Module Author Guide

Learn how to create a Nuxt Module to integrate, enhance or extend any Nuxt applications.

Nuxt's configuration and hooks systems make it possible to customize every aspect of Nuxt and add any integration you might need (Vue plugins, CMS, server routes, components, logging, etc.).

Nuxt Modules are functions that sequentially run when starting Nuxt in development mode using `nuxi dev` or building a project for production with `nuxi build`. With modules, you can encapsulate, properly test, and share custom solutions as npm packages without adding unnecessary boilerplate to your project, or requiring changes to Nuxt itself.

Quick Start

We recommend you get started with Nuxt Modules using our starter template:

```
> npx nuxi init -t module my-module
```

This will create a `my-module` project with all the boilerplate necessary to develop and publish your module.

Next steps:

1. Open `my-module` in your IDE of choice
2. Install dependencies using your favorite package manager
3. Prepare local files for development using `npm run dev:prepare`
4. Follow this document to learn more about Nuxt Modules

Using the Starter

Learn how to perform basic tasks with the module starter.

How to Develop

While your module source code lives inside the `src` directory, in most cases, to develop a module, you need a Nuxt application. That's what the `playground` directory is about. It's a Nuxt application you can tinker with that is already configured to run with your module.

You can interact with the playground like with any Nuxt application.

- Launch its development server with `npm run dev`, it should reload itself as you make changes to your module in the `src` directory
- Build it with `npm run dev:build`

> All other `nuxi` commands can be used against the `playground` directory (e.g. `nuxi <COMMAND> playground`). Feel free to declare additional `dev:*` scripts within your `package.json` referencing them for convenience.

How to Test

The module starter comes with a basic test suite:

- A linter powered by ESLint, run it with `npm run lint`
- A test runner powered by Vitest, run it with `npm run test` or `npm run test:watch`

> Feel free to augment this default test strategy to better suit your needs.

How to Build

Nuxt Modules come with their own builder provided by `@nuxt/module-builder`. This builder doesn't require any configuration on your end, supports TypeScript, and makes sure your assets are properly bundled to be distributed to other Nuxt applications.

You can build your module by running `npm run prepack`.

> While building your module can be useful in some cases, most of the time you won't need to build it on your own: the `playground` takes care of it while developing, and the release script also has you covered when publishing.

How to Publish

> Before publishing your module to npm, makes sure you have an npmjs.com account and that you're authenticated to it locally with `npm login`.

While you can publish your module by bumping its version and using the `npm publish` command, the module starter comes with a release script that helps you make sure you publish a working version of your module to npm and more.

To use the release script, first, commit all your changes (we recommend you follow Conventional Commits to also take advantage of automatic version bump and changelog update), then run the release script with `npm run release`.

When running the release script, the following will happen:

- First, it will run your test suite by:
  - Running the linter (`npm run lint`)
  - Running unit, integration, and e2e tests (`npm run test`)
  - Building the module (`npm run prepack`)
- Then, if your test suite went well, it will proceed to publish your module by:
  - Bumping your module version and generating a changelog according to your Conventional Commits
  - Publishing the module to npm (for that purpose, the module will be built again to ensure its updated version number is taken into account in the published artifact)
  - Pushing a git tag representing the newly published version to your git remote origin

> As with other scripts, feel free to fine-tune the default `release` script in your `package.json` to better suit your need.

Developing Modules

Nuxt Modules come with a variety of powerful APIs and patterns allowing them to alter a Nuxt application in pretty much any way possible. This section teaches you how to take advantage of those.

Module Anatomy

We can consider two kinds of Nuxt Modules:

- published modules are distributed on npm - you can see a list of some community modules on the Nuxt website.
- "local" modules, they exist within a Nuxt project itself, either inlined in Nuxt config or as part of the `modules` directory.

In either case, their anatomy is similar.

Module Definition

> When using the starter, your module definition is available at `src/module.ts`.

The module definition is the entry point of your module. It's what gets loaded by Nuxt when your module is referenced within a Nuxt configuration.

At a low level, a Nuxt Module definition is a simple, potentially asynchronous, function accepting inline user options and a `nuxt` object to interact with Nuxt.

```
export default function (inlineOptions, nuxt) {
  // You can do whatever you like here..
  console.log(inlineOptions.token) // `123`
  console.log(nuxt.options.dev) // `true` or `false`
  nuxt.hook('ready', async nuxt => {
    console.log('Nuxt is ready')
  })
}
```

You can get type-hint support for this function using the higher-level `defineNuxtModule` helper provided by Nuxt Kit.

```ts
import { defineNuxtModule } from '@nuxt/kit'

export default defineNuxtModule((options, nuxt) => {
  nuxt.hook('pages:extend', pages => {
    console.log(`Discovered ${pages.length} pages`)
  })
})
```

However, **we do not recommend** using this low-level function definition. Instead, to define a module, **we recommend** using the object-syntax with `meta` property to identify your module, especially when publishing to npm.

This helper makes writing Nuxt Module more straightforward by implementing many common patterns seen in modules, guaranteeing future compatibility, and improving your module author developer experience and the one of your module users.

```ts
import { defineNuxtModule } from '@nuxt/kit'

export default defineNuxtModule({
  meta: {
    // Usually the npm package name of your module
    name: '@nuxtjs/example',
    // The key in `nuxt.config` that holds your module options
    configKey: 'sample',
    // Compatibility constraints
    compatibility: {
      // Semver version of supported nuxt versions
      nuxt: '^3.0.0'
    }
  },
  // Default configuration options for your module, can also be a function returning those
  defaults: {},
  // Shorthand sugar to register Nuxt hooks
  hooks: {},
  // The function holding your module logic, it can be asynchronous
  setup(moduleOptions, nuxt) {
    // ...
  }
})
```

Ultimately `defineNuxtModule` returns a wrapper function with the lower level `(inlineOptions, nuxt)` module signature. This wrapper function applies defaults and other necessary steps before calling your `setup` function:

- ✓ Support `defaults` and `meta.configKey` for automatically merging module options
- ✓ Type hints and automated type inference
- ✓ Add shims for basic Nuxt 2 compatibility
- ✓ Ensure module gets installed only once using a unique key computed from `meta.name` or `meta.configKey`
- ✓ Automatically register Nuxt hooks
- ✓ Automatically check for compatibility issues based on module meta
- ✓ Expose `getOptions` and `getMeta` for internal usage of Nuxt
- ✓ Ensuring backward and upward compatibility as long as the module is using `defineNuxtModule` from the latest version of `@nuxt/kit`
- ✓ Integration with module builder tooling

Runtime Directory

> When using the starter, the runtime directory is available at `src/runtime`.

Modules, like everything in a Nuxt configuration, aren't included in your application runtime. However, you might want your module to provide, or inject runtime code to the application it's installed on. That's what the runtime directory enables you to do.

Inside the runtime directory, you can provide any kind of assets related to the Nuxt App:

- Vue components
- Composables
- Nuxt plugins

To the server engine, Nitro:

- API routes
- Middlewares
- Nitro plugins

Or any other kind of asset you want to inject in users' Nuxt applications:

- Stylesheets

- 3D models
- Images
- etc.

You'll then be able to inject all those assets inside the application from your module definition.

> Learn more about asset injection in the recipes section.

> Published modules cannot leverage auto-imports for assets within their runtime directory. Instead, they have to import them explicitly from `#imports` or alike.
>
> Indeed, auto-imports are not enabled for files within `node_modules` (the location where a published module will eventually live) for performance reasons. That's why the module starter deliberately disables them while developing a module.
>
> If you are using the module starter, auto-imports will not be enabled in your playground either.

Tooling

Modules come with a set of first-party tools to help you with their development.

`@nuxt/module-builder`

Nuxt Module Builder is a zero-configuration build tool taking care of all the heavy lifting to build and ship your module. It ensures proper compatibility of your module build artifact with Nuxt applications.

`@nuxt/kit`

Nuxt Kit provides composable utilities to help your module interact with Nuxt applications. It's recommended to use Nuxt Kit utilities over manual alternatives whenever possible to ensure better compatibility and code readability of your module.

> 👉 Read more in API > Advanced > Kit.

`@nuxt/test-utils`

Nuxt Test Utils is a collection of utilities to help set up and run Nuxt applications within your module tests.

Recipes

Find here common patterns used to author modules.

Altering Nuxt Configuration

Nuxt configuration can be read and altered by modules. Here's an example of a module enabling an experimental feature.

```
import { defineNuxtModule } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    // We create the `experimental` object if it doesn't exist yet
    nuxt.options.experimental ||= {}
    nuxt.options.experimental.componentIslands = true
  }
})
```

When you need to handle more complex configuration alterations, you should consider using defu.

Exposing Options to Runtime

Because modules aren't part of the application runtime, their options aren't too. However, in many cases, you might need access to some of these module options within your runtime code. We recommend exposing the needed config using Nuxt's `runtimeConfig`.

```
import { defineNuxtModule } from '@nuxt/kit'
import { defu } from 'defu'

export default defineNuxtModule({
  setup (options, nuxt) {
    nuxt.options.runtimeConfig.public.myModule = defu(nuxt.options.runtimeConfig.public.myModule,
```

```
      foo: options.foo
    })
  }
})
```

Note that we use `defu` to extend the public runtime configuration the user can provides instead of overwriting it.

You can then access your module options in a plugin, component, the application like any other runtime configuration:

```
const options = useRuntimeConfig().public.myModule
```

Be careful not to expose any sensitive module configuration on the public runtime config, such as private API keys, as they will end up in the public bundle.

👉 Read more in Guide > Going Further > Runtime Config.

Injecting Plugins With `addPlugin`

Plugins are a common way for a module to add runtime logic. You can use the `addPlugin` utility to register them from your module.

```
import { defineNuxtModule, addPlugin, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    // Create resolver to resolve relative paths
    const { resolve } = createResolver(import.meta.url)

    addPlugin(resolve('./runtime/plugin'))
  }
})
```

👉 Read more in API > Advanced > Kit.

Injecting Vue Components With `addComponent`

If your module should provide Vue components, you can use the `addComponent` utility to add them as auto-imports for Nuxt to resolve.

```
import { defineNuxtModule, addComponent } from '@nuxt/kit'

export default defineNuxtModule({
  setup(options, nuxt) {
    const resolver = createResolver(import.meta.url)

    // From the runtime directory
    addComponent({
      name: 'MySuperComponent', // name of the component to be used in vue templates
      export: 'MySuperComponent', // (optional) if the component is a named (rather than default)
      filePath: resolver.resolve('runtime/components/MySuperComponent.vue')
    })

    // From a library
    addComponent({
      name: 'MyAwesomeComponent', // name of the component to be used in vue templates
      export: 'MyAwesomeComponent', // (optional) if the component is a named (rather than default
      filePath: '@vue/awesome-components'
    })
  }
})
```

Injecting Composables With `addImports` and `addImportsDir`

If your module should provide composables, you can use the `addImports` utility to add them as auto-imports for Nuxt to resolve.

```
import { defineNuxtModule, addImports, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup(options, nuxt) {
    const resolver = createResolver(import.meta.url)

    addImports({
      name: 'useComposable', // name of the composable to be used
      as: 'useComposable',
```

```
      from: resolver.resolve('runtime/composables/useComposable') // path of composable
    })
  }
})
```

Alternatively, you can add an entire directory by using `addImportsDir`.

```
import { defineNuxtModule, addImportsDir, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup(options, nuxt) {
    const resolver = createResolver(import.meta.url)

    addImportsDir(resolver.resolve('runtime/composables'))
  }
})
```

Injecting Server Routes With `addServerHandler`

```
import { defineNuxtModule, addServerHandler, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup(options, nuxt) {
    const resolver = createResolver(import.meta.url)

    addServerHandler({
      route: '/api/hello',
      handler: resolver.resolve('./runtime/server/api/hello/index.get.ts')
    })
  }
})
```

You can also add a dynamic server route:

```
import { defineNuxtModule, addServerHandler, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup(options, nuxt) {
    const resolver = createResolver(import.meta.url)
```

```
    addServerHandler({
      route: '/api/hello/:name',
      handler: resolver.resolve('./runtime/server/api/hello/[name].get.ts')
    })
  }
})
```

Injecting Other Assets

If your module should provide other kinds of assets, they can also be injected. Here's a simple example module injecting a stylesheet through Nuxt's `css` array.

```
import { defineNuxtModule, addPlugin, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    const { resolve } = createResolver(import.meta.url)

    nuxt.options.css.push(resolve('./runtime/style.css'))
  }
})
```

And a more advanced one, exposing a folder of assets through Nitro's `publicAssets` option:

```
import { defineNuxtModule, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    const { resolve } = createResolver(import.meta.url)

    nuxt.hook('nitro:config', async (nitroConfig) => {
      nitroConfig.publicAssets ||= []
      nitroConfig.publicAssets.push({
        dir: resolve('./runtime/public'),
        maxAge: 60 * 60 * 24 * 365 // 1 year
      })
    })
  }
})
```

Using Other Modules in Your Module

If your module depends on other modules, you can add them by using Nuxt Kit's `installModule` utility. For example, if you wanted to use Nuxt Tailwind in your module, you could add it as below:

```
import { defineNuxtModule, createResolver, installModule } from '@nuxt/kit'

export default defineNuxtModule<ModuleOptions>({
  async setup (options, nuxt) {
    const { resolve } = createResolver(import.meta.url)

    // We can inject our CSS file which includes Tailwind's directives
    nuxt.options.css.push(resolve('./runtime/assets/styles.css'))

    await installModule('@nuxtjs/tailwindcss', {
      // module configuration
      exposeConfig: true,
      config: {
        darkMode: 'class',
        content: {
          files: [
            resolve('./runtime/components/**/*.{vue,mjs,ts}'),
            resolve('./runtime/*.{mjs,js,ts}')
          ]
        }
      }
    })
  }
})
```

Using Hooks

Lifecycle hooks allow you to expand almost every aspect of Nuxt. Modules can hook to them programmatically or through the `hooks` map in their definition.

```
import { defineNuxtModule, addPlugin, createResolver } from '@nuxt/kit'

export default defineNuxtModule({
  // Hook to the `app:error` hook through the `hooks` map
  hooks: {
    'app:error': (err) => {
```

```
      console.info(`This error happened: ${err}`);
    }
  },
  setup (options, nuxt) {
    // Programmatically hook to the `pages:extend` hook
    nuxt.hook('pages:extend', (pages) => {
      console.info(`Discovered ${pages.length} pages`);
    })
  }
})
```

👉 Read more in API > Advanced > Hooks.

Module cleanup

If your module opens, handles, or starts a watcher, you should close it when the Nuxt lifecycle is done. The `close` hook is available for this.

```
import { defineNuxtModule } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    nuxt.hook('close', async nuxt => {
      // Your custom code here
    })
  }
})
```

Adding Templates/Virtual Files

If you need to add a virtual file that can be imported into the user's app, you can use the `addTemplate` utility.

```
import { defineNuxtModule, addTemplate } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    // The file is added to Nuxt's internal virtual file system and can be imported from '#build/m
    addTemplate({
```

```
      filename: 'my-module-feature.mjs',
      getContents: () => 'export const myModuleFeature = () => "hello world !"'
    })
  }
})
```

Adding Type Declarations

You might also want to add a type declaration to the user's project (for example, to augment a Nuxt interface or provide a global type of your own). For this, Nuxt provides the `addTypeTemplate` utility that both writes a template to the disk and adds a reference to it in the generated `nuxt.d.ts` file.

If your module should augment types handled by Nuxt, you can use `addTypeTemplate` to perform this operation:

```
import { defineNuxtModule, addTemplate, addTypeTemplate } from '@nuxt/kit'

export default defineNuxtModule({
  setup (options, nuxt) {
    addTypeTemplate({
      filename: 'types/my-module.d.ts',
      getContents: () => `// Generated by my-module
        interface MyModuleNitroRules {
          myModule?: { foo: 'bar' }
        }
        declare module 'nitropack' {
          interface NitroRouteRules extends MyModuleNitroRules {}
          interface NitroRouteConfig extends MyModuleNitroRules {}
        }
        export {}`
    })
  }
})
```

If you need more granular control, you can use the `prepare:types` hook to register a callback that will inject your types.

```
const template = addTemplate({ /* template options */ })
nuxt.hook('prepare:types', ({ references }) => {
  references.push({ path: template.dst })
})
```

Updating Templates

If you need to update your templates/virtual files, you can leverage the `updateTemplates` utility like this :

```
nuxt.hook('builder:watch', async (event, path) => {
  if (path.includes('my-module-feature.config')) {
    // This will reload the template that you registered
    updateTemplates({ filter: t => t.filename === 'my-module-feature.mjs' })
  }
})
```

Testing

Testing helps ensuring your module works as expected given various setup. Find in this section how to perform various kinds of tests against your module.

Unit and Integration

We're still discussing and exploring how to ease unit and integration testing on Nuxt Modules.

Check out this RFC to join the conversation.

End to End

Nuxt Test Utils is the go-to library to help you test your module in an end-to-end way. Here's the workflow to adopt with it:

1. Create a Nuxt application to be used as a "fixture" inside `test/fixtures/*`
2. Setup Nuxt with this fixture inside your test file
3. Interact with the fixture using utilities from `@nuxt/test-utils` (e.g. fetching a page)
4. Perform checks related to this fixture (e.g. "HTML contains ...")
5. Repeat

In practice, the fixture:

test/fixtures/ssr/nuxt.config.ts

```ts
// 1. Create a Nuxt application to be used as a "fixture"
import MyModule from '../../../src/module'

export default defineNuxtConfig({
  ssr: true,
  modules: [
    MyModule
  ]
})
```

And its test:

test/rendering.ts

```ts
import { describe, it, expect } from 'vitest'
import { fileURLToPath } from 'node:url'
import { setup, $fetch } from '@nuxt/test-utils'

describe('ssr', async () => {
  // 2. Setup Nuxt with this fixture inside your test file
  await setup({
    rootDir: fileURLToPath(new URL('./fixtures/ssr', import.meta.url)),
  })

  it('renders the index page', async () => {
    // 3. Interact with the fixture using utilities from `@nuxt/test-utils`
    const html = await $fetch('/')

    // 4. Perform checks related to this fixture
    expect(html).toContain('<div>ssr</div>')
  })
})

// 5. Repeat
describe('csr', async () => { /* ... */ })
```

An example of such a workflow is available on the module starter.

Manual QA With Playground and Externally

Having a playground Nuxt application to test your module when developing it is really useful. The module starter integrates one for that purpose.

You can test your module with other Nuxt applications (applications that are not part of your module repository) locally. To do so, you can use `npm pack` command, or your package manager equivalent, to create a tarball from your module. Then in your test project, you can add your module to `package.json` packages as: `"my-module": "file:/path/to/tarball.tgz"` .

After that, you should be able to reference `my-module` like in any regular project.

Best Practices

With great power comes great responsibility. While modules are powerful, here are some best practices to keep in mind while authoring modules to keep applications performant and developer experience great.

Async Modules

As we've seen, Nuxt Modules can be asynchronous. For example, you may want to develop a module that needs fetching some API or calling an async function.

However, be careful with asynchronous behaviors as Nuxt will wait for your module to setup before going to the next module and starting the development server, build process, etc. Prefer deferring time-consuming logic to Nuxt hooks.

> If your module takes more than **1 second** to setup, Nuxt will emit a warning about it.

Always Prefix Exposed Interfaces

Nuxt Modules should provide an explicit prefix for any exposed configuration, plugin, API, composable, or component to avoid conflict with other modules and internals.

Ideally, you should prefix them with your module's name (e.g. if your module is called `nuxt-foo` , expose `<FooButton>` and `useFooBar()` and **not** `<Button>` and `useBar()`).

Be TypeScript Friendly

Nuxt 3, has first-class TypeScript integration for the best developer experience.

Exposing types and using TypeScript to develop modules benefits users even when not using TypeScript directly.

Avoid CommonJS Syntax

Nuxt 3, relies on native ESM. Please read Native ES Modules for more information.

Document Module Usage

Consider documenting module usage in the readme file:

- Why use this module?
- How to use this module?
- What does this module do?

Linking to the integration website and documentation is always a good idea.

Provide a StackBlitz Demo or Boilerplate

It's a good practice to make a minimal reproduction with your module and StackBlitz that you add to your module readme.

This not only provides potential users of your module a quick and easy way to experiment with the module but also an easy way for them to build minimal reproductions they can send you when they encounter issues.

Do Not Advertize With a Specific Nuxt Version

Nuxt 3, Nuxt Kit, and other new toolings are made to have both forward and backward compatibility in mind.

Please use "X for Nuxt" instead of "X for Nuxt 3" to avoid fragmentation in the ecosystem and prefer using `meta.compatibility` to set Nuxt version constraints.

Stick With Starter Defaults

The module starter comes with a default set of tools and configurations (e.g. ESLint configuration). If you plan on open-sourcing your module, sticking with those defaults ensures your module shares a consistent coding style with other community modules out there, making it easier for others to contribute.

Ecosystem

Nuxt Module ecosystem represents more than 15 million monthly NPM downloads and provides extended functionalities and integrations with all sort of tools. You can be part of this ecosystem!

Module Types

Official modules are modules prefixed (scoped) with `@nuxt/` (e.g. `@nuxt/content`). They are made and maintained actively by the Nuxt team. Like with the framework, contributions from the community are more than welcome to help make them better!

Community modules are modules prefixed (scoped) with `@nuxtjs/` (e.g. `@nuxtjs/tailwindcss`). They are proven modules made and maintained by community members. Again, contributions are welcome from anyone.

Third party and other community modules are modules (often) prefixed with `nuxt-`. Anyone can make them, using this prefix allows these modules to be discoverable on npm. This is the best starting point to draft and try an idea!

Private or personal modules are modules made for your own use case or company. They don't need to follow any naming rules to work with Nuxt and are often seen scoped under an npm organization (e.g. `@my-company/nuxt-auth`)

Listing Your Community Module

Any community modules are welcome to be listed on the module list. To be listed, open an issue in the nuxt/modules repository. The Nuxt team can help you to apply best practices before listing.

Joining `nuxt-modules` and `@nuxtjs/`

By moving your modules to nuxt-modules, there is always someone else to help, and this way, we can join forces to make one perfect solution.

If you have an already published and working module, and want to transfer it to `nuxt-modules`, open an issue in nuxt/modules.

By joining `nuxt-modules` we can rename your community module under the `@nuxtjs/` scope and provide a subdomain (e.g. `my-module.nuxtjs.org`) for its documentation.

Edit on Github