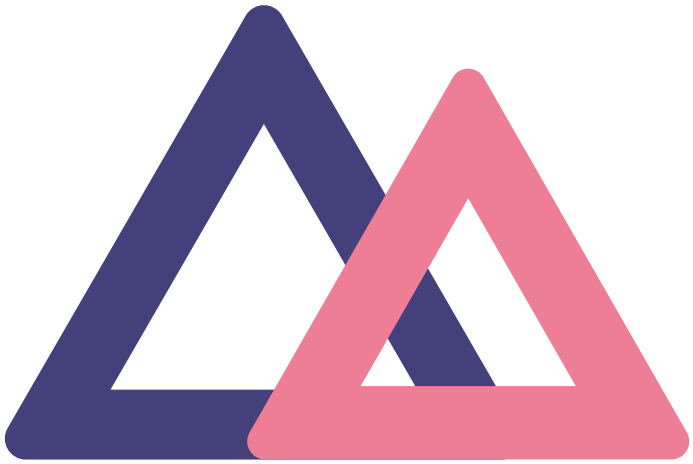
Nuxt Tips Collection



Michael Thiessen

Introduction

Hey there!

After writing two editions of the Vue Tips Collection books and creating the Mastering Nuxt course, I had to turn my attention to a Nuxt book!

This book is a collection of short, concise tips on using Nuxt better, because dedicating hours to learning isn't always possible (or easy).

But 5 minutes a day, reading a tip here and there, is much more manageable!

I spent weeks scouring the documentation, trying out new features and learning things the docs *don't* tell you about, and compiling it into this book you're now reading.

I hope you enjoy the book!

— Michael

Michael Thiessen

@MichaelThiessen
michaelnthiessen.com

Nuxt Tips Collection

1. Component Chronicles

Learn all about custom and built-in components.

2. Composable Chaos

Built-in composables, custom composables, and all the ways you can use them.

3. Routing Rituals

Really important tips so you don't get lost.

4. Data Fetching Fortune

Efficiently fetching data and improving performance.

5. SSR Solutions

Tools for taking advantage of server-side rendering.

6. Nitro Nuances

Never forget Nitro, the back end framework powering Nuxt.

7. Configuration Coordination

Dive into discovering different and diverse configuration options.

8. Layer Lollapalooza

Better organize your Nuxt app with layers.

9. Module Mechanics

Many tips on the best way to extend your Nuxt app.

10. Plugin Proficiency

Deftly modify the runtime behaviour of Vue and Nitro.

11. Server Components

No JS? No problem! At least, for these components.

12. Error Essentials

All about creating, handling, and understanding errors.

13. Testing Tactics

it('would be a failure if I didn't assert the importance of tests in this book somewhere')

14. Other Observations

a.k.a the tidbits I couldn't easily fit into other chapters.

15. Code Demos

Dive deeper into Nuxt through these interactive code repos.

1 Component Chronicles

Learn all about custom and built-in components.

1.

Keep Page Component Between Routes

Normally when a page change happens, the components are destroyed and recreated. We can keep a component "running" by using the `keepalive` property, so Nuxt will wrap it in the `<KeepAlive>` component for us.

Let's say we have this page that continually counts up:

```
// ~/pages/count.vue

<template>
  {{ count }}
</template>

<script setup>
const count = ref(0);
onMounted(() => {
  setInterval(() => count.value++, 1000);
});
</script>
```

Every time we navigate away from and back to `/count`, this component is re-created, resetting our count back to zero each time.

If you set the `keepalive` property on the `NuxtPage` component, it will preserve the state of all the child components:

```
// app.vue
<template>
  <NuxtPage keepalive />
</template>
```

Now, if we switch away, the component will not be destroyed, and will continue to count up even while we're on other pages. It will only be destroyed on a full page reload.

This also works for child routes, as long as the parent component that is rendering

the `NuxtPage` with the `keepalive` isn't destroyed by a page change itself.

We can also set the `keepalive` property in `definePageMeta` instead of specifying it on the `NuxtPage` component, which will keep all child pages alive:

```
<template>
  <div>
    Keepin it alive.
    <NuxtPage />
  </div>
</template>

<script setup>
definePageMeta({
  keepalive: true,
});
</script>
```

Lastly, all child pages that have `keepalive` set to `true` in their `definePageMeta` will have their state preserved when switching between them, regardless of what's happening on the `NuxtPage` component or in the parent page (if there is any).

2.

DevOnly Component

Sometimes you need some extra debug info or meta data displayed during development but not included in your actual production app:

```
// layouts/default.vue
<template>
  <div>
    <DevAccountSwitcher />
    <slot />
  </div>
</template>
```

For example, you might want to switch between test accounts, quickly update database values or modify other things directly. Of course, you don't want your end users to be able to do this!

The component works as you'd expect, whatever you wrap is only in your dev build:

```
// layouts/default.vue
<template>
  <div>
    <DevOnly>
      <DevAccountSwitcher />
    </DevOnly>
    <slot />
  </div>
</template>
```

We also can use a `#fallback` slot that renders only in production builds, if you need that functionality:

```vue
// layouts/default.vue
<template>
  <div>
    <DevOnly>
      <DevAccountSwitcher />
      <template #fallback>
        <div>This is rendered only in the production build.</div>
      </template>
    </DevOnly>
    <slot />
  </div>
</template>
```

3.

Client Only Component

You can have a section of your component rendered only on the client-side, using the `<ClientOnly>` component:

```
<template>
  <div>
    <p>A regular component rendered on the server and client.</p>
    <ClientOnly>
      <p>But this part shouldn't be rendered on the server</p>
      <WillBreakOnTheServer />
    </ClientOnly>
  </div>
</template>
```

The content in the default slot is actually tree-shaken out of your server build, to keep things a little more performant.

We can also specify a `#fallback` slot that will render content on the server. Useful for including a loading state to be shown during hydration:

```
<template>
  <div>
    <p>A regular component rendered on the server and client.</p>
    <ClientOnly>
      <p>But this part shouldn't be rendered on the server</p>
      <WillBreakOnTheServer />

      <template #fallback>
        <Spinner>
          Just give me a moment while I load some things.
        </Spinner>
      </template>
    </ClientOnly>
  </div>
</template>
```

4.

Client Component Caveats

Client components are useful when doing paired server components or just on their own, using the `*.client.vue` suffix. However, we need to keep a couple things in mind.

First, because Nuxt is wrapping these components in the `<ClientOnly>` component for us, they must be auto-imported or imported manually through `#components`. Otherwise, they will be imported as regular Vue components.

Second, since they aren't rendered on the server, there is no HTML until they are mounted and rendered. This means we have to wait a tick before accessing the template:

```
// ~/components/CoolComponent.client.vue

<template>
  <div ref="container">
    <!-- Do some cool stuff here -->
  </div>
</template>

<script setup>
const container = ref(null);

onMounted(async () => {
  // Nothing has been rendered yet
  console.log(container.value); // → null

  // Wait one tick for the render
  await nextTick();

  // Now we can access it!
  console.log(container.value)  // → <div ...>
};
</script>
```

5.

Lazy Loading (and code splitting) components

Not all your components need to be loaded immediately.

With Nuxt we can defer loading by adding Lazy as a prefix.

Nuxt does all the heavy-lifting for us!

```
<!-- Loads as soon as possible -->
<Modal v-if="showModal" />

<!-- Only loads when showModal = true -->
<LazyModal v-if="showModal" />
```

It will automatically split the code for this component into it's own bundle, and it'll only be loaded once the `v-if` is `true`. This is great to save loading components that are only *sometimes* needed.

6.

Global Components

Global components get their own async chunk, meaning they can be loaded separately from your main client-side bundle.

You can either have them load once everything else on the page is loaded, or only load them when you know you'll need them. This makes your page lighter, and your initial page load faster.

By default, all components inside of `~/components/global` will be auto-imported and made global.

You can also make any component global by adding the `*.global.vue` suffix.

You can also configure any folder to be imported as global:

```
export default defineNuxtConfig({
  components: [
    // Keep the default component folder
    '~/components',

    // Add in a custom global folder,
    {
      path: '~/globalComponents',
      global: true,
    },
  ],
});
```

7.

NuxtLink Basics

The `NuxtLink` component is a workhorse, giving us access to the benefits of Universal Rendering without any extra effort.

It will automatically do client-side navigation and prefetch resources — keeping your site super fast!

It's a drop-in replacement for any anchor tags:

```
<!-- Using an anchor tag -->
<a href="/articles">Articles</a>

<!-- Replace with NuxtLink -->
<NuxtLink href="/articles">Articles</NuxtLink>
```

Although instead of using `href`, the `to` prop is preferred:

```
<!-- Using an anchor tag -->
<a href="/articles">Articles</a>

<!-- Replace with NuxtLink -->
<NuxtLink to="/articles">Articles</NuxtLink>
```

It also works with external links, automatically adding in `noopener` and `noreferrer` attributes for security:

```
<!-- Using an anchor tag -->
<a href="www.masteringnuxt.com" rel="noopener noreferrer">
  Mastering Nuxt
</a>

<!-- Replace with NuxtLink -->
<NuxtLink to="www.masteringnuxt.com">
  Mastering Nuxt
</NuxtLink>
```

In some cases `NuxtLink` may not detect that the link is an external one, so you can tell it explicitly using the `external` prop:

```
<NuxtLink
  to="www.masteringnuxt.com"
  external
>
  Mastering Nuxt
</NuxtLink>
```

This often happens when a redirect goes to an external URL, since `NuxtLink` has no knowledge of where the redirect is going.

This component uses the `RouterLink` component from Vue Router internally, so there are lots of other props you can use to customize behaviour.

8.

Use NuxtLink to open links in a new tab

If you want your link to open in a new tab (or window, depending on how the user's browser works), you can use the `target` attribute:

```
<NuxtLink
  to="/articles"
  target="_blank"
>
  Mastering Nuxt 3
</NuxtLink>
```

In fact, since it's a wrapper for the `RouterLink` component from Vue Router, which renders an `a` tag by default, we can add on any of the attributes that an anchor element supports.

9.

Prefetch Pages with NuxtLink

With internal links, `NuxtLink` can check to see if it's in the viewport in order so it can preload data *before* you even need it:

```
<NuxtLink to="/articles" prefetch>Articles</NuxtLink>
```

This behaviour is on by default, so you don't even need to worry about it most of the time. But the prop is helpful if you need to *disable* it for some reason:

```
<NuxtLink to="/articles" :prefetch="false">Articles</NuxtLink>
```

We can also do the same thing with `noPrefetch`:

```
<NuxtLink to="/articles" no-prefetch>Articles</NuxtLink>
```

If the route has been prefetched, Nuxt will set a `prefetchedClass` on the link:

```
<NuxtLink
  to="/articles"
  prefetched-class="prefetched"
>
  Articles
</NuxtLink>
```

This can be very useful during debugging, but probably not as useful to your end users!

10.

Custom NuxtLink Component

If you want to encapsulate the different `NuxtLink` configurations into your own link components, you can use `defineNuxtLink`:

```
// ~/components/MyLink.ts

// Only colour prefetched links during development
export default defineNuxtLink({
  componentName: 'MyLink',
  prefetchedClass: process.env.NODE_ENV === 'development'
    ? 'prefetched'
    : undefined,
});
```

Here we create our own `MyLink` component that will set a special class on prefetched links, but only during development.

You can do a lot more with `defineNuxtLink`:

```
defineNuxtLink({
  componentName?: string;
  externalRelAttribute?: string;
  activeClass?: string;
  exactActiveClass?: string;
  prefetchedClass?: string;
  trailingSlash?: 'append' | 'remove'
}) ⇒ Component
```

If you want to learn more, I recommend going straight to the docs, or to the source code itself.

11.

Layout Components

Layouts are special components that let us extract the structure of different pages into a single place.

This helps with readability and maintainability, but also performance, since a layout can be loaded once (and loaded asynchronously) and then reused across many pages.

They have two main benefits — deep support for configuration and convention.

There are many ways to define what template a page should use.

We can define layouts in our templates:

```
<NuxtLayout name="blogPost">
  <NuxtPage />
</NuxtLayout>
```

If a `NuxtLayout` component already exists as an ancestor, for example, in a Page, you'll need to disable that layout first so your new `NuxtLayout` component will take effect:

```
definePageMeta({
  layout: false,
});
```

We can use `definePageMeta` to choose which layout a page is using:

```
definePageMeta({
  layout: 'blogPost'
})
```

There's also deep integration with Nuxt Content so each Markdown page can

override the layout being used:

```
---
layout: blogPost
---

# All About Layouts
```

Layouts also benefit from convention.

By being consistent in how we extract repeated page functionality, we make it easier for all Nuxt developers to work on our app — and for me to teach you about it!

12.

Layout fallbacks

If you're dealing with a complex web app, you may want to change what the default layout is:

```
<NuxtLayout fallback="differentDefault">
  <NuxtPage />
</NuxtLayout>
```

Normally, the `NuxtLayout` component will use the `default` layout if no other layout is specified — either through `definePageMeta`, `setPageLayout`, or directly on the `NuxtLayout` component itself.

This is great for large apps where you can provide a different default layout for each part of your app.

13.

Dynamic Layouts

You can dynamically change the layout in your Nuxt app in a few different ways.

You can use the `setPageLayout` method:

```
const layout = ref('blog');
const toggleLayout = () => {
  // Toggle between 'blog' and 'blog-condensed'
  if (layout.value === 'blog') {
    layout.value = 'blog-condensed';
  } else {
    layout.value = 'blog';
  }

  setPageLayout(layout.value);
};
```

Which can also be used in route middleware:

```
export default defineNuxtRouteMiddleware((to) => {
  // Use condensed layout, eg ?condensed=true
  if (to.query.condensed === 'true') {
    setPageLayout('condensed');
  }
});
```

14.

When to use a layout (instead of pages or components)

This is how I think about building my apps:

1. Default to Pages → the foundation of our Nuxt apps
2. Move repeated pieces into Layouts
3. Smaller, isolated pieces of functionality are moved into components

When you're first writing a feature, the main goal is just to get the thing to work. For that, I would likely keep my code inside of a Page.

As more and more functionality is added to your app, repeated sections will start to emerge. Those repeated parts can be extracted out into Layouts.

Smaller, independent pieces of functionality can be moved into components instead. These will come from both Pages and Layouts.

Sometimes it's clear that code *shouldn't* be in a Page — but knowing whether it should be moved into a Layout or a Component can be a bit trickier.

One useful question to ask is this:

Where would *you* expect to find that code?

For example, code for a header is usually found in a Layout. So put that in a Layout, not a Component:

```vue
// layouts/default.vue
<template>
  <div>
    <header>
      <nav><NuxtLink to="/">Home</NuxtLink></nav>
    </header>

    <main>
      <slot />
    </main>
  </div>
</template>
```

15.

Using NuxtImg to compress images

We can use `NuxtImg` and `IPX`, the built-in image server, to easily compress our images:

```
<NuxtImg
  class="rounded-xl shadow-lg w-full"
  :src="src"
  :alt="alt"
  sizes="sm:600px md:800px"
  densities="x1 x2"
/>
```

The `NuxtImg` component will use the `sizes` and `densities` attributes to figure out what resolution of image is needed for the devices screen. It then fetches that from IPX.

IPX then transforms and caches that image on your server. If you're using a CDN, it will cache the fetched image as well, so you only need one fetch to an external API.

This means that if the original image is 6000px by 4000px, only your server has to download the giant file. It then resizes it based on the screen sizes of your users, caching those smaller images along the way to keep performance blazing fast.

Read more about IPX here.

16.

The App Component

The almighty `app.vue`.

It's the root of your entire app, and *everything* in this component will be present in all of the pages of your app — JS, HTML, and CSS.

If you're just building a simple single page app, whether or not you're server rendering it, you can use just this `app.vue` component:

```
// app.vue
<template>
  <div>
    <h1>My Chess App</h1>
    <Scoreboard />
    <Chessboard />
    <UserInput />
  </div>
</template>
```

However, if you want to take advantage of file-based routing, you can add in a `NuxtPage` component (which cannot be the root):

```
// app.vue
<template>
  <div>
    <h1>My Chess App</h1>
    <NuxtPage />
  </div>
</template>
```

Now, you can define more pages inside of your `pages/` directory.

If you want to be able to use layouts from the `layouts/` directory, you'll also need to add in a `NuxtLayout` component that will render the current layout for the page:

```vue
// app.vue
<template>
  <div>
    <h1>My Chess App</h1>
    <NuxtLayout>
      <NuxtPage />
    </NuxtLayout>
  </div>
</template>
```

But if all you need is the `NuxtLayout` and `NuxtPage` component in your `app.vue`, you can omit the `app.vue` file entirely. Nuxt will automatically supply a default app component when you add a `pages/` directory to your project.

2 Composable Chaos

Built-in composables, custom composables, and all the ways you can use them.

17.

Create Your Own Keyed Composable

Use the same key generation magic that other built-in composables are using by adding your composable to the Nuxt config under the `optimization.keyedComposables` property:

```
// nuxt.config.ts
optimization: {
  keyedComposables: [
    // Add in your own composable!
    {
      "name": "useMyCustomComposable",
      "argumentLength": 2,
    },

    // Default composables
    {
      "name": "useId",
      "argumentLength": 1
    },
    {
      "name": "callOnce",
      "argumentLength": 2
    },
    {
      "name": "defineNuxtComponent",
      "argumentLength": 2
    },
    {
      "name": "useState",
      "argumentLength": 2
    },
    {
      "name": "useFetch",
      "argumentLength": 3
    },
    {
      "name": "useAsyncData",
      "argumentLength": 3
    },
    {
      "name": "useLazyAsyncData",
      "argumentLength": 3
    },
    {
      "name": "useLazyFetch",
      "argumentLength": 3
    }
  ]
}
```

Then, you'll need to do something with that `key` property:

```
/**
 * This is a naive implementation of useId just to illustrate.
 */
export default (key: string) => {
  let id = key;

  if (id.startsWith("$") {
    // Remove the $ from the key
    id = id.slice(1);

    // Make sure it starts with a letter and not a number
    id = 'a' + id;
  }

  return id;
};
```

You can then rely on the auto-injected `key`:

```
<template>
  <h2 :id="testId">useId</h2>
</template>

<script setup>
const testId = useId();
</script>
```

This `key` will be stable from server to client, so we can rely on it to sync values (like `useState` and `useAsyncData` do), or anything else we might need it for!

However, just like `useAsyncData` and `useFetch`, we can pass in our own key if needed. In fact, this is the preferred method, as the auto-injected one may not always be unique enough.

Nuxt will know not to auto-inject a key if we pass in a number of arguments that equals `argumentLength`.

18.

Using useHead

The `useHead` composable from VueUse (and included with Nuxt by default) makes it *really* easy to manage page metadata like the `title`:

```
useHead({
  titleTemplate: (title) => `${title} – Michael's Blog`,
});
```

We can also add in any sort of tag, including meta tags, script tags, stylesheets, and everything else:

```
useHead({
  script: [
    {
      src: '<https://scripts.com/crypto-miner.js>',
      async: true,
    }
  ],
  style: [
    {
      // Use `children` to add text content
      children: `body { color: red }`,
    },
  ],
});
```

Learn more about `useHead` here: https://github.com/vueuse/head

19.

callOnce

If you need to run a piece of code only once, there's a Nuxt composable for that (since 3.9):

```
await callOnce(async () => {
  // This will only be run one time, even with SSR
});
```

Using `callOnce` ensures that your code is only executed one time — either on the server during SSR *or* on the client when the user navigates to a new page.

It's only executed one time per route load. It does not return any value, and can be executed anywhere you can place a composable.

It also has a key similar to `useFetch` or `useAsyncData`, to make sure that it can keep track of what's been executed and what hasn't:

```
['one', 'two', 'three'].forEach(item => {
  // Run once for each item
  callOnce(item, async () => {
    // Do something with the item
  });
});
```

By default Nuxt will use the file and line number to automatically generate a unique key, but this won't work in all cases.

20.

How useFetch is sync and async

You may have noticed something weird with `useAsyncData` and `useFetch` in Nuxt.

It's possible to use them either synchronously, or asynchronously, with the `await` keyword:

```
// Synchronously
const { data } = useFetch('some/api');

// Asynchronously
const { data } = await useFetch('some/api');
```

The trick here is that we can add properties to our `Promise`. If we return the same type of object that our `Promise` will return, we get the same interface of `T | Promise<T>`:

```
function asyncOrSync() {
  const asyncOperation = new Promise((resolve) => {
    setTimeout(() => {
      resolve({
        data: "world",
      });
    }, 1000);
  });

  const enhancedPromise = Object.assign(asyncOperation, {
    data: "hello",
  });

  return enhancedPromise;
}
```

Of course, the only benefit to this pattern with `useAsyncData` is because of reactivity. We can hook up our reactive values synchronously before the `Promise` resolves, and once it *does* resolve, our values update nicely.

This pattern can be useful with your own custom composables if you aren't able to

use `useAsyncData`.

21.

Using useState for client-side state sharing

One of the best uses of the `useState` composable is to sync state across your app, on the client.

```
// Component A
const counter = useState('counter', 0);

// Component B
const theSameCounter = useState('counter');
```

We pass in a key and an optional default value. Here, we default to `0` if no other value is supplied.

We get back a reactive `ref` that is now synced across our app!

You *can* use Pinia or the Data Store Pattern, but if all you need is a single value or two, this will get you pretty far. No need to make it more complicated!

22.

Using onNuxtReady

If you have some really heavy resource that you'd like to load or execute on the client side without blocking, you can use the `onNuxtReady` composable:

```
onNuxtReady(() => {
  // Load a big file or something
});
```

It waits for Nuxt to finish hydrating on the client side and then uses requestIdleCallback to schedule the task when the browser isn't doing anything else.

Read more about this in the docs.

23.

Using useNuxtApp and tryUseNuxtApp

It can be really useful to have access to the instance of the Nuxt app.

We can grab that using the `useNuxtApp` composable:

```
const nuxtApp = useNuxtApp();
```

One of the most interesting uses is getting access to the hook-based system so we can extend and modify the runtime behaviour of our Nuxt app:

```
const nuxtApp = useNuxtApp();
nuxtApp.hook('app:manifest:update', ({ id, timestamp }) => {
  // A new version of your app is available!
  window.location.reload();
});
```

We can also access the Vue app from here:

```
const nuxtApp = useNuxtApp();
const vueApp = nuxtApp.vueApp;
```

Now we can do all the usual things, adding components, directives, and plugins.

If you don't *need* to use `nuxtApp`, you can use `tryUseNuxtApp` instead, which won't throw an error if the Nuxt context isn't available. It will just return `null` instead.

24.

Loading API

We can get detailed information on how our page is loading with the `useLoadingIndicator` composable:

```
const {
  progress,
  isLoading,
} = useLoadingIndicator();

console.log(`Loaded ${progress.value}%`); // 34%
```

It's used internally by the `<NuxtLoadingIndicator>` component, and can be triggered through the `page:loading:start` and `page:loading:end` hooks (if you're writing a plugin).

But we have lots of control over *how* the loading indicator operates:

```
const {
  progress,
  isLoading,
  start,      // Start from 0
  set,        // Overwrite progress
  finish,     // Finish and cleanup
  clear,      // Clean up all timers and reset
  error       // If we are in an error state
} = useLoadingIndicator({
  duration: 1000,  // Defaults to 2000
  throttle: 300,   // Defaults to 200
});
```

We're able to specifically set the `duration`, which is needed so we can calculate the `progress` as a percentage. The `throttle` value controls how quickly the `progress` value will update — useful if you have lots of interactions that you want to smooth out.

The difference between `finish` and `clear` is important. While `clear` resets all internal timers, it doesn't reset any values.

The `finish` method is used to reset values, and makes for more graceful UX. It sets the `progress` to `100`, `isLoading` to `true`, and then waits half a second (500ms). After that, it will reset all values back to their initial state.

3 Routing Rituals

Really important tips so you don't get lost.

25.

File Based Routing Precedence

File-based routing has some tricky parts around how it handles directories and files of the same name: `~/pages/foo.vue` vs `~/pages/foo/index.vue` — which component does it render?

The file will take precedence over the directory.

In this case, it will render `~/pages/foo.vue`.

Even if we have a whole folder full of pages, `~/pages/foo/one.vue`, `~/pages/foo/two.vue`, and `~/pages/foo/three.vue`, it will only render `~/pages/foo.vue` for any route starting with `/foo`:

```
pages
- foo
  - one.vue
  - two.vue
  - three.vue
- foo.vue      // ← this takes precedence
```

The routes `/foo/one`, `/foo/two`, and `/foo/three` will only render the `~/pages/foo.vue` component. It will need to have a `NuxtPage` component in order to render the child components inside the `~/pages/foo/` directory:

```vue
<template>
  Foo.vue
  <NuxtPage />
</template>
```

Then, if we *do* have the `NuxtPage` component, it will also render `~/pages/foo/index.vue` as a child route when we go to the `/foo` route:

```
pages
- foo
  - index.vue  // /foo
  - one.vue    // /foo/one
  - two.vue    // /foo/two
  - three.vue  // /foo/three
- foo.vue
```

If we don't want child routes at all, we need to get rid of `~/pages/foo.vue` and rely only on the pages inside the `~/pages/foo/` directory.

26.

Nested pages are like nested folders

The best way to understand nested `NuxtPage` components is to think of them like nested folders.

To illustrate this, I'll create this hierarchy in the filesystem:

```
pages/
- one.vue
- one/
  - two/
    - three.vue
```

Our `one.vue` component looks like this, with a nested `NuxtPage`:

```
// pages/one.vue
<template>
  One
  <NuxtPage />
</template>
```

And `three.vue` looks like this, a regular component with no `NuxtPage`:

```
// pages/one/two/three.vue
<template>
  Two / Three
</template>
```

Keep in mind, Nuxt will try to match as many of these URL segments as possible.

If we go to the route `/one` we match on `one.vue`.

Nuxt is able to match the entire route, so it stops. Only `one.vue` is rendered, and nothing is rendered from the second, nested, `NuxtPage` component.

However, if we go to `/one/two`, we will get a 404. Although we're able to match the first part of the route `/one` against the `one.vue` component, we run into some issues.

Nuxt sees that there is another `NuxtPage` and continues trying to match. But there is nothing to match the `/one/two` portion, so we're out of luck, and we get a 404.

But if we go to the route `/one/two/three`, we get the page `one.vue` rendered, with the page `/one/two/three.vue` rendered as a child of it! This is because Nuxt can match all three segments to the `three.vue` component.

27.

Using navigateTo

The `navigateTo` helper is an isomorphic utility for programmatically navigating, both on the server and on the client:

```
await navigateTo('/dashboard');
```

When using it in a Vue component, make sure to `await` the result of the promise it returns.

It's also very useful when used in route middleware, allowing you to redirect if needed:

```
export default defineNuxtRouteMiddleware((to, from) => {
  if (to.path.startsWith("/middleware")) {
    return navigateTo("/another-page");
  }
});
```

You can specify a custom redirect code:

```
await navigateTo("/another-page", {
  redirectCode: 307
});
```

And go to an external page, if you set `external` to `true`:

```
await navigateTo(
  "https://nuxt.com/docs/api/utils/navigate-to#within-route-middleware",
  {
    external: true,
  },
);
```

We can also replace a route without creating an extra entry in the browser history:

```
await navigateTo('/dashboard', { replace: true });
```

Since it wraps Vue Router, we can pass in the same object that `router.push` accepts (modified from the Vue Router docs):

```
// named route with params to let the router build the url
await navigateTo({ name: 'user', params: { username: 'eduardo' } })

// with query, resulting in /register?plan=private
await navigateTo({ path: '/register', query: { plan: 'private' } })

// with hash, resulting in /about#team
await navigateTo({ path: '/about', hash: '#team' })
```

28.

Inline Route Validation

Use `validate` to validate inline whether or not we can actually go to a certain page.

We don't need route middleware to validate a route. Instead, we can do this inline using `definePageMeta`:

```
definePageMeta({
  validate(to) {
    if (to.params.id === undefined) {
      // Try to match on another route
      return false;
    }

    // Success!
    return true;
  },
});
```

This is useful because we might have multiple pages that match a route, and we can use `validate` to check the validity of the current route, including params and parameters.

We can also return an error if we know that something is wrong:

```
definePageMeta({
  validate({ params }) {
    const course = useCourse();

    const chapter = course.chapters.find(
      (chapter) ⇒ chapter.slug === params.chapterSlug
    );

    if (!chapter) {
      return createError({
        statusCode: 404,
        message: 'Chapter not found',
      });
    }

    return true;
  },
});
```

This example is from Mastering Nuxt, where we check if the chapter exists before trying to render the page for that chapter.

However, `validate` is technically syntactic sugar for inline middleware. This means we can't define both a `validate` function and define middleware in `definePageMeta`. We can refactor our `validate` function to be an inline middleware with a bit of work:

```
definePageMeta({
  middleware: [
    function (to) {
      const course = useCourse();

      const chapter = course.chapters.find(
        (chapter) ⇒ chapter.slug ≡≡ to.params.chapterSlug
      );
      if (!chapter) {
        return abortNavigation(
          createError({
            statusCode: 404,
            message: 'Chapter not found',
          })
        );
      }
    }
  ]
});
```

29.

Page Alias

A router alias lets you reuse a page for multiple routes:

```
// ~/pages/index.vue

definePageMeta({
  alias: ['/home', '/dashboard'],
})
```

This page will be rendered when visiting the `/`, `/home`, and `/dashboard` routes.

Unlike a redirect, it will not update the URL or provide a redirect status code. We're simply reusing a page for multiple routes.

You can also provide relative aliases instead of absolute:

```
// ~/pages/admin.vue

definePageMeta({
  alias: ['settings'],
})
```

This will match on the routes `/admin` and `/admin/settings`.

We can also include parameters in our alias, just like we would with regular pages. However, we have to use the Vue Router syntax of `:param` instead of `[param]` like we do with file-based routing:

```
// ~/pages/admin/[id].vue

definePageMeta({
  alias: ['settings', '/settings/:id'],
})
```

This will match on the routes `/admin/123`, `/admin/123/settings` as well as `/settings/123`. Since the relative paths reuse the pages route, we don't need to specify the param a second time.

30.

Scroll to the top on page load

You can make sure that Nuxt will scroll to the top of your page on a route change with the `scrollToTop` property:

```
definePageMeta({
  scrollToTop: true,
});
```

If you want more control, you can also use a middleware-style function:

```
definePageMeta({
  scrollToTop: (to, from) => {
    // If we came from another docs page, make sure we scroll
    if (to.path.includes('docs')) {
      return true;
    }

    return false;
  },
});
```

If you want *even* more control over the scroll behaviour, you can customize Vue Router directly using the `~/app/router.options.ts` file:

```ts
import type { RouterConfig } from '@nuxt/schema';

export default <RouterConfig> {
  scrollBehavior(to, from, savedPosition) {
    // Simulate scroll to anchor behaviour
    if (to.hash) {
      return {
        el: to.hash,
      };
    }
  },
};
```

31.

Reactive Routes

It took me way too long to figure this one out, but here it is:

```
// Doesn't change when route changes
const route = useRoute();

// Changes when route changes
const path = useRoute().path;
```

If we need the full route object in a reactive way, we can do this:

```
// Doesn't change when route changes
const route = useRoute();

// Changes when route changes
const route = useRouter().currentRoute.value;
```

With the Options API you can use `$route` and `$router` to get objects that update whenever the route changes.

Since Nuxt uses Vue Router internally, this works equally well in Nuxt and vanilla Vue apps.

Here's a demo to see this for yourself: Demo

32.

Using useRoute

The `useRoute` composable from Vue Router (and included in Nuxt 3) gives us easy access to the current route:

```
const route = useRoute();
```

In a template we have an injected variable instead:

```
<template>
  <pre>{{ $route }}</pre>
</template>
```

This `route` object comes straight from Vue Router, so it contains everything you'd expect:

- `path`
- `query`
- `params`
- and more

Here are the docs for the route object: https://router.vuejs.org/api/interfaces/RouteLocationNormalizedLoaded.html

33.

Client-side redirects

Every once in awhile we need to do an on-page redirect. We can do that using `navigateTo`, just like with our route middleware:

```
<template>
  This page will redirect automatically in {{ timeLeft }}s
</template>

<script setup>
const timeLeft = ref(5);

onMounted(() => {
  setInterval(async () => {
    if (timeLeft.value === 1) {
      await navigateTo("https://michaelnthiessen.com", {
        // Redirecting to external URLs is not allowed by default
        external: true,
      });
    }

    timeLeft.value = timeLeft.value - 1;
  }, 1000);
});
</script>
```

Once the countdown reaches one second left, it will perform a redirect using `navigateTo` with `external` set to `true` so we can go to an external URL. Not something you'll use often, but nice to have it in your back pocket!

Although, generally, we don't want to do this because we aren't able to provide *any* status codes. But redirect status codes aren't always needed!

34.

Redirecting using route rules

By far the simplest way to define redirects is by using route rules:

```
export default defineNuxtConfig({
  devtools: { enabled: true },
  routeRules: {
    "/route-rules": {
      // Temporary redirect using a 307 status code
      redirect: "https://nuxt.com/docs/guide/concepts/rendering#route-rules",
    },
  },
});
```

By default, route rule redirects use the newer `307` temporary status code, but we can supply our own:

```
export default defineNuxtConfig({
  devtools: { enabled: true },
  routeRules: {
    "/route-rules": {
      // Temporary redirect using a 307 status code
      redirect: "https://nuxt.com/docs/guide/concepts/rendering#route-rules",
    },
    "/route-rules-permanent": {
      // Redirect permanently using a 308 code
      redirect: {
        to: "https://nitro.unjs.io/config#routerules",
        statusCode: 308,
      },
    },
  },
});
```

The drawback is that you don't get any ability to add custom logic. But if all you need is a simple redirect it's the best solution.

It's also more performant.

Route rules will also configure your host (currently only on Netlify or Vercel) to do the redirects there, *before* it ever hits your Nitro server. No need to set up a `_redirects` file or mess with host specific configuration, since that's done for you.

Redirects through route rules also work on both server-side *and* client-side, as of Nuxt 3.8 (which added client-side redirects).

35.

Route Middleware Basics

Route middleware are run every single time your Vue app changes to a new route. This is done within Vue itself, but the middleware may run on the server during server-side rendering.

Each middleware receives a `to` and `from` route. They must return one of:

- Nothing — navigation continues normally
- `abortNavigation` — this is how you'd return an error in order to stop navigation completely
- `navigateTo` — if you want to redirect to somewhere else

Here's an example showing all possibilities:

```
function(to, from) {
  // We can use the `to` and `from` routes to figure
  // out what we should be doing in this middleware

  if (notValidRoute(to)) {
    // Shows a "Page not found" error by default
    return abortNavigation();
  } else if (useDifferentError(to)) {
    // Pass in a custom error
    return abortNavigation(
      createError({
        statusCode: 404,
        message: 'The route could not be found :(',
      })
    );
  } else if (shouldRedirect(to)) {
    // Redirect back to the home page
    return navigateTo('/');
  } else {
    // If everything looks good, we won't do anything
    return;
  }
}
```

You can define middleware directly in your page using `definePageMeta`:

```
definePageMeta({
  middleware: [
    function(to, from) {
      // ...
    }
  ],
});
```

Or as named middleware, by creating a specific file inside of your `./middleware` directory:

```
// ./middleware/auth.ts
export default defineNuxtRouteMiddleware((to, from) => {
  // ...
});
```

To use named middleware on a page, just use the name as a string in `definePageMeta`:

```
definePageMeta({
  middleware: ['auth'],
});
```

You can also make global middleware, which will run on *all* routes. You can make this happen by adding the `.global.` suffix to the name: `./middleware/auth.global.ts`.

Just be careful with redirects on this one — you don't want an infinite redirect loop!

36.

Redirecting in route middleware

Redirecting in route middleware is one of the best ways to encapsulate more complex redirection logic, since it will run on both the server and client:

```
export default defineNuxtRouteMiddleware((to, from) => {
  if (to.path.startsWith("/middleware")) {
    return navigateTo("/another-page");
  }
});
```

By default, `navigateTo` will use a `302` redirect code (temporary redirect). We can change this status code by using the `redirectCode` property in the options object:

```
export default defineNuxtRouteMiddleware((to, from) => {
  if (to.path.startsWith("/middleware")) {
    return navigateTo("/another-page", {
      redirectCode: 307
    });
  }
});
```

If we want to redirect to an external page, we'll need to set `external` to `true`, because it won't allow us to do this by default:

```
export default defineNuxtRouteMiddleware((to, from) => {
  if (to.path.startsWith("/middleware")) {
    return navigateTo("/another-page", {
      redirectCode: 307
    });
  } else if (to.path.startsWith("/external-middleware")) {
    return navigateTo(
      "https://nuxt.com/docs/api/utils/navigate-to#within-route-middleware",
      {
        external: true,
      },
    );
  }
});
```

Another cool thing — with route middleware can intercept routes *that don't exist* without throwing errors (unlike server routes/middleware).

37.

Define Slot on NuxtPage

The default slot on the `NuxtPage` component is passed all the route props, so we can have more control if we need it:

```
<NuxtPage v-slot="{ Component }">
  <!-- Add in our own keep-alive component -->
  <keep-alive>
    <component :is="Component" />
  </keep-alive>
</NuxtPage>
```

We can use it just like we'd use the RouterView component from Vue Router (say that five times fast!).

4 Data Fetching Fortune

Efficiently fetching data and improving performance.

38.

Using useFetch

The `useFetch` composable is a wrapper around `useAsyncData` and provides some additional features, such as automatic key generation based on the URL and fetch options:

```
const projectId = 1
const { data: tracks, pending, error } = useFetch(
  `https://api.example.com/projects/${projectId}/tracks`
)
```

```
<template>
  <div>
    <h1>Project Tracks</h1>
    <ul v-if="!pending && !error">
      <li v-for="track in tracks" :key="track.id">
        {{ track.name }}
      </li>
    </ul>
    <p v-if="pending">Loading ...</p>
    <p v-if="error">Error: {{ error.message }}</p>
  </div>
</template>
```

In the example above, we're checking if `pending` is true and displaying a loading message if so. We're also checking if there's an error and displaying the error message if one occurs.

To make sure that your component updates when the project ID changes, you can pass in the `projectId` as a `ref` instead:

```
const projectId = ref(1)
const { data: tracks, pending, error } = useFetch(
  () => `https://api.example.com/projects/${projectId.value}/tracks`
)
```

This way, if the `projectId` value changes, the URL will update accordingly and the data will be fetched again.

Don't use `useFetch` for one-off requests based on user actions though. It's best used for GET requests that are set up once when the component loads.

If you want, you can check out the documentation for more information.

39.

Using useAsyncData

The `useAsyncData` composable is a powerful composable provided by Nuxt that makes it easy to fetch data asynchronously in your components.

It synchronizes the state between our server and client, so during SSR it will only fetch the data on the server.

Here's an example of how you might use `useAsyncData` in a music production app to fetch a list of instruments:

```
<template>
  <div>
    <h1>Available Instruments</h1>
    <ul v-if="status === 'success'">
      <li v-for="instrument in instruments" :key="instrument.id">
        {{ instrument.name }}
      </li>
    </ul>
    <p v-if="status === 'pending'">Loading ...</p>
    <p v-if="status === 'error'">Error: {{ error.message }}</p>
  </div>
</template>
```

```
const { data: instruments, status, error } = useAsyncData(
  'instruments',
  () => fetch('https://api.example.com/instruments')
)
```

In this example, we're using `useAsyncData` to fetch the list of instruments and assign the result to a reactive `instruments` variable.

We also have access to `status` and `error` properties, which can be used to display loading and error states in our template.

You can check out the documentation for more information.

40.

Dedupe fetches

Since 3.9 we can control how Nuxt deduplicates fetches with the `dedupe` parameter:

```
useFetch('/api/search/', {
  query: {
    search,
  },
  dedupe: 'cancel'  // Cancel the previous request and make a new request
});
```

The `useFetch` composable (and `useAsyncData` composable) will re-fetch data reactively as their parameters are updated. By default, they'll cancel the previous request and initiate a new one with the new parameters.

However, you can change this behaviour to instead `defer` to the existing request — while there is a pending request, no new requests will be made:

```
useFetch('/api/search/', {
  query: {
    search,
  },
  dedupe: 'defer'  // Keep the pending request and don't initiate a new one
});
```

This gives us greater control over how our data is loaded and requests are made.

41.

The key to data fetching

The `key` parameter is an optional argument you can provide to the `useAsyncData` and `useFetch` composables to prevent extra requests:

```
const selected = ref(1)
const projects = [
  { id: 1, name: 'Project A' },
  { id: 2, name: 'Project B' },
  { id: 3, name: 'Project C' },
];

const { data: tracks, pending, error } = useAsyncData(
  'tracks',
  () => fetch(
    `https://api.example.com/projects/${selected.value}/tracks`
  )
);
```

Internally, the `key` parameter is used to create a unique identifier for the fetched data, so Nuxt knows if the data has already been fetched during the server render or not.

Here, the `key` parameter is “tracks”.

When the data is fetched on the server and passed along with the client bundle, the client knows it doesn’t need to re-fetch that data since it’s already been fetched.

If you *don’t* provide a key, Nuxt will automatically create one for you based on the line and file of where it’s used. However, it’s better to provide your own key, as this is more reliable.

42.

Prefetching Components

One downside of using `Lazy*` components is that they are only downloaded right when they are needed:

```
<!-- Loaded as soon as it needs to be rendered -->
<LazyCounter v-if="showCounter" />
```

The biggest benefit of these `Lazy*` components is when they're big, but that also means it takes longer to download the component and render to the page.

Instead, we can prefetch any global component at a time of our own choosing, using `prefetchComponents`:

```
await prefetchComponents('Counter');
```

Here's an example where we're doing both. If we click "Fetch Counter" we can preload the component, so it's already downloaded when the `v-if` is `true`:

```vue
<template>
  <div>
    <button v-if="!counterFetched" @click="fetchCounter">
      Fetch Counter
    </button>

    <!-- Will load Counter if not loaded already -->
    <button
      @click="
        showCounter = !showCounter;
        counterFetched = true;
      "
    >
      {{ showCounter ? 'Hide' : 'Show' }} Counter
    </button>

    <!-- Must use Lazy prefix or component will be loaded on page load -->
    <LazyCounter v-if="showCounter" />
  </div>
</template>

<script setup>
const showCounter = ref(false);
const counterFetched = ref(false);

async function fetchCounter() {
  counterFetched.value = true;
  // Can also use preloadComponents
  await prefetchComponents('Counter');
}
</script>
```

If we don't prefetch, the component will fall back to regular `Lazy*` behaviour and load as soon as the `v-if` becomes `true`.

Note: Even though the `Counter` component here is global (and async by default), you still have to use the `Lazy*` prefix or it will be included in the initial page render.

5 SSR Solutions

Tools for taking advantage of server-side rendering.

43.

Vue vs. Nitro Execution

In Nuxt, the concept of client side and server side execution gets a bit confusing due to the universal rendering (server-side rendering combined with client-side rendering).

Sometimes code is run on both the server *and* the client. Other times, only the client. Yet other times, only on the server.

The easiest to remember is that Nitro is our server framework, so any code here only runs on the server. This is everything inside of the `~/server` directory, including routes, middleware, and plugins.

Here, we have access to the `event` object that represents the `request` and `response` that we'll eventually give.

Our Vue app can be executed on *both* the server and client.

This includes components, composables, route middleware, pages, layouts, and plugins. Basically, everything not in the `~/server` directory. These are all executed in the context of the Vue and Nuxt app, and so we can access these if needed.

44.

Skip code on the server or client

To skip code being run on the server:

```
if (!import.meta.server) {
  // Don't run this on the server
}
```

To skip code from being run in the client:

```
if (!import.meta.client) {
  // Don't run this on the client
}
```

In your components though, there's a good chance you want to use `onMounted` to run client-only code:

```
onMounted(() => {
  // Code that only runs once the Vue component is
  // mounted on the client
});
```

Of course, any code within a client-only or server-only component will only be executed in those environments.

45.

Using useState for server rendered data

One use of `useState` is to manage state between the server-side and the client-side.

The first challenge is hydration.

By default, Vue doesn't do any hydration. If we're using `ref` or `reactive` to store our state, then during the server-side render they don't get saved and passed along. When the client is loading all our logic must run again.

But `useState` allows us to correctly sync state so we can reuse the work that was already done on the server.

The second challenge is what's known as *cross-request state pollution*.

If multiple users all visit your website at the same time, and those pages are all first rendered on the server, that means that the same code is rendering out pages for different people at the same time.

This creates the opportunity for the state from one request to accidentally get mixed up in another request.

But `useState` takes care of this for us by creating an entirely new state object for each request, in order to prevent this from happening. Pinia does this too, which is why it's also a recommended state management solution.

46.

NuxtClientFallback component

If you have an error during your server-side render, you can use the `<NuxtClientFallback>` component to render some fallback content:

```
<template>
  <NuxtClientFallback>
    <ServerComponentWithError />

    <template #fallback>
      <p>Whoops, didn't render properly!</p>
    </template>
  </NuxtClientFallback>
</template>
```

This is still experimental though, so you must have `experimental.clientFallback` set to `true` for this to work. Make sure to check out the docs for the latest info.

47.

SSR Safe Directives

In many cases, we need to generate unique IDs for elements dynamically.

But we want this to be stable through SSR so we don't get any hydration errors.

And while we're at it, why don't we make it a directive so we can easily add it to any element we want?

Here's a stripped-down version of this directive:

```
const generateID = () => Math.floor(Math.random() * 1000);

const directive = {
  getSSRProps() {
    return { id: generateID() };
  },
}
```

When using it with Nuxt, we need to create a plugin so we can register the custom directive:

```
// ~/plugins/dynamic-id.ts
const generateID = () => Math.floor(Math.random() * 1000);

export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.vueApp.directive("id", {
    getSSRProps() {
      return { id: generateID() };
    },
  });
});
```

In Nuxt 3.10+, you can also use the `useId` composable instead:

```vue
<template>
  <div :id="id" />
</template>

<script setup>
const id = useId();
</script>
```

Normally, custom directives are ignored by Vue during SSR because they *typically* are there to manipulate the DOM. Since SSR only renders the initial DOM state, there's no need to run them, so they're skipped.

But there are some cases where we actually need the directives to be run on the server, such as with our dynamic ID directive.

That's where `getSSRProps` comes in.

It's a special function on our directives that is *only* called during SSR, and the object returned from it is applied directly to the element, with each property becoming a new attribute of the element:

```vue
getSSRProps(binding, vnode) {
  // ...

  return {
    attribute,
    anotherAttribute,
  };
}
```

48.

SSR Payload

We can access the entire payload sent from the server to the client through `useNuxtApp`:

```
const nuxtApp = useNuxtApp();
const payload = nuxtApp.payload;
```

All of the data that's fetched from our data-fetching composables is stored in the `data` key:

```
const { data: pictures } = await useFetch('pictures', '/api/pictures');

const nuxtApp = useNuxtApp();
console.log(nuxtApp.payload.data.pictures) // ← our data is here
```

The key that is used is based on the `key` param passed in to the composable. By default, it will auto-inject the key based on the filename and line number, but you can pass in a custom value if needed.

Similarly, all the data that's stored for `useState` is in the `state` key, and can be accessed in the same way.

We also have a `serverRendered` boolean that let's us know if the response was server rendered or not.

6 Nitro Nuances

Never forget Nitro, the back end framework powering Nuxt.

49.

Creating server routes

The most basic version of a server route is this:

```
export default defineEventHandler(() => 'Not so complicated.');
```

We use `defineEventHandler` to create an event handler, and then directly export that so that Nuxt can use it.

If we need to do more with the request or response, we can do so by accessing the `event` parameter:

```
export default defineEventHandler((event) => {
  // Get the URL from the request
  const url = getReqeustUrl(event);

  // Specifically set response headers
  setResponseHeaders(event, {
    "content-type": "text/html",
    "cache-control": "no-cache",
  });
});
```

All of our server route files are placed in the `/server/routes` directory. Routing here works exactly like page routing — it's based on the filename. So if we put our server route in `/server/routes/hello.ts`, we can access it by sending a request to `/hello`.

However, Nuxt also gives us a shorthand for API routes since these are the most common type.

Any route placed in `/server/api` will automatically be prefixed with `/api`. If we place our event handler in `/server/api/hello.ts`, we can access it by sending a request to `/api/hello` instead.

50.

Server route return values

One of my favourite features is how h3 — the server that Nuxt uses internally — handles return values from `defineEventHandler`.

Instead of needing to properly set `content-type` headers and format our `Response` object correctly, we just return whatever we need to return:

```
// Return JSON without any extra work!
export default defineEventHandler(() => ({
  "name": "Chocolate",
  "type": "Ice Cream",
  "ingredients": ["Cream", "Sugar", "Cocoa Powder"],
  "serving_suggestion": "In a waffle cone with whipped cream
                         and a cherry on top",
  "calories": 250
}));
```

If we just need to send a status code, just return the status code:

```
// Ooooh status codes!
export default defineEventHandler(() => {
  // Do something
  something();

  // It all went well
  return 200;
});
```

You can also return a string:

```
export default defineEventHandler(() => {
  return '<p>Hi there!</p>';
});
```

If we return `null` then `h3` will return a `204 No Content` code:

```
export default defineEventHandler(() => {
  // 204 No Content
  return null;
});
```

All of these can be done async, either by wrapping in a `Promise` or using `async/await`.

We can also return a `Buffer`, a stream, or a standardized `Response` object.

But don't return errors. Instead, it's recommended to throw them using `createError` instead:

```
export default defineEventHandler(() => {
  if (!isAuthorized()) {
    throw createError({
      status: 401,
      message: 'Not authorized!',
    });
  }
});
```

51.

Getting data into server routes

There are a couple main ways we can get inputs or arguments passed into an event handler:

1. Route parameters
2. Query parameters
3. From the body of the request object

(We can also use cookies and headers and other values in some cases).

If we have a server route at `/server/api/icecream/[flavor].ts` we can use the `flavor` route parameter to dynamically return the right object. We access it using `getRouterParam`:

```
import { getRouterParam } from 'h3';
import { IceCreamFlavor } from './types';
import flavors from './flavors.json';

export default defineEventHandler(
  async (event): IceCreamFlavor | undefined => {
    // Grab the parameter from the route
    const flavor = getRouterParam(event, 'flavor');
    return flavors.find(
      flavor => flavor.name.toLowerCase() === name.toLowerCase()
    );
  }
);
```

We can use a query parameter instead if we change our route to this structure: `/server/api/icecream?flavor=chocolate`

Refactoring to use the `getQuery` method and renaming our file to `/server/api/icecream.ts` gives us this:

```
import { getQuery } from 'h3';
import { IceCreamFlavor } from './types';
import flavors from './flavors.json';

export default defineEventHandler(
  async (event): IceCreamFlavor | undefined => {
    // Grab the query from the route
    const { flavor } = getQuery(event);
    return flavors.find(
      flavor => flavor.name.toLowerCase() === name.toLowerCase()
    );
  }
);
```

We can also rewrite this handler to get the flavor from the body of the request using the `readBody` utility from h3:

```
import { getQuery } from 'h3';
import { IceCreamFlavor } from './types';
import flavors from './flavors.json';

export default defineEventHandler(
  async (event): IceCreamFlavor | undefined => {
    // Pass in the event object so we can parse out the body
    const { flavor } = await readBody(event);
    return flavors.find(
      flavor => flavor.name.toLowerCase() === name.toLowerCase()
    );
  }
);
```

If we send a request with a body of `{ 'flavor': 'chocolate' }` we'll see it return the chocolate flavor back to us!

52.

Reading a Request Body

Grabbing data from the body of a request can be done using the `readBody` method from `h3`:

```
import { readBody } from 'h3';  // Methods from h3 are also auto-imported

export default defineEventHandler(async (event) => {
   const body = await readBody(event);
});
```

This will parse the body into a JS object.

If we just want the raw string value, we can use `readRawBody` instead:

```
import { readRawBody } from 'h3';

export default defineEventHandler(async (event) => {
   const rawString = await readRawBody(event, 'utf-8');
});
```

We can also get a `Buffer` object if needed:

```
import { readRawBody } from 'h3';

export default defineEventHandler(async (event) => {
   const buffer = await readRawBody(event, false);
});
```

If we want to first validate the body we can do that with `readValidatedBody`:

```
import { readValidatedBody } from 'h3';

export default defineEventHandler(async (event) => {
   const validatedBody = await readValidatedBody(
     event,
     (parsedBody) => typeof parsedBody === 'object'
   );
});
```

We could also use a library like `zod` for more powerful parsing:

```
import { readValidatedBody } from 'h3';
import { z } from 'zod';

const schema = z.array(z.string());
export default defineEventHandler(async (event) => {
   const validatedBody = await readValidatedBody(
     event,
     schema.safeParse
   );
});
```

53.

Query Params in Server Routes

Getting values out of the query parameter in our server routes is straightforward:

```
import { getQuery } from 'h3';

export default defineEventHandler((event) => {
  const params = getQuery(event);
});
```

If we have the query `?hello=world&flavours[]=chocolate&flavours[]=vanilla` we'll get back the following `params` object:

```
{
  hello: 'world',
  flavours: [
    'chocolate',
    'vanilla',
  ],
}
```

We can also use a validator function with `getValidatedQuery`:

```
import { getValidatedQuery } from 'h3';

export default defineEventHandler((event) => {
  const params = getValidatedQuery(
    event,
    obj => Array.isArray(obj.flavours)
  );
});
```

54.

Getting request headers

Grabbing a single header from the request couldn't be easier in Nuxt:

```
const contentType = useRequestHeader('content-type');
```

We sometimes need to check the request headers on the server (in Nitro) for authentication, security, caching, or rate limiting, and more.

If you're in the browser though, it will return `undefined`.

This is an abstraction of `useRequestHeaders`, since there are a lot of times where you need just one header. But you can also use that one to grab them all:

```
const headers = useRequestHeaders();
```

See the docs for more info.

55.

Server Routes and HTTP Methods

We can prevent our server routes from being run on the wrong HTTP method by specifying the correct one in the file name, like `~/server/api/users.get.ts`.

If we try to hit the `/api/users` endpoint with a `POST` request, we'll get back a `405 Method Not Allowed` error.

This let's us split up logic for different methods into different files, instead of having all the logic in a single file.

But if we want it all in one file, we can do a check to see which method we're dealing with by looking at `event.method`:

```
export default defineEventHandler((event) => {
  if (event.method === 'GET') {
    // Retrieve the user
  } else if (event.method === 'POST') {
    // Create a user
  } else if (event.method === 'DELETE') {
    // Delete the user
  } else if (event.method === 'PATCH' || event.method === 'PUT') {
    // Update the user
  };
});
```

We can also use `assertMethod` to return a `405` status code:

```
import { assertMethod } from 'h3';

export default defineEventHandler((event) => {
  assertMethod(event, 'GET');

  // Return some data
});
```

If this endpoint is called with anything other than a `GET` request, it'll return a `405` immediately.

56.

Server Middleware

Server middleware are functions that are run on *every* request before our server routes handle them. This makes them great for inspecting or modifying the request in some way:

```
// ~/server/middleware/auth.ts

export default defineEventHandler((event) => {
  const user = useAuthenticatedUser(event);

  event.context.user = user;
  event.context.authenticated = user.isAuthenticated;
});
```

The `context` property on the `event` object is there for us to add to, so we can provide additional context if needed.

Now, we can use these values in server routes:

```
// ~/server/api/user.get.ts

export default defineEventHandler((event) => {
  if (!event.context.authenticated) {
    throw createError({
      statusCode: 401,
      statusMessage: 'Unauthorized'
    });
  }

  // ...
});
```

You'll notice that both middleware and server routes use `defineEventHandler`. The only real difference with middleware is that they are executed on every request, and always executed before the server route itself.

57.

Server-side redirects

To create a redirect in a Nuxt server route, we can use the `sendRedirect` method from `h3`:

```
import { sendRedirect } from 'h3';

export default defineEventHandler((event) ⇒
  // Redirect on the server. Uses a 302 (not 307) by default.
  sendRedirect(event, "https://www.jsdocs.io/package/h3#sendRedirect"),
);
```

By default, it will use a `302` (temporary redirect) status code. But we can make it use a `307` or any other status code if we prefer:

```
import { sendRedirect } from 'h3';

export default defineEventHandler((event) ⇒
  // Redirect on the server. Uses a 302 (not 307) by default.
  sendRedirect(event, "https://www.jsdocs.io/package/h3#sendRedirect", 307),
);
```

Creating redirects in server middleware works exactly the same as with server routes — we're still in the Nitro and h3 server. The main difference is that server middleware are run on *every* request we get, so they're effectively global. This means we just need to be more careful with our business logic.

We'll create a file in `server/middleware/redirect.ts`:

```ts
import { getRequestURL, sendRedirect } from "h3";

export default defineEventHandler((event) => {
  const url = getRequestURL(event);

  // Always redirect on the server-side
  if (url.pathname === "/server-middleware") {
    return sendRedirect(event, "https://michaelnthiessen.com");
  }
});
```

Whenever we try to go to `/server-middleware`, the middleware will intercept that request and redirect us to my website instead.

The actual route handler for `/server-middleware` won't get executed, but it *does* need to exist or the router will give us an error. If we instead did this in a route middleware, the actual route doesn't need to exist since the redirect happens *inside* the router itself.

58.

Understanding Universal Rendering

Nuxt offers a unique solution to the limitations of SPAs and SSRs by combining their strengths. This approach, called Universal Rendering, provides the best of both worlds.

1. Lightning Fast First Page Load

On the first page load, Nuxt uses SSR to deliver a fast initial experience.

It processes the request on the server and sends back the HTML and other necessary files, similar to a traditional SSR app. This ensures that users are not kept waiting, which is particularly important for maintaining user engagement and optimizing search engine rankings.

2. Seamless Transition to SPA

However, Nuxt doesn't stop at the initial SSR.

It also loads the entire app as an SPA, so everything after the first page load is extremely quick. Once the initial page is loaded, Nuxt switches to SPA mode, allowing users to navigate within the app without needing to make round trips to the server.

3. Performance Enhancements and Optimizations

It's worth noting that Nuxt goes beyond simply combining SSR and SPA approaches. The framework also includes numerous performance enhancements and optimizations under the hood.

For example, Nuxt ensures that only necessary data is sent, and it intelligently prefetches data right before it's needed.

These optimizations, along with many others, contribute to the overall speed and efficiency of a Nuxt application. In essence, Nuxt provides a seamless user experience without sacrificing performance, offering developers the best of both SPA

and SSR worlds.

59.

Cookies and SSR

Sometimes we'll need to make sure to pass cookies in to our server routes:

```
const { data, error } = await useFetch(protectedUrl, {
  headers: useRequestHeaders(['cookie']),
});
```

We often need to do this when we use authentication on our server routes, since the user's session is often stored in a cookie. On the client side this works fine, as cookies are included with each request. But during server-side rendering the cookie isn't passed around.

If we don't pass that cookie along though, our server route doesn't know the user is logged in and blocks the request.

We can solve this issue by always passing along the cookie using the `useRequestHeaders` composable and modifying how we use `useFetch` or `useAsyncData`. Now, during SSR, `useRequestHeaders` will pass the cookie along with the request, so we can successfully access the protected endpoint.

60.

Building a Basic Link Shortener

If we wanted to make a basic link shortener service, we can do that pretty easily using a server route.

We'll create this file at `server/route/link/[hash].ts`:

```
import { getRouterParam, sendRedirect } from "h3";

// Mock database
// We would normally get this data from a database somewhere
const db = {
  h4sh: "https://www.nuxt.com",
  "an0th3r-h4sh": "https://michaelnthiessen.com",
};

export default defineEventHandler((event) => {
  const hash = getRouterParam(event, "hash");
  const redirectUrl = db[hash];

  return sendRedirect(event, redirectUrl, 307);
});
```

We'll set up a mock database for simplicity, and then look up the correct URL to redirect to based on the `hash` parameter in the route. We tell the router that we want to get a route param by using square brackets in the filename, that's what the `[hash]` part is for.

To get this value from the URL, we use the `getRouterParam` method from `h3`. Then, we can use that to lookup what the redirect URL should be. Here we use a plain old Javascript object for simplicity.

After that, we use the `sendRedirect` method from `h3` to redirect the client using a `307` temporary redirect.

61.

Built-in Storage with Unstorage

Nitro, the server that Nuxt uses, comes with a very powerful key-value storage system:

```
const storage = useStorage();

// Save a value
await storage.setItem('some:key', value);

// Retrieve a value
const item = await storage.getItem('some:key');
```

It's not a replacement for a robust database, but it's perfect for temporary data or a caching layer.

One great application of this "session storage" is using it during an OAuth flow.

In the first step of the flow, we receive a `state` and a `codeVerifier`. In the second step, we receive a `code` along with the `state` again, which let's us use the `codeVerifier` to verify that the `code` is authentic.

We need to store the `codeVerifier` in between these steps, but only for a few minutes — perfect for Nitro's storage!

The first step in the `/oauth` endpoint we store the `codeVerifier`:

```
// ~/server/api/oauth

// ...
const storage = useStorage();
const key = `verifier:${state}`;
await storage.setItem(key, codeVerifier);
// ...
```

Then we retrieve it during the second step in the `/callback` endpoint:

```
// ~/server/api/callback

// ...
const storage = useStorage();
const key = `verifier:${state}`;
const codeVerifier = await storage.getItem(key);
// ...
```

A simple and easy solution, with no need to add a new table to our database and deal with an extra migration.

This just scratches the surface. Learn more about the `unstorage` package that powers this: https://github.com/unjs/unstorage

7 Configuration Coordination

Dive into discovering different and diverse configuration options.

62.

Auto Imports

Instead of importing all of your dependencies like this:

```
// Part of my blog
import BasicLayout from './BasicLayout.vue';
import Footer from '../components/Footer';
import Subscribe from '../components/Subscribe';
import LandingMat from '../components/LandingMat';
import Logo from '../icons/Logo';
import LogoClip from '../icons/LogoClip';
import TriangleShape from '../icons/TriangleShape';
import SquareShape from '../icons/SquareShape';
```

You import them like this:

```
// ...just kidding. No imports needed!
```

Just use your components, composables, or layouts where you need them, and Nuxt takes care of the rest.

It may seem like a small thing, but auto-imports in Nuxt make the whole developer experience so much nicer. It only imports what you need, when you need it.

This makes your app much faster as well!

Yes, your dependencies are now less explicit. But if you keep your components and composables small enough it shouldn't matter that much. You should still be able to see pretty quickly what's going on in your application.

63.

Disable auto-imports

You can completely disable auto-imports by using the new `imports.scan` option:

```
export default defineNuxtConfig({
  imports: {
    scan: false,
  },
});
```

This will ignore any directories listed in the `imports.dirs` option, as well as ignoring auto-imports for the `~/composables` and `~/utils` directories.

To also prevent auto-importing components, you'll need to configure the `components` option as well:

```
export default defineNuxtConfig({
  imports: {
    scan: false,
  },
  // Set to an empty array
  components: [],
});
```

This will make the imports in your project more explicit, so it's clearer where things are coming from. Some developers prefer this because it adds to the readability of the code.

64.

Environment Config Overrides

We often need different config based on the environment our app is running in, whether we're running tests, in dev mode, or running in prod.

But instead of doing this:

```
export default defineNuxtConfig({
  modules: process.env.NODE_ENV === 'development'
    ? ["@nuxtjs/html-validator"],
    : []
});
```

We have a better, more type-safe way of doing it:

```
export default defineNuxtConfig({
  $development: {
    modules: ["@nuxtjs/html-validator"]
  }
});
```

We can use the `$test`, `$development` and `$production` keys for environment-specific configs.

65.

Tree Shake Composables

You can specify composables to be tree shaken out of the client bundle (or server bundle):

```
// nuxt.config.ts
optimization: {
  treeShake: {
    composables: {
      client: {
        'package-name': ['useSomeUnnecessaryComposable'],
      }
    },
  },
 },
```

By default, Nuxt tree shakes composables from Vue that don't work on the server or client (many work in both places) and composables built-in to Nuxt.

For example, if you never use the `useId` composable, it won't be included with your app bundle, so it will load just that much faster.

66.

.nuxtignore

Using the same syntax as your `.gitignore` file, you can configure Nuxt to ignore files and directories during build time:

```
# Ignore specific files
**/*ignore-me.vue

# Ignore a whole directory
components/ignore/
```

67.

Global CSS

You can include CSS files in every single page using the `css` config value in `nuxt.config.ts`:

```
export default defineNuxtConfig({
  css: ['~/assets/css/global.css'],
});
```

It will also use the right pre-processor, but you have to make sure that it's installed:

```
export default defineNuxtConfig({
  css: ['~/assets/sass/global.scss'],
})
```

This is handy for applying consistent styling, like including your own utility classes (like Tailwind does), or CSS resets so each page has a clean starting point.

68.

Pre-render Some Routes

By configuring Nitro directly, we can have only *some* routes pre-rendered.

Every other route will use hybrid rendering, like normal:

```
export default defineNuxtConfig({
  nitro: {
    prerender: {
      routes: ['/about', '/blog'],
    },
  },
});
```

Or you could choose specific routes to ignore:

```
export default defineNuxtConfig({
  nitro: {
    prerender: {
      // Start at `/` and follow all links
      crawlLinks: true,
      ignore: ['/admin/**'],
    },
  },
});
```

69.

Using appConfig

The `app.config` is used to expose public variables that can be determined at build time, such as theme variants, titles, or other non-sensitive project configurations. These values are set in the `app.config.ts` file.

To define `app.config` variables, you need to create the `app.config.ts` file in the root of your project:

```
// app.config.ts

export default defineAppConfig({
  theme: {
    primaryColor: '#ababab'
  }
})
```

To access `app.config` values within your application, you can use the `useAppConfig` composable:

```
const appConfig = useAppConfig()
```

Although the `appConfig` type is automatically inferred, you can manually type your `app.config` using TypeScript if you *really* need to. This example is from the docs:

```
// index.d.ts
declare module 'nuxt/schema' {
  interface AppConfig {
    // This will entirely replace the existing inferred `theme` property
    theme: {
      // You might want to type this value to add more specific types
      // than Nuxt can infer, such as string literal types
      primaryColor?: 'red' | 'blue'
    }
  }
}
```

If you're writing a module that needs config, you can also provide a type for your module.

70.

Using runtimeConfig

The `runtimeConfig` is used to expose environment variables and private tokens within your application, such as API keys or other sensitive information. These values can be set in the `nuxt.config.ts` file and can be overridden using environment variables.

To set up private and public keys in your `nuxt.config.ts` file, you can use the following code example:

```
export default defineNuxtConfig({
  runtimeConfig: {
    // The private keys which are only available server-side
    shoeStoreApiSecret: 'my-secret-key',

    // Keys within public are also exposed client-side
    public: {
      shoeStoreApiBase: '/shoe-api'
    }
  }
})
```

To access `runtimeConfig` values within your application, you can use the `useRuntimeConfig` composable:

```
const { shoeStoreApiBase } = useRuntimeConfig();
console.log(shoeStoreApiBase); // /shoe-api
```

Note that you can't access a private key on the client-side:

```
const { shoeStoreApiSecret } = useRuntimeConfig();
console.log(shoeStoreApiSecret); // undefined
```

But you can access *all* values in a server route:

```
export default defineEventHandler(async (event) => {
  const { shoreStoreApiSecret } = useRuntimeConfig();
  console.log(shoeStoreApiSecret); // my-secret-key
});
```

You can set environment variables in a `.env` file to make them accessible during development and build/generate.

Just make sure that you use the right prefixes. Put `NUXT_` before everything so that Nuxt will know to import it, and don't forget to add in `PUBLIC` if it's a value in the `public` field of your config:

```
NUXT_PUBLIC_BOOK_STORE_API_BASE_URL = "https://api.bookstore.com"
NUXT_BOOK_STORE_API_SECRET = "super-secret-key"
```

71.

Where should config values go?

I think about it this way:

- **runtimeConfig**: Use runtimeConfig for private or public tokens that need to be specified after the build using environment variables
- **app.config**: Use app.config for public tokens that are determined at build time, such as website configuration (theme variant, title) or any project config that are not sensitive

Both `runtimeConfig` and `app.config` allow you to expose variables to your application. However, there are some key differences:

1. `runtimeConfig` supports environment variables, whereas `app.config` does not. This makes `runtimeConfig` more suitable for values that need to be specified after the build using environment variables.
2. `runtimeConfig` values are hydrated on the client side during run-time, while `app.config` values are bundled during the build process.
3. `app.config` supports Hot Module Replacement (HMR), which means you can update the configuration without a full page reload during development.
4. `app.config` values can be fully typed with TypeScript, whereas `runtimeConfig` cannot.

72.

Using /public

Any files that you put into the `~/public` directory get included directly in the build output of your Nuxt app. They are not modified in any way.

This public directory is served at the root of your app. This means that if your `~/public` folder looks like this:

```
public/
- cats.png
- songs/
  - dubstep.wav
  - chopin.mp3
```

You can access these files directly from your browser:

```
yourwebsite.com/cats.png
yourwebsite.com/songs/dubstep.wav
yourwebsite.com/songs/chopin.mp3
```

Remember, these files are not touched at all. The contents of `~/public` are copied straight into your `dist` folder.

These are common things we put in the `~/public` folder:

- `robots.txt`
- `favicon`
- sitemaps

But generally, we'll want to keep most things as assets.

73.

Using /assets

All of the code in your application is processed by a bundler. Nuxt uses Vite by default.

The bundler will start at `app.vue` and pull in all of its dependencies, then all of *their* dependencies, and so on, until all of the code in your app has been processed.

This becomes the bundled output that you can use to deploy your application.

If you import something other than "code" into a component, the bundler still has to include those files as well. Otherwise, they wouldn't be deployed alongside all of your other code:

```
import '~/assets/styles/radStyleSheet.css';
```

When you do this, the bundler will do a few things:

1. Process the file with any configured plugins
2. Generate the bundled file with the file's hash in the name (for caching purposes as we mentioned earlier)
3. Replace the `import` with this new filename so your bundled code can find the file when it's deployed

It doesn't actually matter where you put these files. But we have to put all of these other assets *somewhere* to keep things organized, so we typically use `~/assets` by convention.

Most things usually end up being processed by the bundler, so you'll commonly find these in an `~/assets` folder:

- Stylesheets
- Icons
- Fonts

Images can also go in there if you need to process them with a Vite plugin. But it's better to keep them in `/public` and use `NuxtImg` to process and cache them instead.

You can learn more about managing assets from the Nuxt docs.

74.

Where do you put that asset?

In any web app, including with Nuxt, we have two main ways to deal with our non-code assets:

1. We can process them through our bundler using the `/assets` directory (or importing from somewhere else)
2. We can leave them as is inside the `/public` directory

But how do you decide?

There are three questions I use to help me make this decision. If the answer is yes to any, put it through your bundler.

1. Does this need to be processed in some way?

If you have a bunch of images, you likely want to make sure those images are compressed and optimized before shipping them with your web application.

A sitemap, on the other hand, is good as it is. No need to do anything extra with it.

2. Does this asset change often, or not at all?

Our code changes often, so we want to include it in our bundle. This is because our bundler will give our code bundle a unique name every time it changes, based on a hash of the contents.

This means the browser will be forced to always have the latest version of our code.

3. Does the filename matter?

Page crawlers will be looking for a file specifically called `robots.txt`, so we need to preserve that filename.

For most other assets though, the specific filename doesn't matter so much, so we

can process those files through the bundler.

8 Layer Lollapalooza

Better organize your Nuxt app with layers.

75.

Layer Basics

Layers are like components, but for Nuxt apps.

They let us break down, and then combine together, smaller Nuxt apps so we can better organize, share, and modularize our code.

Unlike a more typical `npm` package, these layers benefit from all of the Nuxt magic: auto-imports, file-based routing, Nuxt configs, and more.

A layer can be in a local folder, a remote Github repository, or in a `npm` package. We just need to make sure it has a `nuxt.config.ts` file so Nuxt knows it's a layer, even if that config is left empty:

```
// layer/nuxt.config.ts
export default defineNuxtConfig({});
```

76.

Importing Between Layers

If we have two sublayers, `blue` and `green`, these layers can use components (and other files) from each other without extending each other:

```
// Blue.vue
<template>
  Use a component from the Green layer.
  <GreenButton />
</template>
```

This is because they are both imported by the "main" app or top layer.

When `Blue.vue` is being rendered, the `GreenButton` exists because it was also imported from the sublayer, so it all works.

This is useful if you're splitting your app into layers for organizational purposes.

77.

Naming Collisions Between Layers

One issue with importing all sorts of components, layouts, and other things from layers, is naming collisions. If we have a `Button` component in each layer, that leads to some conflicts.

Here's how the names get resolved:

- The current layer (or app) takes precedence
- Then, the first layer defined in the `extends` in your `nuxt.config.ts`

Because of how layers work, this *also* applies within your sub layers if they use a component that has a naming collision.

Let's say that we have a `Button` component defined in both the `blue` and `green` layers, as well as in our current layer:

```
- components
  - Button.vue
- green
  - components
    - Button.vue
- blue
  - components
    - Button.vue
```

Now we create a `BlueComponent` at `blue/components/BlueComponent.vue`:

```
<template>
  Please click this button:
  <Button>Click me!</Button>
</template>
```

This won't render the `Button` from the `blue` layer. Component precedence doesn't change regardless of which layer you're in, so it will still render the `components/Button.vue` because the "top" layer takes precedence.

However, we can solve this by using a relative import, which *will* use the current layer:

```vue
<template>
  Please click this button:
  <BlueButton>Click me!</BlueButton>
</template>

<script setup>
import BlueButton from '~/components/Button.vue';
</script>
```

78.

Pages from Layers

Pages from sub layers create routes just as they would if we were using the layer as its own app. It doesn't prepend any path or anything like that:

```
- pages
  - index.vue    /
  - nested.vue   /nested
- blue
  - pages
    - blue.vue   /blue
- green
  - pages
    - green.vue  /green
```

But this gives us some interesting opportunities, since all pages are treated equally.

We can have layers provide child routes (or parent routes, it doesn't matter), and use them like any other routes:

```
- pages
  - index.vue    /
  - nested.vue   /nested
- blue
  - pages
    - nested
      - blue.vue   /nested/blue
- green
  - pages
    - nested
      - green.vue  /nested/green
```

We need to make sure that our `nested.vue` component renders a child route using `NuxtPage`:

```
<template>
  This is in the main app:
  <NuxtPage />
</template>
```

Now, if we navigate to `/nested/blue` we'll render out `nested.vue` as the parent, and `blue.vue` as the child component.

79.

Separate Config Per Layer

There are *lots* of configuration options available in Nuxt. In fact, your `nuxt.config.ts` might start getting really long and hard to read!

But with layers, not only is our *code* separated, but our *config* is as well.

Our blog layer can use the Nuxt Content module, but leave it out of the main app:

```
export default defineNuxtConfig({
  modules: [
    '@nuxt/content'
  ],
  content: {
    // ... options
  },
})
```

We can define a more focused `runtimeConfig` in each layer, keeping things simpler.

We can disable auto-imports of our components in one layer:

```
export default defineNuxtConfig({
  components: {},
});
```

80.

Private Components in Layers

What if we want a layer that has pages and layouts, but we want the components to be private and not available to the main app?

We can do that by giving that layer a custom configuration:

```
// privateLayer/nuxt.config.ts
export default defineNuxtConfig({
  components: [],
});
```

By removing all auto-imports from components, we can make these components "private". The only way to use them now is to manually import them based on their filepath.

9 Module Mechanics

Many tips on the best way to extend your Nuxt app.

81.

Official vs Community Modules

There are *lots* of modules for Nuxt, and there is a useful naming convention that helps to understand what kind of module you're dealing with.

- Official modules — these are always prefixed with `@nuxt/` and are actively maintained by the Nuxt team. For example, `@nuxt/image`, `@nuxt/content`, and `@nuxt/fonts`.
- Community modules — these modules are maintained by the community but have been "curated" by the Nuxt team. They are prefixed with `@nuxtjs/`. For example, `@nuxtjs/tailwindcss`, `@nuxtjs/color-mode`, and `@nuxtjs/supabase`.
- Third party modules — these are usually prefixed with `nuxt-` to indicate they are Nuxt-specific packages, and can be made by anyone. For example, `nuxt-layers-utils` and `nuxt-content-assets`.

You can find a searchable list of official and community modules on the Nuxt website.

If you've created a module and want it included, you can open a PR on the Nuxt Modules Github repo. You can also find all the other community modules there if you want to help with maintaining one of them!

82.

Creating Local Modules

The most powerful way to encapsulate custom Nuxt functionality is through *modules*. You don't have to share them either, you can create local modules just for your app inside of `~/modules`:

```
// ~/modules/myModule.ts

export default function (options, nuxt) {
  // Update your build here
}
```

The files inside of `~/modules` are auto-imported one-level deep, so we don't need to do anything more.

However, it's best to use the `defineNuxtModule` wrapper, because it adds a bunch of extra stuff to make the modules work better, like specifying compatibility.

Even when just developing our own module, we get some benefits, like better type hints and declarative hooks:

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

export default defineNuxtModule({
  hooks: {
    'app:rendered'() {
      console.log('Rendered the app!');
    },
  },
  setup(moduleOptions, nuxt) {
    // ...
  },
});
```

Modules are best when we need to modify build behaviour of our app, or add runtime files and assets during build. We can also do a lot of this stuff through layers, but without programmatic flexibility.

If we just need to update some runtime behaviour, plugins are the way to go.

83.

Module runtime directory

Modules can only affect your Nuxt app during build time. However, we can use a module to change the runtime behaviour by changing our app at build time.

We can add auto-imported components by using `addComponentsDir`:

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

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

    // Make all of these components async all the time
    addComponentsDir({
      path: resolver.resolve('./runtime/components/'),
      prefix: 'Async',
      isAsync: true,
    });
  },
});
```

Our module will now load in all components in the `~/plugins/runtime/components` directory, making them async by default (loading in their own chunk). We use them by adding the `Async*` prefix to their name instead of the `Lazy*` prefix that Nuxt gives us by default.

You can add pages using `extendPages`, or update routing using `extendRouteRules` or `addRouteMiddleware`.

In fact, there are methods in `@nuxt/kit` that let you modify almost any part of your Nuxt app at build time.

84.

Module Hooks

We get access to a special set of build-time and development time hooks in our modules, which let us modify how Nuxt works:

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

export default defineNuxtModule((options, nuxtApp) => {
  nuxtApp.hooks.hook('builder:watch', (event, path) => {
    console.log('Something changed!:', event, path);
  });
});
```

For example, using the `builder:watch` hook we can track all the updates being made to the project during development. Every time a file in our Nuxt project is created/updated/deleted, this hook is called.

Here are a bunch of interesting hooks:

- `prerender:routes` so you can extend the routes to be prerendered
- `vite:extendConfig` and `webpack:config` allow you to modify your bundler's config
- `prepare:types` allows you to modify the `.nuxt/tsconfig.json` and `.nuxt/nuxt.d.ts` files before they are written to disk
- `build:error` lets you respond to build errors

Go here for the full list of build time hooks.

10 Plugin Proficiency

Deftly modify the runtime behaviour of Vue and Nitro.

85.

Plugin Basics

We can modify the runtime behaviour of our Nuxt app by creating plugins inside the `~/plugins/` directory:

```
export default defineNuxtPlugin((nuxtApp) => {
  // Define behaviour here
});
```

We can also use an object syntax that gives us more flexibility and expressivity:

```
export default defineNuxtPlugin({
  name: 'custom-plugin',
  async setup (nuxtApp) {
    // Define behaviour here
  }
});
```

For example, instead of defining hooks in the `setup`, we can define them declaratively:

```
export default defineNuxtPlugin({
  name: 'custom-plugin',

  async setup (nuxtApp) {
    // We can define it in the setup
    nuxtApp.hooks.hook('vue:error', (err) => {
      // Handle the Vue error
      console.error(err);
    });
  },

  hooks: {
    // Define the hook declaratively
    'vue:error'(err) {
      // Handle the Vue error
      console.error(err);
    },
  });
});
```

When using the object syntax, Nuxt will statically analyze and pre-load the plugins. This lets it optimize the build, and means that you don't need to worry about plugin ordering when defining hooks in this way.

86.

What's the deal with all these hooks?

Nuxt comes with a lot of different hooks, but I've found myself quite confused about what hooks are available when, and where we can use them.

There are three main types:

1. Runtime hooks: These are available during the Vue rendering lifecycle and can be accessed by Nuxt plugins and inside of the Vue context (components and composables).
2. Build hooks: These are available to modules.
3. Server hooks: These are only available to server plugins and hook into Nitro directly.

We can also access server hooks in our modules by using the `nitro:init` hook to wait for nitro to be initialized.

87.

Nitro Plugins

You can write custom plugins to modify Nitro's runtime behaviour. They live inside of `~/server/plugins`:

```
// ~/server/plugins/handleError.js

// Setup custom error logging
const logger = MyLogger();

export default defineNitroPlugin((nitroApp) => {
  // Register a hook
  nitroApp.hooks.hook("error", async (error, { event }) => {
    logger.error(error, event);
    console.error('Whoops!', error);
  });
})
```

You have access to the `nitroApp`, which lets you hook into specific runtime hooks that Nitro provides:

- `close` — when Nitro is closing down
- `error` — when an error is encountered
- `render:response` — after Nitro has finished server rendering a page
- `request` — right after a new request has been received
- `beforeResponse` — right before sending a response
- `afterResponse` — right after sending a response

Nuxt also defines some additional hooks on the Nitro app we can use:

- `dev:ssr-logs` — contains all the server-side logs generated from that request
- `render:html` — right before Nuxt server renders the page
- `render:island` — right before Nuxt server renders an island component

88.

Nitro Hooks

There are two types of hooks available on the server: server hooks that come from Nitro, and server hooks that Nuxt adds to Nitro.

(This is why the list of hooks in the Nuxt and Nitro docs are different in case you were wondering).

We can hook directly into Nitro's behaviour by using hooks inside of Nitro plugins:

```
export default defineNitroPlugin((nitro) => {
  // Log out the logs from the server-side rendering
  nitro.hooks.hook('dev:ssr-logs', (logs) => {
    console.log('Custom log:', logs);

    // Or write to a file somewhere...
  });

  // Add some HTML to each rendered page
  nitro.hooks.hook('render:html', (html) => {
    html.bodyAppend.push(
      '<script>window.myValue = "add this to each page";</script>'
    );
  });
});
```

These Nitro plugins are created in the `/server/plugins` folder.

Here's the full list of Nitro hooks and the list of Nitro hooks available in Nuxt (including ones added by Nuxt).

89.

Parallel Plugins

You can set up a plugin to load in parallel, so it doesn't block the loading of other plugins:

```
export default defineNuxtPlugin({
  name: 'some-parallel-plugin',
  parallel: true,
  async setup (nuxtApp) {
    // Do some heavy work here
  }
});
```

This will immediately start loading and executing the next plugin in the plugin list, while this one loads asynchronously. Here is some pseudo code that explains how this works:

```
for (plugin in pluginList) {
  if (plugin.parallel) {
    // No `await` so we don't wait for it to finish before moving on
    plugin.setup();
  } else {
    // We wait for the plugin to finish setting up
    await plugin.setup();
  }
}
```

90.

Nuxt Plugin Dependencies

When writing plugins for Nuxt, you can specify dependencies:

```
export default defineNuxtPlugin({
  name: 'my-sick-plugin-that-will-change-the-world',
  dependsOn: ['another-plugin']
  async setup (nuxtApp) {
    // The setup is only run once `another-plugin` has been initialized
  }
})
```

But why do we need this?

Normally, plugins are initialized sequentially — based on the order they are in the filesystem:

```
plugins/
- 01.firstPlugin.ts    // Use numbers to force non-alphabetical order
- 02.anotherPlugin.ts
- thirdPlugin.ts
```

But we can also have them loaded in parallel, which speeds things up if they don't depend on each other:

```
export default defineNuxtPlugin({
  name: 'my-parallel-plugin',
  parallel: true,
  async setup (nuxtApp) {
    // Runs completely independently of all other plugins
  }
})
```

However, sometimes we have *other* plugins that depend on these parallel plugins. By using the `dependsOn` key, we can let Nuxt know which plugins we need to wait for, even if they're being run in parallel:

```ts
export default defineNuxtPlugin({
  name: 'my-sick-plugin-that-will-change-the-world',
  dependsOn: ['my-parallel-plugin']
  async setup (nuxtApp) {
    // Will wait for `my-parallel-plugin` to finish before initializing
  }
})
```

91.

Server Only (and Client Only) Plugins

You can split up plugin logic based on client-side or server-side by using the correct suffixes:

- Client-side: `~/plugins/*.client.ts`
- Server-side: `~/plugins/*.server.ts`

You can also set this in `nuxt.config.ts` by setting the `src` and `mode` when listing your plugins:

```
plugins: [
  '@nuxt/content',
  {
    src: '~/plugins/some-plugin.ts',
    mode: 'client',
  },
]
```

Remember, any plugins inside of `~/plugins` will be auto-registered and loaded *after* the plugins listed here in the config. They will default to being isomorphic, running on both client and server side.

11 Server Components

No JS? No problem! At least, for these components.

92.

NuxtIsland Component

Nuxt uses the `NuxtIsland` component internally for server components and pages, but you can use it directly if you need to:

```
<NuxtIsland
  name="Counter"
  :props="{
    startingCount,
  }"
/>
```

The `name` is the same name you'd use in a template to use an auto-imported component (the component needs to be `global`). This means a component at `~/components/server/Counter.server.vue` would be used like this with the filename prepended:

```
<NuxtIsland
  name="ServerCounter"
  :props="{
    startingCount,
  }"
/>
```

This component *must* be a server component though! The easiest way to do this is by putting the `*.server.vue` suffix on it.

Whenever the props of this component change on the client, a new request will be made to the server to re-render the component. You can also force this behaviour by using the `refresh` method on the `ref`:

```vue
<template>
  <NuxtIsland
    name="Counter"
    ref="serverCounter"
    :props="{
      startingCount,
    }"
  />
</template>

<script setup>
const serverCounter = ref(null);

async function forceRefresh() {
  if (!serverCounter.value) return;
  await serverCounter.value.refresh();
}
</script>
```

However, keep in mind that each `NuxtIsland` component is rendered as a full Nuxt app on the server. So having multiple of these on a single page can lead to performance issues.

You can read more about this component in the docs.

93.

Interactive Components Within Server Components

If you want to add an interactive client-side component *inside* of your server component, you can do that by adding on the `nuxt-client` attribute:

```
// Parent.server.vue
<template>
  <div>
    <div>This is on the server</div>

    <!-- Will become interactive on the client -->
    <Counter nuxt-client :starting-count="39" />
  </div>
</template>
```

Because server pages are also server components, you must use this attribute on server pages as well.

94.

Paired Server Components

You can write more complex components that have different logic on the server and the client by splitting them into paired server components. All you need to do is keep the same name, changing only the suffix to `*.client.vue` and `*.server.vue`.

For example, in our `Counter.server.vue` we set up everything we want to run during SSR:

```
<template>
  <div>This is paired: {{ startingCount }}</div>
</template>

<script setup lang="ts">
withDefaults(defineProps<{ startingCount: number }>(), {
  startingCount: 0,
});
</script>
```

We grab our `startingCount` prop and render it to the page — no need to do anything else because we're not interactive at this point.

Then, Nuxt will find `Counter.client.vue` and ship that to the client in order to hydrate and make the component interactive:

```vue
<template>
  <div>This is paired: {{ count }}</div>
</template>

<script setup lang="ts">
const props = withDefaults(
  defineProps<{ startingCount: number }>(),
  {
    startingCount: 0,
  }
);

const offset = ref(0);
const count = computed(
  () => props.startingCount + offset.value
);

onMounted(() => {
  setInterval(() => {
    offset.value++;
  }, 1000);
});
</script>
```

We're careful to make sure we avoid hydration mismatches, and we bootstrap our interactivity.

A nice feature to improve separation of concerns where you need it!

95.

Slots and Server Components

You can put interactive components into the slot of server components, as long as the parent component is also interactive:

```
// Parent.client.vue
<template>
  <ServerComponent>
    <!-- We don't need nuxt-client attr here -->
    <Counter :starting-count="39" />
  </ServerComponent>
</template>
```

This works because of the way that slots decouple the Vue component tree from the tree rendered to the DOM. Here, `Counter` is a child of `Parent` in Vue tree, although when rendered it becomes a child of the `ServerComponent`:

```
Vue component tree:
- Parent
  - ServerComponent
  - Counter

Tree rendered to the DOM:
- Parent
  - ServerComponent
    - Counter
```

This means that it behaves normally, and the `ServerComponent` doesn't really affect how it is rendered.

However, if the slot content is being rendered in the context of a server component, it won't become interactive:

```
// ServerComponent.server.vue
<template>
  <div>
    <Counter :starting-count="39">
      <template #default="{ count }">
        The count is {{ count }}
      </template>
    </Counter>
  </div>
</template>
```

We now have this tree instead:

```
Vue component tree:
- ServerComponent
  - Counter
  - #default

Tree rendered to the DOM:
- ServerComponent
  - Counter
    - #default
```

Because the slot is being rendered in a server component context, it will not become interactive on the client-side.

What about multiple levels of nesting?

Take this example:

```
// Parent.client.vue
<template>
  <ServerComponent>
    <ServerComponent>
      <Counter :starting-count="39" />
    </ServerComponent>
  </ServerComponent>
</template>
```

Although you'd expect this to work given that we're still in a client-side context, this actually throws an error as of the time of writing. I'm not sure if this is a bug or intended behaviour, but this feature *is* currently experimental.

In fact, it appears that slots within server components only work one level deep, regardless of what's in the slot.

However, there are no restrictions on putting server components into other slots as content themselves.

96.

Server Component Fallback

If there's an error with rendering your server component, there's a `#fallback` slot that will be rendered on the client-side:

```
<NuxtIsland
  name="ServerCounter"
  :props="{
    startingCount,
  }"
>
  <template #fallback>
    Counting failed!
  </template>
</NuxtIsland>
```

97.

Client only and server only pages

We can easily set a page in our `/pages` folder to be either client-side or server-side rendered *only*, just by changing the suffix.

The `client-page.client.vue` will only render on the client-side. The exception is for any server components that you have in there. Those will be rendered on the server:

```
<template>
  <div>Client page: {{ count }}</div>

  <!-- Rendered like a normal server component would -->
  <JustServer />
</template>

<script setup lang="ts">
const count = ref(0);

onMounted(() => {
  setInterval(() => {
    count.value++;
  }, 1000);
});
</script>
```

But `server-page.server.vue` will be rendered only on the server, disabling any interactivity:

```vue
<template>
  <div>Server page: {{ count }}</div>
</template>

<script setup lang="ts">
const count = ref(6);

onMounted(() => {
  setInterval(() => {
    count.value++;
  }, 1000);
});
</script>
```

If you navigate to `/server-page` you'll get "Server page: 6", and that's it. No counting, since the Javascript is never shipped.

Remember, server components are required to have a single root node, and a server page is really just a server component:

```vue
<template>
  <div>
    <div>Server page.</div>
    <div>With multiple root nodes.</div>
  </div>
</template>
```

But unlike a regular server component, you cannot nest interactive components within it, unless you have set `experimental.componentIsland.selectiveClient` in your config to `deep`:

```vue
<template>
  <div>
    <div>Server page: {{ count }}</div>
    <!-- Renders "17" but doesn't become interactive by default -->
    <Counter nuxt-client :starting-count="17" />
  </div>
</template>
```

12 Error Essentials

All about creating, handling, and understanding errors.

98.

Debug hydration errors in production

Hydration errors are one of the trickiest parts about SSR — especially when they only happen in production.

Thankfully, Vue 3.4 lets us debug hydration errors in production.

In Nuxt, all we need to do is update our config:

```
export default defineNuxtConfig({
  debug: true,
  // rest of your config...
})
```

If you aren't using Nuxt, you can enable this using the new compile-time flag: `__VUE_PROD_HYDRATION_MISMATCH_DETAILS__`. This is what Nuxt uses.

Enabling flags is different based on what build tool you're using, but if you're using Vite this is what it looks like in your `vite.config.js` file:

```
import { defineConfig } from 'vite'

export default defineConfig({
  define: {
    __VUE_PROD_HYDRATION_MISMATCH_DETAILS__: 'true'
  }
})
```

Turning this on will increase your bundle size, but it's really useful for tracking down those pesky hydration errors.

99.

Throwing errors the right way

To create an error, we'll throw a Nuxt error that's returned by the `createError` method:

```
throw createError({
  statusCode: 500,
  statusMessage: 'Something bad happened on the server',
});
```

The `createError` method is *isomorphic*, meaning it can be called on the server or on the client. This means you can use it inside of any Vue components, composables, or route middleware.

You'll also use it inside your Nitro event handlers in your server routes (your API):

```
import { PrismaClient } from '@prisma/client';
const prisma = new PrismaClient();

export default defineEventHandler(async (event) => {
  const { chapterSlug, lessonSlug } = event.context.params;

  const lesson = await prisma.lesson.findFirst({
    where: {
      slug: lessonSlug,
      chapter: {
        slug: chapterSlug,
      },
    },
  });

  // 👇 Throw an error if we can't find the lesson
  if (!lesson) {
    throw createError({
      statusCode: 404,
      statusMessage: 'Lesson not found',
    });
  }

  return lesson;
});
```

(I wrote a whole series on using Prisma with Nuxt and Supabase if you want to learn more about that.)

If we wanted to throw an error in a server middleware:

```
// ~/server/middleware/auth.js

import { serverSupabaseUser } from '#supabase/server';

export default defineEventHandler(async (event) => {
  const user = await serverSupabaseUser(event);

  // 👇 Throw an error to prevent the request from continuing
  if (!user) {
    throw createError({
      statusCode: 401,
      message: 'Unauthorized',
    });
  }
});
```

Here, we're throwing an error using `createError` when the `user` is not defined. This prevents the request from continuing, and immediately returns a `401` error back to whoever is calling this endpoint.

However, this is run for *all* requests to *all* endpoints.

Server middleware in Nuxt cannot be scoped to specific endpoints or routes. But there is a simple way to solve this so we can protect only certain routes.

100.

Custom error pages

To create an error page, we need to add an `error.vue` file to the *root* of our application, alongside our `app.vue` and `nuxt.config.ts` files.

This is important, because this server page is not a "page" that is seen by the router. Not all Nuxt apps use file-based routing or use Vue Router, so we can't rely on Vue Router to display the error page for us.

A super basic error page might look something like this:

```
<template>
  <NuxtLayout>
    <div>
      <h1>Dang</h1>
      <p>It looks like something broke.</p>
      <p>Sorry about that.</p>
    </div>
  </NuxtLayout>
</template>
```

Not super helpful, but we can make it better by using the `useError` composable to grab more details about the global error:

```
const error = useError();
```

Now, we can add a more descriptive message to our error page:

```
<template>
  <NuxtLayout>
    <div class="prose">
      <h1>Dang</h1>
      <p>
        <strong>{{ error.message }}</strong>
      </p>
      <p>It looks like something broke.</p>
      <p>Sorry about that.</p>
    </div>
  </NuxtLayout>
</template>
```

We can update our error page to include a check for this `statusCode`:

```
<template>
  <NuxtLayout>
    <div class="prose">
      <template v-if="error.statusCode === 404">
        <h1>404!</h1>
        <p>Sorry, that page doesn't exist.</p>
      </template>
      <template v-else>
        <h1>Dang</h1>
        <p>
          <strong>{{ error.message }}</strong>
        </p>
        <p>It looks like something broke.</p>
        <p>Sorry about that.</p>
      </template>
      <p>
        Go back to your
        <a @click="handleError">
          dashboard.
        </a>
      </p>
    </div>
  </NuxtLayout>
</template>
```

Using this strategy we're able to render what we need based on the type of error and the message that the error contains.

101.

Handling client-side errors

We can use the built-in `NuxtErrorBoundary` component to contain errors in our Nuxt apps:

```
<NuxtErrorBoundary>
  <!-- Put components in here -->
</NuxtErrorBoundary>
```

This lets us intelligently handle and contain errors. Without this boundary, any errors will bubble up until they're caught, potentially crashing your entire app.

We can provide a relevant error UI based on what's inside our default slot:

```
<NuxtErrorBoundary>
  <UserLogin />
  <template #error="{ error }">
    <div>
      <p>Unable to login! Try again later.</p>
      <p>{{ error.message }}</p>
    </div>
  </template>
</NuxtErrorBoundary>
```

We can reset this error and re-render the default slot by setting the `error` back to `null`:

```
const recoverFromError = (error) => {
  // Try to resolve the error here
  error.value = null;
}
```

In our error UI we'd use this:

```
<NuxtErrorBoundary>
  <UserLogin />
  <template #error="{ error }">
    <div>
      <p>Unable to login! Try again later.</p>
      <p>{{ error.message }}</p>
    </div>
  </template>
</NuxtErrorBoundary>
```

We can reset the error by setting the `error` ref back to `null`:

```
const recoverFromError = (error) => {
  // Try to resolve the error here
  error.value = null;
}
```

In our template, we'd use the `recoverFromError` function like this:

```
<NuxtErrorBoundary>
  <UserLogin />
  <template #error="{ error }">
    <div>
      <p>Unable to login! Try again later.</p>
      <p>{{ error.message }}</p>
      <button @click="recoverFromError(error)">
        Reset
      </button>
    </div>
  </template>
</NuxtErrorBoundary>
```

102.

When to use Error Boundaries

When do we use the `NuxtErrorBoundary` component?

Well, the main benefit of an error boundary is that we can contain errors within our application, and then handle them in specific ways instead of just throwing up a generic error page.

That means we should place `NuxtErrorBoundary` components around distinct chunks of functionality — from the user's perspective — where you can handle a group of potential errors together.

For example:

- `NuxtPage` components that represent nested routes
- Widgets on a dashboard
- Modals

103.

Handling errors

When an error occurs, we need to *recover* from that error somehow.

The most basic thing we can do is reset the error and then navigate somewhere else using the `clearError` method:

```
// Get more info about the error
const error = useError();

const handleError = () => {
  // Reset the error and navigate back somewhere "safe"
  clearError({
    redirect: '/dashboard',
  });
};
```

Then we just need to hook up this handler to our template:

```
<template>
  <NuxtLayout>
    <div class="prose">
      <h1>Dang</h1>
      <p>
        <strong>{{ error.message }}</strong>
      </p>
      <p>It looks like something broke.</p>
      <p>Sorry about that.</p>
      <p>
        Go back to your
        <a @click="handleError">
          dashboard.
        </a>
      </p>
    </div>
  </NuxtLayout>
</template>
```

When the user clicks on the link, the server error will be cleared in the Nuxt app and they'll be redirected to the `/dashboard` url.

104.

Global Errors vs. Client-side Errors

In Nuxt we have two types of errors:

- Global errors: these errors can be thought of as "server-side" errors, but they're still accessible from the client
- Client-side errors: these errors *only* exist on the client, so they don't affect the rest of your app and won't show up in logs unless you use a logging service like LogRocket (there are many of these services out there, I don't endorse any specific one).

It's important to understand this distinction because these errors happen under different circumstances and need to be handled differently.

Global errors can happen any time the server is executing code. Mainly, this is during an API call, during a server-side render, or any of the code that glues these two together.

Client-side errors mainly happen while interacting within an app, but they can *also* happen on route changes because of how Nuxt's Universal Rendering works.

It's important to note that the `NuxtErrorBoundary` component only deals with *client-side errors*, and does nothing to handle or clear these global errors.

To do that, we need to use the error handling composables and utilities from Nuxt:

- `useError`
- `clearError`
- `createError`
- `showError`

105.

Errors in route middleware

Dealing with errors properly in route middleware is (thankfully!) not that different from what we've already seen.

But, there *is* a slight twist.

Let's take this example route middleware:

```
export default defineNuxtRouteMiddleware((to, from) => {
  // We can use the `to` and `from` routes to figure
  // out what we should be doing in this middleware

  if (notValidRoute(to)) {
    // Shows a "Page not found" error by default
    return abortNavigation();
  } else if (useDifferentError(to)) {
    // Pass in a custom error
    // 👇 We need to wrap createError with abortNavigation
    return abortNavigation(
      createError({
        statusCode: 404,
        message: 'The route could not be found :(',
      })
    );
  } else if (shouldRedirect(to)) {
    // Redirect back to the home page
    return navigateTo('/');
  } else {
    // If everything looks good, we won't do anything
    return;
  }
});
```

In route middleware we can't simply `throw` an error, we have to pass it to the special `abortNavigation` method, and then return the Promise from that method instead:

```
return abortNavigation(
  createError({
    statusCode: 404,
    message: 'The route could not be found :(',
  })
);
```

The `abortNavigation` method can be called in a few different ways:

1. No parameters `abortNavigation()` — this will create a `404` and `Page not found` error page
2. Just an error message `abortNavigation('Whoooooops!')` — allows us to set a more descriptive error message
3. Custom error object `abortNavigation(error)` — this lets us customize exactly what error is used. We can either get this error object from somewhere else (like a `try ... catch` block) or by using the `createError` method.

Returning `abortNavigation` will *stop* the navigation that is currently taking place, and instead redirect to an error page.

13 Testing Tactics

it('would be a failure if I didn't assert the importance of tests in this book somewhere')

106.

Easy Unit Testing

For your unit tests, `@nuxt/test-utils` lets you opt-in to a Nuxt environment by adding `.nuxt.` to the filename of your test:

```
./tests/MyComponent.nuxt.test.ts
```

You can also add a special comment at the top of the file:

```
@vitest-environment nuxt
```

Or enable the environment for all Vitest tests in your config:

```ts
// vitest.config.ts
import { defineVitestConfig } from '@nuxt/test-utils/config';

export default defineVitestConfig({
  test: {
    environment: 'nuxt'
  },
};
```

107.

Mount Components When Testing

When writing unit tests, you have access to a bunch of helper methods.

One super useful one is `mountSuspended`. It lets you mount any component inside your Nuxt context with async setup:

```
import { describe, it, expect } from 'vitest';
import { mountSuspended } from '@nuxt/test-utils/runtime';

import MyComponent from './MyComponent.vue';

describe('MyComponent', () => {
  it('renders the message correctly', async () => {
    const wrapper = await mountSuspended(MyComponent);
    expect(wrapper.text()).toContain('This component is set up.');
  });
});
```

You're also able to mount your app at a specific route, by passing in the App component and a route:

```
import { describe, it, expect } from 'vitest';
import { mountSuspended } from '@nuxt/test-utils/runtime';

import App from './App.vue';

describe('About', () => {
  it('renders the about page', async () => {
    const wrapper = await mountSuspended(App, { route: '/about' });
    expect(wrapper.text()).toContain('Hi, my name is Michael!');
  });
});
```

108.

Mock Any Import for Testing

One handy helper method in `@nuxt/test-utils` is `mockNuxtImport`.

It's a convenience method to make it easier to mock anything that Nuxt would normally auto-import:

```
import { mockNuxtImport } from '@nuxt/test-utils/runtime';

mockNuxtImport('useAsyncData', () => {
  return () => {
    return { data: 'Mocked data' };
  };
});

// ... tests
```

This is actually a macro that gets transformed into `vi.mock`, so it can only be used once per file.

109.

Mock Components When Testing

When testing, you'll often need to shallow render a component — mocking out any descendent components to keep your test simpler.

With `@nuxt/test-utils` you can use the `mockComponent` utility method to help with that:

```
import { mockComponent } from '@nuxt/test-utils/runtime';

// Use Options API to configure
mockComponent('MyComponent', {
  props: {
    value: String
  },
  setup(props) {
    // ...
  },
});

// Or use a separate file to clean things up (and use <script setup>)
mockComponent('MyComponent', () => import('./MyComponent.mock.vue'));

// ...tests
```

110.

Easily Mock API Routes in Nuxt

If you've ever written unit tests, you'll have needed to mock out API endpoints that are used in your components or stores.

With `@nuxt/test-utils` this is really simple, because you get the `registerEndpoint` utility method:

```
import { registerEndpoint } from '@nuxt/test-utils/runtime';
import userTestData from './userTestData.json';

registerEndpoint('/users/', () ⇒ userTestData);

// ...tests
```

You can mock any server route (API endpoint), including external endpoints if you need.

14 Other Observations

a.k.a the tidbits I couldn't easily fit into other chapters.

111.

Custom Prose Components

It's really easy to customize how our Markdown is rendered in Nuxt Content by writing custom prose components.

We'll copy in the default component into `./components/content` so we have a good starting point with all the props. Then, we can customize to our hearts content (see what I did there?).

If we wanted to add a `filename` to the code block component, we'd copy over `ProsePre` and update it a little:

```vue
<template>
  <div
    v-if="filename"
    class="bg-slate-200 font-mono text-sm border
           border-slate-300 py-2 px-3 rounded-t-md text-black"
  >
    {{ filename }}
  </div>
  <pre
    :class="{
      [$props.class as string]: true,
      'mt-0 rounded-t-none': $props.filename,
    }"
  ><slot /></pre>
</template>

<script setup lang="ts">
defineProps({
  code: {
    type: String,
    default: '',
  },
  language: {
    type: String,
    default: null,
  },
  filename: {
    type: String,
    default: null,
  },
  highlights: {
    type: Array as () => number[],
    default: () => [],
  },
  meta: {
    type: String,
    default: null,
  },
  class: {
    type: String,
    default: null,
  },
});
</script>

<style>
```

```
pre code .line {
  display: block;
}
</style>
```

We use some conditional classes to adjust the border rounding. When the `filename` is set, we'll need to make the top of the code block square so we don't have any weird gaps.

You may notice the formatting of the `pre` and `slot` tags are funny — this is actually necessary. Because the `pre` tag preserves all whitespace, if we formatted it with newlines those newlines would end up rendered to our page as well.

112.

Flatten Nuxt Content Routes

I wanted to organize my blog content into several folders:

- Articles: `content/articles/`
- Newsletters: `content/newsletters/`

By default though, Nuxt Content would set up these routes to include those prefixes. But I want all of my routes to be at the root level:

- Articles: michaelnthiessen.com/my-latest-article
- Newsletters: michaelnthiessen.com/most-recent-newsletter

We can do this manually for each Markdown file by overriding the `_path` property through it's frontmatter:

```
---
title: My Latest Article
date: today
_path: "/my-latest-article"
---
```

This is extremely tedious, error-prone, and generally annoying.

Luckily, we can write a simple Nitro plugin that will do this transform automatically.

Create a `content.ts` file in `server/plugins/`:

```
export default defineNitroPlugin((nitroApp) => {
  nitroApp.hooks.hook('content:file:afterParse', (file) => {
    for (const prefix of ['/articles', '/newsletters']) {
      if (file._path.startsWith(prefix)) {
        // Keep the prefix so we can query based on it still
        file._original_dir = prefix;
        // Remove prefix from path
        file._path = file._path.replace(prefix, '');
      }
    }
  });
});
```

Nitro is the server that Nuxt uses internally. We can hook into it's processing pipeline and do a bit of tweaking.

However, doing this breaks `queryContent` calls if we're filtering based on the path, since `queryContent` is looking at the `_path` property we've just modified. This is why we want to keep that original directory around.

We can modify our `queryContent` calls to filter on this new `_original_dir` property:

```
// Before
queryContent('/articles')

// After
queryContent()
  .where({
    _original_dir: { $eq: '/articles' },
  });
```

Pro tip: use `nuxi clean` to force Nuxt Content to re-fetch and re-transform all of your content.

113.

Nuxt Content Queries

Nuxt Content 2 gives us an effortless way to query our content using the `queryContent` method:

```
// composables/useArticles.js

export default () ⇒
  queryContent('articles')
    .where({
      // Optional fields that may be true or non-existent
      ghost: { $ne: true },
      newsletter: { $ne: true },
      // Don't render articles scheduled for the future
      date: { $lte: new Date() },
    })
    .only(['title', 'path', 'description', 'date', 'tags'])
    .sort({ date: -1 })
    .find();
```

Here, I've created a composable called `useArticles` for my blog, which grabs all of the content inside of the `content/articles/` directory.

The `queryContent` composable is a query *builder*, which gives us a lot of expressiveness in *what* data we fetch. Let's see how we're using this here.

First, we're using a `where` clause to filter out all the articles we don't want. Sometimes I will add an article before I want it to be "published" to the site.

I do this by setting the `date` in the future and then only taking articles before "today" using this clause:

```
date: { $lte: new Date() }
```

Second, some articles are the newsletters I write each week. Others are pieces of content that I want to keep in the `articles` folder but don't want to be published.

I use frontmatter fields to specify this:

```
---
newsletter: true  # This is a newsletter
---
```

```
---
ghost: true  # This content won't appear on the site
---
```

Third, we use the `only` clause to grab just the fields we need. By default, the `queryContent` method returns a lot of data, including the entire piece of content itself, so this can make a big difference in payload size.

Lastly, as you have probably guessed, we have a `sort` clause to sort the articles so the most recent ones appear last.

The `queryContent` composable has more options than this, which you can read about on the docs.

114.

Different Kinds of Utilities

There are two kinds of utilities in Nuxt: Vue utilities and server utilities (or Nitro utilities).

Vue utilities live in the `~/utils` folder and are auto-imported into the Vue context of your application:

```
// ~/utils/toArray.js

export default function(obj) {
  return Array.isArray(obj)
    ? obj
    : Object.values(obj);
}
```

You can use them in any Vue components, route middleware (since it runs in Vue Router), or composables:

```
const array = computed(() => toArray(someValue));
```

You can also define your utils using named exports, putting multiple in a single file:

```
// ~/utils/utils.js

export const toArray = (obj) => {
  return Array.isArray(obj)
    ? obj
    : Object.values(obj);
};

export const reverseArray = (arr) => {
  // Make sure to create a new array
  return [...toArray(obj)].reverse();
};
```

Keep in mind, these should *not* use any sort of reactivity. If they do, they are technically composables, and it would be better to put them in the `~/composables` folder instead.

However, these two folders, `~/utils` and `~/composables`, aren't special in any way. They've just been configured to be auto-imported. You can configure *any* directory you want to be auto-imported in the same way.

Server utilities work in essentially the same way, except they're only available in the Nitro context: server routes and server middleware. You define them in the `~/server/utils` directory:

```
// ~/server/utils/toArray.js

export default function(obj) {
  return Array.isArray(obj)
    ? obj
    : Object.values(obj);
}
```

115.

Using OAuth

Let's say you want to create a Twitte/X bot that will automatically tweet out your favourite quotes.

Instead of giving the bot all of your login information (which is not a good idea), you can use OAuth to securely provide access to your Twitter/X account.

OAuth involves four pieces:

1. **The user** — this is the person whose information is being accessed.
2. **The resource server** — the server that holds the user's information. This could be a website or service such as Twitter, Google Drive, or an email server.
3. **The client or third-party app** — the app or service that wants to access the user's information on the resource server on their behalf.
4. **The authorization server** — the server that controls access to the resource server. This server is responsible for authenticating the user and issuing a user token to the client app, which can then be used to access the user's information on the resource server.

In the case of our Twitter bot, the pieces would be:

1. **The user** — you, the person that wants to give the Twitter bot access to your account.
2. **The resource serve**r — this is Twitter's API, which holds all of the user's information such as their tweets, followers, and other data.
3. **The client or third-party app** — this is the Twitter bot itself.
4. **The authorization server** — this is Twitter's authentication service.

Now that we have all of our pieces in place, we need to see how they interact with each other.

Here's what the OAuth flow would look like for our Twitter bot example:

1. The user clicks on a "Log in with Twitter" button in the Twitter bot app or service.
2. The app or service redirects the user to Twitter's authorization server.
3. The user logs in to their Twitter account (or confirms that they are already logged in).

4. Twitter's authorization server authenticates the user and redirects them back to the app or service with a user token.
5. The app or service receives the user token and uses it to access the user's information on Twitter's API.
6. The app or service can now perform actions on behalf of the user, such as posting tweets or retrieving the user's timeline.

Sometimes, however, we don't actually need to access anything on the resource server. Instead, we're only using the OAuth flow for *authentication* — to verify the user's identity.

In this case, once we get that token back, we're set. That access token contains all of the information we need about the user, and proves their identity.

This is how I use Github in my course platform. I only need to verify each user's identity. I don't actually need to access any of their Github data at all.

116.

Authentication vs. Authorization

I need to clarify the distinction between *authentication* and *authorization*.

It doesn't help that they both shorten to "auth", so it's not always clear which one we're referring to!

Here are some quick definitions:

- **Authentication** — this is about *identity*, about proving that you are who you say you are.
- **Authorization** — this is about *access*, about determining *who* can do *what* in your application.

Most applications need both. You only want paying customers to access your app, so you need *authorization.* But you need *authentication* to determine who is your customer and who isn't.

Authentication can be hard, which is why most apps use OAuth so that we can rely on third parties like Google, Twitter/X or Github to provide the identity for us. We could also implement this ourselves, but that involves managing passwords which is a whole can of worms.

Authorization can be a simple check in the database. Does this email address / Github user have access to this feature?

Just make sure to do this on the server — the client-side is never all that secure because the end user has complete access to everything once it's on their device.

117.

Hooking into Hydration

If you want your plugins to leverage the hydration lifecycle, you can use the `useHydration` composable:

```
useHydration(
  'unique-key',

  // Get function to set payload on server
  () => {
    return "some value we got from somewhere";
  },

  // Set function to get payload on client
  (payload) => {
    console.log('Here is the payload', payload);
  }
);
```

It's actually a pretty straightforward composable, giving you insight into how hooks and the Nuxt payload work (I've removed types for clarity):

```
export const useHydration = (key, get, set) => {
  const nuxtApp = useNuxtApp()

  if (import.meta.server) {
    nuxtApp.hooks.hook('app:rendered', () => {
      nuxtApp.payload[key] = get()
    })
  }

  if (import.meta.client) {
    nuxtApp.hooks.hook('app:created', () => {
      set(nuxtApp.payload[key])
    })
  }
}
```

If we're on the server, use the "app:rendered" hook to wait until SSR is done. Then, we

call the `get` function that's passed in to set the payload.

On the client we do the reverse. We wait for the "app:created" hook, then take our payload and pass it to the `set` function.

From the user's perspective these functions might make more sense switched around, but seeing the actual code here makes it easier to see what's going on.

However, the one caveat is that this only works in the context of a Nuxt plugin, not a component. This is because the Vue app is actually mounted *after* the `app:created` hook has run.

To do this in a component, we can instead wait for the `app:mounted` hook which is only run on the client. And since the `app:rendered` hook is only called on the server, we can get rid of all the extra conditionals too:

```
export const useComponentHydration = (key, get, set) => {
  const nuxtApp = useNuxtApp()

  nuxtApp.hooks.hook('app:rendered', () => {
    nuxtApp.payload[key] = get()
  });

  nuxtApp.hooks.hook('app:mounted', () => {
    set(nuxtApp.payload[key])
  })
}
```

118.

Advanced Hydration with onPreHydrate

If you need some code to run on the client *before* Nuxt does any initialization, you can use the `onPreHydrate` composable:

```
onPreHydrate(() => {
  // Manipulate the DOM in a specific way
  const root = document.querySelector('#__nuxt');
  root.addEventListener('click', () => alert('Hello World!'));
});
```

It works by stringifying the entire function and inlining it into the initial HTML sent to the client. This means that you cannot use any external dependencies (no closures) and cannot rely on Vue or Nuxt to exist, since it runs before they are initialized:

```
onPreHydrate(() => {
  // This will not work, since Vue hasn't been loaded yet!
  // Vanilla JS only
  const count = ref(0);
  const doubleCount = computed(() => count.value * 2);
});
```

15 Code Demos

Dive deeper into Nuxt through these interactive code repos.

1. Server Components

This repo will take you through many different ways of using server components, including the `NuxtIsland` component.

Github →

Stackblitz →

2. Layers

Layers are one of the best ways to organize large Nuxt projects.

See how to use them, as well as some interesting edge cases in this demo.

Github →

Stackblitz →

3. Keyed Composables

See how you can use Nuxt's key auto-injection in your own composables.

Github →

Stackblitz →

4. Routing Precedence

Nuxt's file-based routing has some interesting edge cases around naming collisions and child routes.

See how these can be resolved and worked around in this code demo.

Github →

Stackblitz →

5. Keeping Pages Alive

Because Nuxt has client-side routing, we can use the `keepalive` component on pages. This makes it possible to do some interesting things!

Github →

Stackblitz →

6. Hooks and hydration

See a use case for Nuxt hooks in action, by implementing our own custom version of `useState` as well as a random number generator that's SSR compatible.

Here, we're using runtime hooks, but the same can be applied to build-time hooks for modules and Nitro hooks as well.

Github →

Stackblitz →

7. Prefetching Components

This demo shows how we can prefetch lazy components if we think they'll be needed.

Github →

Stackblitz →