

[Table of Contents](#) >

Plugins Directory

Nuxt automatically reads the files in your `plugins` directory and loads them at the creation of the Vue application. You can use `.server` or `.client` suffix in the file name to load a plugin only on the server or client side.

All plugins in your `plugins/` directory are auto-registered, so you should not add them to your `nuxt.config` separately.

Which Files Are Registered

Only files at the top level of the `plugins/` directory (or index files within any subdirectories) will be registered as plugins.

For example:

```
> plugins
> | - myPlugin.ts // scanned
> | - myOtherPlugin
> | --- supportingFile.ts // not scanned
> | --- componentToRegister.vue // not scanned
> | --- index.ts // currently scanned but deprecated
```

Only `myPlugin.ts` and `myOtherPlugin/index.ts` would be registered. You can configure `plugins` to include unscanned files.

Creating Plugins

The only argument passed to a plugin is `nuxtApp`.

```
export default defineNuxtPlugin(nuxtApp => {  
  // Doing something with nuxtApp  
})
```

Object Syntax Plugins

It is also possible to define a plugin using an object syntax, for more advanced use cases. For example:

```
export default defineNuxtPlugin({  
  name: 'my-plugin',  
  enforce: 'pre', // or 'post'  
  async setup (nuxtApp) {  
    // this is the equivalent of a normal functional plugin  
  },  
  hooks: {  
    // You can directly register Nuxt app hooks here  
    'app:created'() {  
      const nuxtApp = useNuxtApp()  
      //  
    }  
  }  
})
```

You can also use the `env` property to define the environments in which your plugin will run.

- **islands**

- **default:** `true`
- **description:** You can set this value to `false` if you don't want the plugin to run when rendering server-only or island components.

If you are using an object-syntax plugin, the properties may be statically analyzed in future to produce a more optimized build. So you should not define them at runtime. For example, setting `enforce: process.server ? 'pre' : 'post'` would defeat any future optimization Nuxt is able to do for your plugins.

Plugin Registration Order


You can control the order in which plugins are registered by prefixing with 'alphabetical' numbering to the file names.

For example:

```
> plugins/  
> | - 01.myPlugin.ts  
> | - 02.myOtherPlugin.ts
```

In this example, `02.myOtherPlugin.ts` will be able to access anything that was injected by `01.myPlugin.ts`.

This is useful in situations where you have a plugin that depends on another plugin.

 In case you're new to 'alphabetical' numbering, remember that filenames are sorted as strings, not as numeric values. For example, `10.myPlugin.ts` would come before `2.myOtherPlugin.ts`. This is why the example prefixes single digit numbers with `0`.

Loading strategy

By default, Nuxt loads plugins sequentially. You can define a plugin as `parallel` so Nuxt won't wait the end of the plugin's execution before loading the next plugin.

```
export default defineNuxtPlugin({  
  name: 'my-plugin',  
  parallel: true,  
  async setup (nuxtApp) {  
    // the next plugin will be executed immediately  
  }  
})
```

Using Composables Within Plugins

You can use composables within Nuxt plugins:

```
export default defineNuxtPlugin((NuxtApp) => {  
  const foo = useFoo()  
})
```

However, keep in mind there are some limitations and differences:

If a composable depends on another plugin registered later, it might not work.

Reason: Plugins are called in order sequentially and before everything else. You might use a composable that depends on another plugin which has not been called yet.

If a composable depends on the Vue.js lifecycle, it won't work.

Reason: Normally, Vue.js composables are bound to the current component instance while plugins are only bound to `nuxtApp` instance.

Automatically Providing Helpers

If you would like to provide a helper on the `NuxtApp` instance, return it from the plugin under a `provide` key. For example:

```
export default defineNuxtPlugin(() => {  
  return {  
    provide: {  
      hello: (msg: string) => `Hello ${msg}!`  
    }  
  }  
})
```

In another file you can use this:

```
<script setup lang="ts">
// alternatively, you can also use it here
const { $hello } = useNuxtApp()
</script>

<template>
  <div>
    {{ $hello('world') }}
  </div>
</template>
```

Typing Plugins

If you return your helpers from the plugin, they will be typed automatically; you'll find them typed for the return of `useNuxtApp()` and within your templates.

If you need to use a provided helper *within* another plugin, you can call `useNuxtApp()` to get the typed version. But in general, this should be avoided unless you are certain of the plugins' order.

Advanced

For advanced use-cases, you can declare the type of injected properties like this:

```
declare module '#app' {
  interface NuxtApp {
    $hello (msg: string): string
  }
}
```

```
declare module 'vue' {
  interface ComponentCustomProperties {
    $hello (msg: string): string
  }
}
```

```
export { }
```

index.d.ts

If you are using WebStorm, you may need to augment `@vue/runtime-core` until [this issue](#) is resolved.

Vue Plugins

If you want to use Vue plugins, like `vue-gtag` to add Google Analytics tags, you can use a Nuxt plugin to do so.

First, install the plugin you want.

```
> yarn add --dev vue-gtag-next
```

Then create a plugin file `plugins/vue-gtag.client.js`.

```
import VueGtag, { trackRouter } from 'vue-gtag-next'

export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.vueApp.use(VueGtag, {
    property: {
      id: 'GA_MEASUREMENT_ID'
    }
  })
  trackRouter(useRouter())
})
```

Vue Directives

Similarly, you can register a custom Vue directive in a plugin. For example, in `plugins/directive.ts`:

```
export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.vueApp.directive('focus', {
    mounted (el) {
      el.focus()
    },
    getSSRProps (binding, vnode) {
      // you can provide SSR-specific props here
      return {}
    }
  })
})
```

```
  })  
})
```

If you register a Vue directive, you *must* register it on both client and server side unless you are only using it when rendering one side. If the directive only makes sense from a client side, you can always move it to `~/plugins/my-directive.client.ts` and provide a 'stub' directive for the server in `~/plugins/my-directive.server.ts` .



© 2016-2023 Nuxt - MIT
License

Enterprise

Design Kit

NuxtLabs

Nuxt Studio



[✏ Edit on Github](#)