middlewares since it will handle the rendering of your web application and won't call `next()`

Go to TOC

nuxt.renderRoute(route, context)

Render a specific route with a given context.

- Type: `Function`
- Arguments:
  1. `String` : route to render
  2. *Optional*, `Object`, context given, available keys: `req` & `res`
- Returns: `Promise`
  - `html`: `String`
  - `error`: `null` or `Object`
  - `redirected`: `false` or `Object`

This method should be used mostly for test purposes as well as with `nuxt.renderAndGetWindow`.

Warning

`nuxt.renderRoute` should be executed after the build process in production mode.

```
const { loadNuxt, build } = require('nuxt')

async function start() {
  // Get nuxt instance for start (production mode)
  // Make sure to have run `nuxt build` before running this script
  const nuxt = await loadNuxt({ for: 'start' })

  const { html, error, redirected } = await nuxt.renderRoute('/')

  // `html` will always be a string

  // `error` not null when the error layout is displayed, the error format is:
  // { statusCode: 500, message: 'My error message' }

  // `redirected` is not `false` when `redirect()` has been used in `asyncData()`
or `fetch()`
  // { path: '/other-path', query: {}, status: 302 }
}

start()
```

Go to TOC

nuxt.renderAndGetWindow(url, options)

Get the `window` from a given URL of a Nuxt Application.

- Type: `Function`
- Argument: `String`
  1. `String`: URL to render
  2. *Optional*, `Object`: options
     - virtualConsole: `Boolean` (default: `true`)
- Returns: `Promise`
  - Returns: `window`

Warning

This method is made for test purposes.

To use this function, you have to install `jsdom`:

```
npm install --save-dev jsdom
```

Example:

```
const { loadNuxt } = require('nuxt')

async function init() {
  // Assuming you've already built your project
  const nuxt = await loadNuxt({ for: 'start' })
  await nuxt.listen(3000)
  const window = await nuxt.renderAndGetWindow('http://localhost:3000')
  // Display the head `<title>`
  console.log(window.document.title)
  nuxt.close()
}

init()
```

What's next

Next

Check out the the Components Glossary book

Go to TOC

$nuxt: The Nuxt helper

$nuxt is a helper designed to improve user experience.

For more info on the Nuxt helper check out the context and helpers chapter in the Concepts book

Connection checker

- `isOffline`
  - Type: `Boolean`
  - Description: `true` when the user's internet connection becomes offline
- `isOnline`
  - Type: `Boolean`
  - Description: Opposite of `isOffline`

```
<template>
  <div>
    <div v-if="$nuxt.isOffline">You are offline</div>
    <nuxt />
  </div>
</template>
```

Refreshing page data

- `refresh()`
  - When you want to only refresh the data provided by asyncData or fetch

```
<template>
  <div>
    <div>{{ content }}</div>
    <button @click="refresh">Refresh</button>
  </div>
</template>

<script>
  export default {
    asyncData() {
      return { content: 'Created at: ' + new Date() }
    },
    methods: {
      refresh() {
        this.$nuxt.refresh()
      }
    }
  }
</script>
```

Controlling the loading bar

- `$loading`
  - When you want to control Nuxt's loading bar programmatically

```
export default {
  mounted() {
    this.$nextTick(() => {
      this.$nuxt.$loading.start()
      setTimeout(() => this.$nuxt.$loading.finish(), 500)
    })
  }
}
```

Go to TOC

Nuxt modules intro

Better understand Nuxt internals

Nuxt has a fully modular architecture which allows developers extending any part of Nuxt Core using a flexible API.

Please see Modules Guide for more detailed information if interested developing your own module.

This section helps getting familiar to Nuxt internals and can be used as a reference to understand it better while writing your own modules.

Core

These classes are the heart of Nuxt and should exist on both runtime and build time.

Nuxt

- `Nuxt` Class
- Source: core/nuxt.js

Renderer

- `Renderer` Class
- Source: vue-renderer/renderer.js

ModuleContainer

- `ModuleContainer` Class
- Source: core/module.js

Build

These classes are only needed for build or dev mode.

Builder

- `Builder` Class
- Source: builder/builder.js

Generator

- `Generator` Class
- Source: generator/generator.js

Common

Utils

- Source: utils/src

Options

- Source: config/options.js

Packaging & Usage

Nuxt exports all classes by default. To import them:

```
import { Nuxt, Builder, Utils } from 'nuxt'
```

Common patterns

All Nuxt classes have a reference to `nuxt` instance and options, this way we always have a consistent API across classes to access `options` and `nuxt`.

```
class SomeClass {
  constructor(nuxt) {
    super()
    this.nuxt = nuxt
    this.options = nuxt.options
  }

  someFunction() {
    // We have access to `this.nuxt` and `this.options`
  }
}
```

Classes are *pluggable* so they should register a plugin on main `nuxt` container to register more hooks.

```
class FooClass {
  constructor(nuxt) {
    super()
    this.nuxt = nuxt
    this.options = nuxt.options

    this.nuxt.callHook('foo', this)
  }
}
```

So we can hook into `foo` module like this:

```
nuxt.hook('foo', foo => {
  // ...
})
```

Go to TOC

The Nuxt class

This is the core container which allows all modules and classes communicate with each other. All modules have access to Nuxt instance using `this.nuxt`.

- Source: **core/nuxt.js**

Hooks

We can register hooks on certain life cycle events.

```
nuxt.hook('ready', async nuxt => {
  // Your custom code here
})
```

| Plugin | Arguments | When | | | `ready` | (nuxt) | Nuxt is ready to work (`ModuleContainer` and `Renderer` ready). | | `error` | (error) | An unhandled error when calling hooks. | | `close` | (nuxt) | Nuxt instance is gracefully closing. | | `listen` | (server, {host, port}) | Nuxt **internal** server starts listening. (Using `nuxt start` or `nuxt dev`). |

Go to TOC

The renderer class

This class is exporting a connect middleware which handles and serves all SSR and asset requests.

- Source: **vue-renderer/renderer.js**

Hooks

We can register hooks on certain life cycle events.

| Hook | Arguments | When | | | `render:before` | (renderer, options) | Before setting up middleware and resources for the Renderer class, useful to overload some methods or options. | | `render:setupMiddleware` | (app) *connect instance* | Before Nuxt adds its middleware stack. We can use it to register custom server side middleware. | | `render:errorMiddleware` | (app) *connect instance* | Before adding Nuxt error middleware, useful to add your own middleware before using Nuxt's. See the Sentry module for more info. | | `render:resourcesLoaded` | (resources) | Called after resources for renderer are loaded (client manifest, server bundle, etc). | | `render:done` | (renderer) | SSR Middleware and all resources are ready (Renderer ready) | | `render:routeContext` | (context.nuxt) | *Every time a route is server-rendered and before* `render:route` *hook*. Called before serializing Nuxt context into `window.__NUXT__`, useful to add some data that you can fetch on client-side. | | `render:route` | (url, result, context) | *Every time a route is server-rendered*. Called before sending back the request to the browser. | | `render:routeDone` | (url, result, context) | *Every time a route is server-rendered*. Called after the response has been sent to the browser. |

Go to TOC

The ModuleContainer class

- Source: **core/module.js**

All modules will be called within context of `ModuleContainer` instance.

Tapable plugins

We can register hooks on certain life cycle events.

```
nuxt.moduleContainer.plugin('ready', async moduleContainer => {
  // Do this after all modules where ready
})
```

Inside modules context we can use this instead:

```
this.plugin('ready', async moduleContainer => {
  // Do this after all modules where ready
})
```

| Plugin | Arguments | When | | | `ready` | moduleContainer | All modules in `nuxt.config.js` has been initialized |

Methods

addVendor (vendor)

Deprecated as `vendor` isn't used anymore

Adds to `options.build.vendor` and apply unique filter.

addTemplate (template)

- **template**: `String` or `Object`
  - `src`
  - `options`
  - `fileName`

Renders given template using lodash template during build into project `buildDir` (`.nuxt`).

If `fileName` is not provided or `template` is string, target file name defaults to `[dirName].[fileName].[pathHash].[ext]`.

This method returns final `{ dst, src, options }` object.

addPlugin (template)

- **template**: Object properties (`src`, `options`, `fileName`, `mode`).

Registers a plugin using `addTemplate` and prepends it it to `plugins[]` array.

```
this.addPlugin({
  src: path.resolve(__dirname, 'templates/foo.js'),
  fileName: 'foo.server.js' // [optional] only include in server bundle
  options: moduleOptions
})
```

Note: You can use `mode` or `.client` and `.server` modifiers with `fileName` option to use plugin only in client or server side. (See plugins for all available options)

If you choose to specify a `fileName`, you can configure a custom path for the `fileName` too, so you can choose the folder structure inside `.nuxt` folder in order to prevent name collisioning:

```
{
  fileName: path.join('folder', 'foo.client.js'), // will result in
`.nuxt/folder/foo.client.js`
}
```

addServerMiddleware (middleware)

Pushes middleware into options.serverMiddleware.

extendBuild (fn)

Allows easily extending webpack build config by chaining options.build.extend function.

extendRoutes (fn)

Allows easily extending routes by chaining options.build.extendRoutes function.

addModule (moduleOpts, requireOnce)

Async function

Registers a module. `moduleOpts` can be a string or an array (`[src, options]`). If `requireOnce` is `true` and the resolved module exports `meta`, it prevents registering same module twice.

requireModule (moduleOpts)

Async function

Is a shortcut for `addModule(moduleOpts, true)`

Hooks

We can register hooks on certain life cycle events.

Hook	Arguments	When
`modules:before`	(moduleContainer, options)	Called before creating ModuleContainer class, useful to overload methods and options.
`modules:done`	(moduleContainer)	Called when all modules have been loaded.

Go to TOC

The Builder class

- Source: **builder/builder.js**

Hooks

We can register hooks on certain life cycle events.

```
// Add hook for build
this.nuxt.hook('build:done', (builder) => {
  ...
})
```

| Hook | Arguments | When | | | `build:before` | (nuxt, buildOptions) | Before Nuxt build started | | `builder:prepared` | (nuxt, buildOptions) | The build directories have been created | | `builder:extendPlugins` | (plugins) | Generating plugins | | `build:templates` | ({ templatesFiles, templateVars, resolve }) | Generating `.nuxt` template files | | `build:extendRoutes` | (routes, resolve) | Generating routes | | `webpack:config` | (webpackConfigs) | Before configuration of compilers | | `build:compile` | ({ name, compiler }) | Before webpack compile (compiler is a webpack `Compiler` instance), if universal mode, called twice with name `'client'` and `'server'` | | `build:compiled` | ({ name, compiler, stats }) | webpack build finished | | `build:done` | (nuxt) | Nuxt build finished |

Go to TOC

The Generator class

- Source: **generator/generator.js**

Hooks

`generate:` hooks:

Hook	Arguments	When
`generate:before`	(generator, generateOptions)	Hook on before generation
`generate:distRemoved`	(generator)	Hook on destination folder cleaned
`generate:distCopied`	(generator)	Hook on copy static and built files
`generate:route`	({ route, setPayload })	Hook before generating the page, useful for dynamic payload, see #7422, available for Nuxt 2.13+
`generate:page`	({ route, path, html })	Hook to let user update the path & html after generation
`generate:routeCreated`	({ route, path, errors })	Hook on saving generated page success
`generate:extendRoutes`	(routes)	Hook to let user update the routes to generate
`generate:routeFailed`	({ route, errors })	Hook on saving generated page failure
`generate:done`	(generator, errors)	Hook on generation finished

Go to TOC

Using Nuxt programatically

You can use Nuxt programmatically to use it as a middleware giving you the freedom of creating your own server for rendering your web applications.

You might want to use your own server with your middleware and your API. That's why you can use Nuxt programmatically.

Nuxt Constructor

To see the list of options to give to Nuxt, see the configuration section.

```
const { loadNuxt, build } = require('nuxt')

// Check if we need to run Nuxt in development mode
const isDev = process.env.NODE_ENV !== 'production'

// Get a ready to use Nuxt instance
const nuxt = await loadNuxt(isDev ? 'dev' : 'start')

// Enable live build & reloading on dev
if (isDev) {
  build(nuxt)
}

// We can use `nuxt.render(req, res)` or `nuxt.renderRoute(route, context)`
```

You can take a look at the nuxt-express and adonuxt starters to get started quickly.

Debug logs

If you want to display Nuxt logs, you can add the following to the top of your file:

```
process.env.DEBUG = 'nuxt:*'
```

Go to TOC

The Fetch hook

The `fetch` hook is for fetching data asynchronously. It is called on server-side when rendering the route, and on client-side when navigating.

Nuxt >= 2.12

Nuxt `v2.12` introduces a new hook called `fetch` which you can use **in any of your Vue components**. Use fetch every time you need to get **asynchronous** data. `fetch` is called on server-side when rendering the route, and on client-side when navigating.

It exposes `$fetchState` at the component level:

- `$fetchState.pending`: `Boolean`, allows you to display a placeholder when `fetch` is being called *on client-side*.
- `$fetchState.error`: `null` or `Error`, allows you to display an error message
- `$fetchState.timestamp`: `Integer`, is a timestamp of the last fetch, useful for caching with `keep-alive`

If you want to call the `fetch` hook from your template use:

```
<button @click="$fetch">Refresh</button>
```

or component method:

```
// from component methods in script section
export default {
  methods: {
    refresh() {
      this.$fetch()
    }
  }
}
```

You can access the Nuxt context within the fetch hook using `this.$nuxt.context`.

Options

- `fetchOnServer`: `Boolean` or `Function` (default: `true`), call `fetch()` when server-rendering the page
- `fetchKey`: `String` or `Function` (defaults to the component scope ID or component name), a key (or a function that produces a unique key) that identifies the result of this component's fetch (available on Nuxt 2.15+) More information available in original PR.
- `fetchDelay`: `Integer` (default: `200`), set the minimum executing time in milliseconds (to avoid quick flashes)

When `fetchOnServer` is falsy (`false` or returns `false`), `fetch` will be called only on client-side and `$fetchState.pending` will return `true` when server-rendering the component.

```
<script>
  export default {
    data() {
      return {
        posts: []
      }
    },
    async fetch() {
      this.posts = await this.$http.$get('https://api.nuxtjs.dev/posts')
    },
    fetchOnServer: false,
    // multiple components can return the same `fetchKey` and Nuxt will track them
both separately
    fetchKey: 'site-sidebar',
    // alternatively, for more control, a function can be passed with access to
the component instance
    // It will be called in `created` and must not depend on fetched data
    fetchKey(getCounter) {
      // getCounter is a method that can be called to get the next number in a
sequence
      // as part of generating a unique fetchKey.
      return this.someOtherData + getCounter('sidebar')
    }
  }
</script>
```

Next

For more info on the Fetch Hook checkout the data fetching chapter of our Features book

Go to TOC

The watchQuery property

Watch query strings and execute component methods on change (asyncData, fetch, validate, layout, ...)

- **Type:** `Boolean` or `Array` or `Function` (default: `[]`)

Use the `watchQuery` key to set up a watcher for query strings. If the defined strings change, all component methods (asyncData, fetch(context), validate, layout, ...) will be called. Watching is disabled by default to improve performance.

If you want to set up a watcher for all query strings, set `watchQuery: true`.

```
export default {
  watchQuery: ['page']
}
```

You can also use the function `watchQuery(newQuery, oldQuery)` to have more refined watchers.

```
export default {
  watchQuery(newQuery, oldQuery) {
    // Only execute component methods if the old query string contained `bar`
    // and the new query string contains `foo`
    return newQuery.foo && oldQuery.bar
  }
}
```

Warning

Warning: The new `fetch` hook introduced in 2.12 is not affected by `watchQuery`. For more information see listening to query string changes.

Go to TOC

The Head method

Nuxt uses vue-meta to update the `headers` and `html attributes` of your application.

- **Type:** `Object` or `Function`

Use the `head` method to set the HTML Head tags for the current page.

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

<script>
  export default {
    data() {
      return {
        title: 'Hello World!'
      }
    },
    head() {
      return {
        title: this.title,
        meta: [
          // hid is used as unique identifier. Do not use `vmid` for it as it will
not work
          {
            hid: 'description',
            name: 'description',
            content: 'My custom description'
          }
        ]
      }
    }
  }
</script>
```

Info

To avoid duplicated meta tags when used in child component, set up a unique identifier with the `hid` key for your meta elements (read more).

Go to TOC

The key property

Set the `key` property of internal `<router-view>` component

- **Type:** `String` or `Function`

The `key` property is propagated into `<router-view>`, which is useful to make transitions inside a dynamic page and different route. Different keys result in re-rendering of page components.

There are several ways to set the key. For more details, please refer to the `nuxtChildKey` prop in the nuxt component.

```
export default {
  key(route) {
    return route.fullPath
  }
}
```

Go to TOC

The layout property

Every file (first level) in the layouts directory will create a custom layout accessible with the layout property in the page component.

- **Type:** `String` or `Function` (default: `'default'`)

Use the `layout` key in your pages components to define which layout to use:

```
export default {
  layout: 'blog',
  // OR
  layout(context) {
    return 'blog'
  }
}
```

Go to TOC

The loading property

The loading property gives you the option to disable the default loading progress bar on a specific page.

- **Type:** `Boolean` (default: `true`)

By default, Nuxt uses its own component to show a progress bar between the routes.

You can disable or customize it globally through the Configuration's loading option, but also disable it for specific pages by setting the `loading` property to `false` :

```
<template>
  <h1>My page</h1>
</template>

<script>
  export default {
    loading: false
  }
</script>
```

Go to TOC

The middleware property

Set the middleware for a specific page of the application.

- Type: `String` or `Array` or `Function`
  - Items: `String` or `Function`

Named middleware

You can create named middleware by creating a file inside the `middleware/` directory, the file name will be the middleware name.

```
export default function ({ store, redirect }) {
  // If the user is not authenticated
  if (!store.state.authenticated) {
    return redirect('/login')
  }
}
```

```
<template>
  <h1>Secret page</h1>
</template>

<script>
  export default {
    middleware: 'authenticated'
  }
</script>
```

Anonymous middleware

If you need to use a middleware only for a specific page, you can directly use a function for it (or an array of functions):

```
<template>
  <h1>Secret page</h1>
</template>

<script>
  export default {
    middleware({ store, redirect }) {
      // If the user is not authenticated
      if (!store.state.authenticated) {
        return redirect('/login')
      }
    }
  }
</script>
```

Go to TOC

The page transition property

Nuxt uses the `<transition>` component to let you create amazing transitions/animations between your pages.

- **Type:** `String` or `Object` or `Function`

To define a custom transition for a specific route, simply add the `transition` key to the page component.

```
export default {
  // Can be a String
  transition: ''
  // Or an Object
  transition: {}
  // or a Function
  transition (to, from) {}
}
```

String

If the `transition` key is set as a string, it will be used as the `transition.name`.

```
export default {
  transition: 'test'
}
```

Nuxt will use these settings to set the component as follows:

```
<transition name="test"></transition>
```

Object

If the `transition` key is set as an object:

```
export default {
  transition: {
    name: 'test',
    mode: 'out-in'
  }
}
```

Nuxt will use these settings to set the component as follows:

```
<transition name="test" mode="out-in"></transition>
```

The `transition` object can have the following properties:

key	Type	Default	definition
`name`	`String`	`"page"`	The transition name applied on all the route transitions.
`mode`	`String`	`"out-in"`	The transition mode applied on all routes, see Vue.js documentation.
`css`	`Boolean`	`true`	Whether to apply CSS transition classes. Defaults to `true`. If

set to `false`, will only trigger JavaScript hooks registered via component events. | | `duration` | `Integer` | n/a | The duration (in milliseconds) applied on the transition, see Vue.js documentation. | | `type` | `String` | n/a | Specify the type of transition events to wait for to determine transition end timing. Available values are `"transition"` and `"animation"`. By default, it will automatically detect the type that has a longer duration. | | `enterClass` | `String` | n/a | The starting state of the transition class. See Vue.js documentation. | | `enterToClass` | `String` | n/a | The ending state for the transition. See Vue.js documentation. | | `enterActiveClass` | `String` | n/a | The class applied across the entire transition duration. See Vue.js documentation. | | `leaveClass` | `String` | n/a | The starting state of the transition class. See Vue.js documentation. | | `leaveToClass` | `String` | n/a | The ending state for the transition. See Vue.js documentation. | | `leaveActiveClass` | `String` | n/a | The class applied across the entire transition duration. See Vue.js documentation. |

You can also define methods in the page `transition` property, these are for the JavaScript hooks:

- `beforeEnter(el)`
- `enter(el, done)`
- `afterEnter(el)`
- `enterCancelled(el)`
- `beforeLeave(el)`
- `leave(el, done)`
- `afterLeave(el)`
- `leaveCancelled(el)`

```
export default {
  transition: {
    afterLeave(el) {
      console.log('afterLeave', el)
    }
  }
}
```

Note: it's also a good idea to explicitly add `css: false` for JavaScript-only transitions so that Vue can skip the CSS detection. This also prevents CSS rules from accidentally interfering with the transition.

Transition Mode

The default transition mode for pages differs from the default mode in Vue.js. The `transition` mode is by default set to `out-in`. If you want to run leaving and entering transitions simultaneously, you have to set the mode to the empty string `mode: ''`.

```
export default {
  transition: {
    name: 'test',
    mode: ''
  }
}
```

Function

If the `transition` key is set as a function:

```
export default {
  transition(to, from) {
    if (!from) {
      return 'slide-left'
    }
    return +to.query.page < +from.query.page ? 'slide-right' : 'slide-left'
  }
}
```

Transitions applied on navigation:

- `/` to `/posts` => `slide-left`,
- `/posts` to `/posts?page=3` => `slide-left`,
- `/posts?page=3` to `/posts?page=2` => `slide-right`.

Go to TOC

The scrollToTop property

The `scrollToTop` property lets you tell Nuxt to scroll to the top before rendering the page.

- **Type:** `Boolean` (default: `false`)

By default, Nuxt scrolls to the top when you go to another page, but with children routes, Nuxt keeps the scroll position. If you want to tell Nuxt to scroll to the top when rendering your child route, set `scrollToTop` to `true`:

```
<template>
  <h1>My child component</h1>
</template>

<script>
  export default {
    scrollToTop: true
  }
</script>
```

Conversely, you can manually set `scrollToTop` to `false` on parent routes as well.

If you want to overwrite the default scroll behavior of Nuxt, take a look at the scrollBehavior option.

Go to TOC

The validate method

Nuxt lets you define a validator method inside your dynamic route component.

- **Type:** `Function` or `Async Function`

`validate` is called every time before navigating to a new route. It will be called server-side once (on the first request to the Nuxt app) and client-side when navigating to further routes. This method takes the `context` object as an argument.

```
validate({ params, query, store }) {
  return true // if the params are valid
  return false // will stop Nuxt to render the route and display the error page
}
```

```
async validate({ params, query, store }) {
  // await operations
  return true // if the params are valid
  return false // will stop Nuxt to render the route and display the error page
}
```

You can also return promises:

```
validate({ params, query, store }) {
  return new Promise((resolve) => setTimeout(() => resolve()))
}
```

Nuxt lets you define a validator method inside your dynamic route component (In this example: `pages/users/_id.vue`).

If the validate method does not return `true`, Nuxt will automatically load the 404 error page.

```
export default {
  validate({ params }) {
    // Must be a number
    return /^\d+$/.test(params.id)
  }
}
```

You can also check some data in your store for example (filled by `nuxtServerInit` before action):

```
export default {
  validate({ params, store }) {
    // Check if `params.id` is an existing category
    return store.state.categories.some(category => category.id === params.id)
  }
}
```

You can also throw expected or unexpected errors during validate function execution:

```
export default {
  async validate({ params, store }) {
    // Throws a 500 internal server error with custom message
    throw new Error('Under Construction!')
  }
}
```

Go to TOC

Colophon

This book is created by using the following sources:

- Nuxt - Español
- GitHub source: nuxt/nuxtjs.org
- Created: 2022-12-10
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf