Components Directory

The `components/` directory is where you put all your Vue components which can then be imported inside your pages or other components (learn more).

Nuxt automatically imports any components in your `components/` directory (along with components that are registered by any modules you may be using).

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

layouts/default.vue

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

Custom directories

By default, only the `~/components` directory is scanned. If you want to add other directories, or change how the components are scanned within a subfolder of this directory, you can add additional directories to the configuration:

nuxt.config.ts

```
export default defineNuxtConfig({
  components: [
    // ~/calendar-module/components/event/Update.vue => <EventUpdate />
    { path: '~/calendar-module/components' },

    // ~/user-module/components/account/UserDeleteDialog.vue => <UserDeleteDialog />
    { path: '~/user-module/components', pathPrefix: false },

    // ~/components/special-components/Btn.vue => <SpecialBtn />
    { path: '~/components/special-components', prefix: 'Special' },

    // It's important that this comes last if you have overrides you wish to apply
    // to sub-directories of `~/components`.
    //
    // ~/components/Btn.vue => <Btn />
    // ~/components/base/Btn.vue => <BaseBtn />
    '~/components'
  ]
})
```

Any nested directories need to be added first as they are scanned in order.

Component extensions

By default, any file with an extension specified in the **extensions key of** `nuxt.config.ts` is treated as a component. If you need to restrict the file extensions that should be registered as components, you can use the extended form of the components directory declaration and its `extensions` key:

```
export default defineNuxtConfig({
  components: [
    {
      path: '~/components',
+     extensions: ['.vue'],
    }
  ]
})
```

Component Names

If you have a component in nested directories such as:

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

... then the component's name will be based on its own path directory and filename, with duplicate segments being removed. Therefore, the component's name will be:

```
<BaseFooButton />
```

For clarity, we recommend that the component's filename matches its name. (So, in the example above, you could rename `Button.vue` to be `BaseFooButton.vue` .)

If you want to auto-import components based only on its name, not path, then you need to set `pathPrefix` option to `false` using extended form of the configuration object:

```
export default defineNuxtConfig({
  components: [
    {
      path: '~/components',
+     pathPrefix: false,
    },
  ],
});
```

This registers the components using the same strategy as used in Nuxt 2. For example, `~/components/Some/MyComponent.vue` will be usable as `<MyComponent>` and not `<SomeMyComponent>` .

Dynamic Components

If you want to use the Vue `<component :is="someComputedComponent">` syntax, you need to use the `resolveComponent` helper provided by Vue or import the component directly from `#components` and pass it into `is` prop.

For example:

```
<script setup lang="ts">
import { SomeComponent } from '#components'
```

```
const MyButton = resolveComponent('MyButton')
</script>

<template>
  <component :is="clickable ? MyButton : 'div'" />
  <component :is="SomeComponent" />
</template>
```

If you are using `resolveComponent` to handle dynamic components, make sure not to insert anything but the name of the component, which must be a string and not a variable.

Alternatively, though not recommended, you can register all your components globally, which will create async chunks for all your components and make them available throughout your application.

```
  export default defineNuxtConfig({
    components: {
+     global: true,
+     dirs: ['~/components']
    },
  })
```

You can also selectively register some components globally by placing them in a `~/components/global` directory.

The `global` option can also be set per component directory.

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's name.

```vue
<template>
  <div>
    <TheHeader />
    <slot />
    <LazyTheFooter />
  </div>
</template>
```
layouts/default.vue

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.

pages/index.vue
```vue
<script>
export default {
  data() {
    return {
      show: false
    }
  }
}
</script>

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

Direct Imports

You can also explicitly import components from `#components` if you want or need to bypass Nuxt's auto-importing functionality.

pages/index.vue
```vue
<script setup lang="ts">
import { NuxtLink, LazyMountainsList } from '#components'
const show = ref(false)
</script>
```

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

`<ClientOnly>` Component

Nuxt provides the `<ClientOnly>` component for purposely rendering a component only on client side. To import a component only on the client, register the component in a client-side only plugin.

pages/example.vue

```
<template>
  <div>
    <Sidebar />
    <ClientOnly>
      <!-- this component will only be rendered on client-side -->
      <Comments />
    </ClientOnly>
  </div>
</template>
```

Use a slot as fallback until `<ClientOnly>` is mounted on client side.

pages/example.vue

```
<template>
  <div>
    <Sidebar />
    <!-- This renders the "span" element on the server side -->
    <ClientOnly fallbackTag="span">
      <!-- this component will only be rendered on client side -->
      <Comments />
      <template #fallback>
        <!-- this will be rendered on server side -->
        <p>Loading comments...</p>
      </template>
    </ClientOnly>
  </div>
</template>
```

.client Components

If a component is meant to be rendered only client-side, you can add the `.client` suffix to your component.

```
> | components/
> --| Comments.client.vue
```

pages/example.vue

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

This feature only works with Nuxt auto-imports and `#components` imports. Explicitly importing these components from their real paths does not convert them into client-only components.

`.client` components are rendered only after being mounted. To access the rendered template using `onMounted()` , add `await nextTick()` in the callback of the `onMounted()` hook.

.server Components

`.server` components can either be used on their own or paired with a `.client` component.

Standalone server components

Standalone server components will always be rendered on the server. When their props update, this will result in a network request that will update the rendered HTML in-place.

```
export defaul        eNuxtConfig({
  experimenta
    componentIslands: true
  }
```

> A video made by LearnVue for the Nuxt documentation.

Server components are currently experimental and in order to use them, you need to enable the 'component islands' feature in your nuxt.config:

```ts
// nuxt.config.ts
export default defineNuxtConfig({
  experimental: {
    componentIslands: true
  }
})
```

Now you can register server-only components with the `.server` suffix and use them anywhere in your application automatically.

```
> | components/
> --| HighlightedMarkdown.server.vue
```

```vue
// pages/example.vue
<template>
  <div>
    <!--
      this will automatically be rendered on the server, meaning your markdown parsing + highlight
      libraries are not included in your client bundle.
     -->
    <HighlightedMarkdown markdown="# Headline" />
  </div>
</template>
```

Server-only components use `<NuxtIsland>` under the hood, meaning that `lazy` prop and `#fallback` slot are both passed down to `<NuxtIsland>`.

Server Component Context

When rendering a server-only or island component, `<NuxtIsland>` makes a fetch request which comes back with a `NuxtIslandResponse`. (This is an internal request if rendered on the server, or a request that you can see in the network tab if it's rendering on client-side navigation.)

This means:

- A new Vue app will be created server-side to create the `NuxtIslandResponse`.
- A new 'island context' will be created while rendering the component.
- You can't access the 'island context' from the rest of your app and you can't access the context of the rest of your app from the island component. In other words, the server component or island is *isolated* from the rest of your app.
- Your plugins will run again when rendering the island, unless they have `env: { islands: false }` set (which you can do in an object-syntax plugin).

Within an island component, you can access its island context through `nuxtApp.ssrContext.islandContext`. Note that while island components are still marked as experimental, the format of this context may change.

Slots can be interactive and are wrapped within a `<div>` with `display: contents;`

Paired with a `.client` component

In this case, the `.server` + `.client` components are two 'halves' of a component and can be used in advanced use cases for separate implementations of a component on server and client side.

```
> | components/
> --| Comments.client.vue
> --| Comments.server.vue
```

pages/example.vue

```
<template>
  <div>
    <!-- this component will render Comments.server server-side then Comments.client once mounted
    <Comments />
  </div>
</template>
```

It is essential that the client half of the component can 'hydrate' the server-rendered HTML. That is, it should render the same HTML on initial load, or you will experience a hydration mismatch.

`<DevOnly>` Component

Nuxt provides the `<DevOnly>` component to render a component only during development.

The content will not be included in production builds and tree-shaken.

pages/example.vue

```
<template>
  <div>
    <Sidebar />
    <DevOnly>
      <!-- this component will only be rendered during development -->
      <LazyDebugBar />

      <!-- if you ever require to have a replacement during production -->
      <!-- be sure to test these using `nuxt preview` -->
      <template #fallback>
        <div><!-- empty div for flex.justify-between --></div>
      </template>
    </DevOnly>
  </div>
</template>
```

`<NuxtClientFallback>` Component

Nuxt provides the `<NuxtClientFallback>` component to render its content on the client if any of its children trigger an error in SSR. You can specify a `fallbackTag` to make it render a specific tag if it fails to render on the server.

pages/example.vue

```
<template>
  <div>
    <Sidebar />
    <!-- this component will be rendered on client-side -->
    <NuxtClientFallback fallback-tag="span">
      <Comments />
      <BrokeInSSR />
    </NuxtClientFallback>
  </div>
</template>
```

Library Authors

Making Vue component libraries with automatic tree-shaking and component registration is super easy ✨

You can use the `components:dirs` hook to 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:dirs` hook:

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

```
export default defineNuxtModule({
  hooks: {
    'components:dirs': (dirs) => {
      const { resolve } = createResolver(import.meta.url)
      // Add ./components dir to the list
      dirs.push({
        path: fileURLToPath(resolve('./components')),
        prefix: 'awesome'
      })
    }
  }
})
```

That's it! Now in your project, you can import your UI library as a Nuxt module in your `nuxt.config` file:

nuxt.config.ts

```
export default defineNuxtConfig({
  modules: ['awesome-ui/nuxt']
})
```

... and directly use the module components (prefixed with `awesome-`) in our `pages/index.vue`:

```
<template>
```

```
</template>
```

It will automatically import the components only if used and also support HMR when updating your components in `node_modules/awesome-ui/components/`.

Read and edit a live example in Docs > Examples > Features > Auto Imports.

Edit on Github