Chapter 1: Introduction to Nuxt 4

Overview of Nuxt

Nuxt.js is a **progressive framework** based on Vue.js that makes building Vue applications easier by providing a ready-to-use architecture with powerful features out of the box. It is designed to help developers build:

- **Universal apps** (rendered both on server and client)
- **Single Page Applications (SPA)**
- **Static generated websites**

Nuxt helps with routing, data fetching, layout management, and many other tasks automatically, so you don't need to write extra code or configure many things manually.

Features and Benefits

- **File-based routing:** Create Vue files inside the pages/ directory and Nuxt automatically creates routes for you.
- **Server-Side Rendering (SSR):** Improves SEO and faster initial page load by rendering HTML on the server.
- **Static Site Generation (SSG):** Generate fully static websites that can be hosted anywhere.
- **Auto-imports:** Automatically imports Vue composables, components, and composables without manual imports.
- **API Routes:** Write backend logic inside your Nuxt project using server API routes.
- **Easy State Management:** Built-in support for Pinia (recommended state manager).
- **Strong TypeScript Support:** Full TypeScript support for better development experience.
- **Plugin System:** Easily extend your app with plugins.
- **Powerful Modules:** Add functionalities using Nuxt modules.
- **Vite Integration:** Uses Vite for faster build and development experience.
- **Great Developer Experience (DX):** Hot Module Replacement, error overlays, and debugging tools.

What’s New in Nuxt 4?

Nuxt 4 is a major upgrade with many improvements:

- **Based on Vue 3:** Using Vue 3's Composition API, Teleport, Fragments, and other new features.
- **Vite by default:** Faster development server and optimized builds.
- **Improved server routes:** Write backend API directly in the server/ directory.
- **Auto-import improvements:** Better automatic importing of Vue APIs, composables, and stores.
- **Improved TypeScript integration:** More accurate type checking and better support.
- **Better performance:** Smaller bundles and faster page loads.
- **Improved module and plugin system:** More flexible and easier to customize.

Vue 3 and Vite Integration

Vue 3 in Nuxt 4

Vue 3 introduced many modern features and performance improvements that Nuxt 4 fully leverages:

- **Composition API:** Organizes component logic using functions instead of options.
- **Fragments:** Return multiple root nodes from a component.
- **Teleport:** Render content outside the current DOM hierarchy.
- **Improved reactivity system:** Faster and more efficient.

Example of Vue 3 Composition API:

vue

CopyEdit

```
<script setup>
import { ref } from 'vue'


const count = ref(0)
function increment() {
  count.value++
}
</script>


<template>
  <button @click="increment">Clicked {{ count }} times</button>
</template>
```

Vite in Nuxt 4

Vite replaces Webpack and provides:

- **Fast cold start:** Starts dev server instantly.
- **Instant Hot Module Replacement (HMR):** Changes reflect immediately without full page reload.
- **Modern build tools:** Uses ES modules for faster builds.

Using Vite makes your development much faster and smoother compared to older bundlers.

Summary of Chapter 1

- Nuxt 4 is a powerful Vue 3 framework designed for server-side rendered, static, and SPA applications.
- It provides automatic routing, SSR, SSG, plugins, modules, and many developer-friendly features.
- New in Nuxt 4: Vue 3 support, Vite integration, improved server API, better TypeScript, and performance enhancements.

Chapter 2: Getting Started

---

1. Installation and Setup

Before you start building your Nuxt 4 app, you need to set up your development environment.

Prerequisites:

- Node.js (v16 or higher recommended)
- npm or yarn package manager

2. Creating a Project with nuxi

Nuxt 4 uses a CLI tool called **nuxi** to scaffold new projects and run commands.

Step to create a new Nuxt 4 project:

Open your terminal and run:

bash

CopyEdit

```
npx nuxi init my-nuxt-app
```

This will create a folder called my-nuxt-app with all the starter files.

Then move into the folder:

bash

CopyEdit

```
cd my-nuxt-app
```

Install dependencies:

bash

CopyEdit

```
npm install
```

Finally, start the development server:

bash

CopyEdit

```
npm run dev
```

Your app will be running on http://localhost:3000

3. Project Structure Overview

Understanding the default Nuxt 4 folder structure is important:

perl

CopyEdit

```
my-nuxt-app/
```

```
├─ app/             # Root app entry and setup
├─ assets/          # Static files like images, fonts, stylesheets
├─ components/      # Vue components (auto-imported)
├─ composables/     # Reusable functions (auto-imported)
├─ layouts/         # Layouts to wrap pages (header/footer)
├─ middleware/      # Functions that run before rendering routes
├─ node_modules/    # Installed npm packages
├─ pages/           # Vue files representing routes
├─ plugins/         # Plugins to add third party or global functionality
├─ public/          # Static files served as-is
├─ server/          # Backend API routes and server code
├─ stores/          # Pinia stores (state management)
├─ nuxt.config.ts   # Nuxt configuration file
├─ package.json     # Project dependencies and scripts
```

---

4. First Nuxt 4 App Example

By default, the project has a pages/index.vue file which renders the home page.

Open pages/index.vue and you will see:

vue

CopyEdit

```
<template>
  <h1>Welcome to Nuxt 4!</h1>
</template>
```

You can edit this file to customize the homepage.

Add some styling and content:

vue

CopyEdit

```
<template>
  <div class="container mx-auto mt-20 text-center">
    <h1 class="text-5xl font-bold mb-4">Welcome to Nuxt 4</h1>
    <p class="text-gray-600">Your Nuxt 4 app is running smoothly!</p>
  </div>
</template>
```

Note: Tailwind CSS utility classes are used here (Nuxt 4 supports Tailwind out of the box if added).

---

5. Running Your Development Server

The most common commands you'll use:

Command	Description
npm run dev	Start development server with hot reload
npm run build	Build your app for production
npm run preview	Preview the production build locally
npm run generate	Generate a fully static version of your app

---

6. Installing Tailwind CSS (Optional)

Tailwind CSS is a popular utility-first CSS framework.

To add Tailwind CSS:

bash

CopyEdit

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Configure tailwind.config.js:

js

CopyEdit

```
export default {
  content: [
    './components/**/*.{vue,js}',
    './layouts/**/*.vue',
    './pages/**/*.vue',
    './app.vue',
    './plugins/**/*.{js,ts}',
    './nuxt.config.{js,ts}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Add Tailwind directives to assets/css/tailwind.css:

css

CopyEdit

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Include this CSS in your nuxt.config.ts:

ts

CopyEdit

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

Now you can use Tailwind CSS classes in your templates.

Summary

- You learned how to create and run a Nuxt 4 project using nuxi.
- Project structure explained for better understanding.
- Wrote your first page with simple HTML and CSS classes.
- Learned basic npm scripts for development and build.
- (Optional) Added Tailwind CSS to style your app with utility classes.

Chapter 3: Pages and Routing

---

1. File-based Routing Explained

Nuxt 4 uses **file-based routing**, which means your routes (URLs) are automatically generated based on the .vue files inside the pages/ directory.

For example:

- pages/index.vue → / (home page)
- pages/about.vue → /about
- pages/contact.vue → /contact

You don’t need to write any route definitions manually.

---

2. Static vs Dynamic Routes

- **Static routes** are fixed routes like /about, /contact.
- **Dynamic routes** can change based on parameters, like blog posts or user profiles.

To create a **dynamic route**, use square brackets [param] in the filename.

Example:

plaintext

CopyEdit

```
pages/
  blog/
    [slug].vue
```

This creates a dynamic route /blog/:slug

---

3. Nested Routes

You can create **nested routes** by creating folders and Vue files inside pages/.

Example:

plaintext

CopyEdit

```
pages/
  dashboard/
    index.vue       -> /dashboard
    settings.vue    -> /dashboard/settings
```

Here, settings.vue is a nested route inside /dashboard.

---

4. Route Parameters and Query Parameters

- **Route Parameters:** Values in dynamic routes.

In [slug].vue:

vue

CopyEdit

```
<script setup>
const route = useRoute()
const slug = route.params.slug
</script>

<template>
  <h1>Post: {{ slug }}</h1>
</template>
```

- **Query Parameters:** After ? in URL, accessed by route.query.

Example URL: /search?term=nuxt

Access in code:

js

CopyEdit

```
const term = route.query.term
```

---

5. Navigation and Links

To navigate between pages, Nuxt provides the <NuxtLink> component which works like Vue Router’s <router-link>.

Example:

vue

CopyEdit

```
<template>
  <nav>
    <NuxtLink to="/">Home</NuxtLink> |
    <NuxtLink to="/about">About</NuxtLink> |
    <NuxtLink :to="{ path: '/blog/my-first-post' }">First Blog</NuxtLink>
  </nav>
</template>
```

Advantages of <NuxtLink>:

- Client-side navigation without full page reload
- Active link classes (automatic CSS classes for current route)

---

6. Programmatic Navigation

Inside your Vue component, you can navigate using:

js

CopyEdit

```
const router = useRouter()

function goToAbout() {
  router.push('/about')
}
```

Example: Dynamic Blog Post Page

pages/blog/[slug].vue

vue

CopyEdit

```
<script setup>
const route = useRoute()
const slug = route.params.slug
</script>

<template>
  <div>
    <h1>Blog Post: {{ slug }}</h1>
    <p>This is the blog content for post "{{ slug }}".</p>
  </div>
</template>
```

Navigate to /blog/my-first-post and see the slug displayed.

Summary

- Nuxt 4 automatically creates routes based on files in pages/.
- Use static .vue files for fixed routes and [param].vue for dynamic routes.
- Nested folders create nested routes.
- Access route parameters with useRoute().
- Use <NuxtLink> for client-side navigation.
- Programmatic navigation via useRouter().

Chapter 4: Layouts

1. What are Layouts?

Layouts in Nuxt 4 are Vue components that **wrap your pages** to provide a consistent structure — such as headers, footers, navigation bars, or sidebars — across multiple pages.

Instead of adding header/footer code in every page, you create a layout once and apply it globally or selectively.

---

2. Default Layout

By default, Nuxt uses the layouts/default.vue file as the global layout.

Example:

vue

CopyEdit

```
<!-- layouts/default.vue -->
<template>
  <div>
    <header class="bg-blue-600 text-white p-4">
      <h1>My Nuxt 4 Site</h1>
    </header>

    <main class="p-6">
      <NuxtPage />
    </main>

    <footer class="bg-gray-800 text-white p-4 text-center">
      &copy; 2025 My Nuxt Site
    </footer>
  </div>
</template>
```

- <NuxtPage /> is a special component that renders the current page inside the layout.
- Every page will be wrapped inside this layout automatically.

---

3. Custom Layouts

You can create multiple layouts for different page types.

Example: Create layouts/admin.vue for admin pages:

vue

CopyEdit

```
<!-- layouts/admin.vue -->
<template>
  <div>
    <nav class="bg-gray-900 text-white p-4">Admin Navigation</nav>
    <section class="p-6">
      <NuxtPage />
    </section>
  </div>
</template>
```

---

4. Using Layouts with Pages

To use a specific layout in a page, add this to your page’s script section:

vue

CopyEdit

```
<script setup>
definePageMeta({
  layout: 'admin'
})
</script>
```

This tells Nuxt to use the admin layout for this page.

If you don’t specify, the default layout is used.

---

5. Nested Layouts

Nuxt 4 supports nested layouts. You can have layouts inside layouts for complex UI.

Example:

- layouts/default.vue — main layout
- layouts/dashboard.vue — dashboard layout nested inside default

In layouts/dashboard.vue, you can wrap <NuxtPage /> with extra UI elements, and pages can use this layout:

vue

CopyEdit

```
<script setup>
definePageMeta({
  layout: 'dashboard'
})
</script>
```

---

6. Layout without Layout

If for some reason you want a page without any layout, you can disable layout:

vue

CopyEdit

```
<script setup>
definePageMeta({
  layout: false
})
</script>
```

Summary

- Layouts wrap your pages to provide common UI structure.
- The default layout is layouts/default.vue.

- Create multiple layouts and assign them per page with definePageMeta.
- Use nested layouts for complex apps.
- You can disable layout on any page by setting layout: false.

Chapter 5: Components

1. What are Components?

Components are reusable Vue building blocks that encapsulate UI and logic. They help you break your UI into smaller, manageable parts, like buttons, cards, headers, or complex widgets.

2. Auto-importing Components

Nuxt 4 automatically imports components that you put inside the components/ folder. You don't need to import them manually!

Example: Suppose you have components/MyButton.vue:

vue

CopyEdit

```
<template>
  <button class="bg-blue-600 text-white px-4 py-2 rounded">
    <slot />
  </button>
</template>
```

You can use <MyButton> anywhere in your pages or other components without importing:

vue

CopyEdit

```
<template>
  <MyButton>Click Me</MyButton>
</template>
```

3. Global vs Local Components

- **Global components**: Those inside the components/ folder are globally available because of auto-import.
- **Local components**: You can manually import components anywhere to limit their scope.

Example of local import:

vue

CopyEdit

```
<script setup>
import MyModal from '~/components/MyModal.vue'
</script>

<template>
  <MyModal />
</template>
```

---

4. Async Components

You can lazy-load components to optimize performance using async components.

Example:

vue

CopyEdit

```
<script setup>
import { defineAsyncComponent } from 'vue'

const AsyncComponent = defineAsyncComponent(() => import('~/components/HeavyComponent.vue'))
</script>

<template>
  <AsyncComponent />
```

```
</template>
```

This delays loading HeavyComponent.vue until it is needed.

---

5. Functional Components

Functional components are stateless and faster because they don’t have component lifecycle or reactivity overhead.

In Vue 3, any component without reactive state and lifecycle hooks behaves functionally by default.

Example:

vue

CopyEdit

```
<script setup>
// No reactive state here
</script>

<template>
  <button class="text-blue-500 hover:text-blue-700">
    <slot />
  </button>
</template>
```

This simple button can be a functional component.

---

6. Best Practices for Components

- Keep components **small and focused** on a single task.
- Use **props** to pass data to components.
- Use **slots** to allow content injection.
- Name components with **PascalCase** (MyButton) for clarity.

Example: Creating a Reusable Card Component

components/InfoCard.vue

vue

CopyEdit

```
<script setup>
defineProps({
  title: String,
  description: String
})
</script>

<template>
  <div class="border p-4 rounded shadow">
    <h2 class="font-bold text-xl mb-2">{{ title }}</h2>
    <p>{{ description }}</p>
  </div>
</template>
```

Usage in a page:

vue

CopyEdit

```
<template>
  <InfoCard
    title="Nuxt 4 Components"
    description="Reusable building blocks of your UI."
  />
</template>
```

Summary

- Components are reusable UI parts.

- Nuxt auto-imports components inside the components/ folder.
- You can manually import components locally.
- Async components help lazy load heavy UI parts.
- Functional components are simple and stateless.
- Follow best practices to keep your app maintainable.

Chapter 6: Plugins

What Are Plugins in Nuxt 4?

Plugins in Nuxt allow you to **add global functionality** or **register third-party libraries** in your app. They provide a way to **extend the core framework** by injecting features or code that can be accessed throughout your app.

For example, you might want to:

- Register a UI library like Vuetify or Element Plus
- Add global methods or properties
- Register global filters or directives
- Initialize analytics or other services

Creating a Plugin

Plugins are Vue plugins that you place in the plugins/ folder.

Example: Create plugins/myPlugin.ts

ts

CopyEdit

```
export default defineNuxtPlugin((nuxtApp) => {
  // Inject a global function called $hello
  nuxtApp.provide('hello', () => console.log('Hello from my plugin!'))
})
```

Using Plugin Features in Components

You can access injected functionality inside your components using useNuxtApp():

vue

CopyEdit

```
<script setup lang="ts">
const { $hello } = useNuxtApp()
$hello()  // Outputs: Hello from my plugin!
</script>
```

Registering Third-Party Libraries

If you want to use an external library globally, import and register it inside a plugin.

Example: Registering dayjs for date formatting

1. Install dayjs:

bash

CopyEdit

```
npm install dayjs
```

2. Create plugin plugins/dayjs.ts

ts

CopyEdit

```
import dayjs from 'dayjs'

export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.provide('dayjs', dayjs)
})
```

3. Use in component:

vue

CopyEdit

```
<script setup lang="ts">
const { $dayjs } = useNuxtApp()
const now = $dayjs().format('YYYY-MM-DD')
```

```
</script>


<template>
  <p>Today is: {{ now }}</p>
</template>
```

Plugin Naming and Auto-Registration

Any file inside plugins/ ending with .ts or .js is auto-loaded.

You can control when the plugin runs by naming:

- plugins/client-only.ts → loads on client side only
- plugins/server-only.ts → loads on server side only

Add .client or .server before extension to control this.

Plugin Options in nuxt.config.ts

You can configure plugins in your nuxt.config.ts like this:

ts

CopyEdit

```
export default defineNuxtConfig({
  plugins: [
    '~/plugins/myPlugin.ts'
  ]
})
```

But by default, Nuxt auto-imports plugins in the plugins folder, so manual registration is often not necessary.

Summary:

- Plugins extend Nuxt functionality globally.
- Create plugins inside the plugins/ directory.

- Use defineNuxtPlugin() to define plugins.
- Inject features using nuxtApp.provide().
- Access injected features using useNuxtApp() in components.
- Register and configure third-party libraries inside plugins.
- Use file naming conventions for client/server-only plugins.

Chapter 7: Middleware

What is Middleware?

Middleware in Nuxt 4 is a function that runs **before a page or route loads**. It’s used to control access, run checks, or perform actions before rendering.

Common uses:

- Authentication checks
- Redirecting users
- Logging page visits
- Role-based access control

Types of Middleware

1. **Global Middleware**: Runs on every route.
2. **Named Middleware**: Runs only on routes/pages where you specify it.

1. Global Middleware

Create a file named middleware/auth.global.ts:

ts

CopyEdit

```
export default defineNuxtRouteMiddleware((to, from) => {
  // For example, block if user is not authenticated
  const isAuthenticated = false; // Replace with real logic
  if (!isAuthenticated && to.path !== '/login') {
    return navigateTo('/login')
  }
})
```

This runs on **every route** and redirects unauthenticated users to /login.

---

2. Named Middleware

Create middleware/auth.ts:

ts

CopyEdit

```
export default defineNuxtRouteMiddleware((to, from) => {
  const isLoggedIn = false; // Replace with real auth check
  if (!isLoggedIn && to.name !== 'login') {
    return navigateTo('/login')
  }
})
```

Use it on a page:

vue

CopyEdit

```
<script setup>
definePageMeta({
  middleware: 'auth'
})
</script>
```

Only pages that declare this middleware run this check.

---

3. Middleware with Parameters (Middleware Factory)

You can create a function that returns middleware to pass parameters.

Example:

ts

CopyEdit

```
export function authMiddleware(roleRequired: string) {
  return defineNuxtRouteMiddleware((to, from) => {
    const userRole = 'guest' // fetch user role
    if (userRole !== roleRequired) {
      return navigateTo('/unauthorized')
    }
  })
}
```

Use it in your page by importing and applying manually.

---

4. Accessing Route and App Context

Inside middleware, you get access to to, from routes and Nuxt app context via composables:

ts

CopyEdit

```
import { useAuth } from '~/composables/useAuth'

export default defineNuxtRouteMiddleware((to, from) => {
  const auth = useAuth()
  if (!auth.isLoggedIn && to.name !== 'login') {
    return navigateTo('/login')
  }
})
```

---

5. Returning Navigation Results

- To redirect, return navigateTo('/some-path').
- To cancel navigation, return false.

- To allow navigation, don’t return anything.

Summary:

- Middleware runs before pages to control access or run logic.
- Global middleware runs on all routes, named middleware runs selectively.
- Use defineNuxtRouteMiddleware() to create middleware.
- Use navigateTo() to redirect users.
- Middleware helps implement authentication, roles, and more.

Chapter 8: State Management with Pinia

What is Pinia?

Pinia is the official state management library for Vue 3 and Nuxt 4. It helps you manage shared state (data) across components in a **simple and scalable way**.

Why Use Pinia?

- Centralized state for your app
- Reactive and type-safe
- Modular stores (small pieces of state)
- DevTools integration
- Supports Composition API

---

1. Installing Pinia

Run:

bash

CopyEdit

```
npm install pinia
```

---

2. Setting Up Pinia in Nuxt 4

Create a plugin plugins/pinia.ts to register Pinia:

ts

CopyEdit

```
import { createPinia } from 'pinia'

export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.vueApp.use(createPinia())
})
```

Nuxt 4 automatically loads plugins in plugins/.

3. Defining a Store

Create a store inside stores/counter.ts:

ts

CopyEdit

```
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0
  }),
  actions: {
    increment() {
      this.count++
    }
  }
})
```

4. Using Store in Components

vue

CopyEdit

```
<script setup lang="ts">
import { useCounterStore } from '~/stores/counter'

const counter = useCounterStore()

function incrementCount() {
  counter.increment()
}
</script>

<template>
  <div>
    <p>Count: {{ counter.count }}</p>
    <button @click="incrementCount">Increment</button>
  </div>
</template>
```

---

5. Store Modules and Composition

You can split state into multiple stores for modularity:

- stores/user.ts for user info
- stores/cart.ts for shopping cart

Stores can also be composed and reused.

---

6. Using Pinia with Nuxt Devtools

Pinia integrates with Vue Devtools for time-travel debugging and inspection.

Summary:

- Pinia manages centralized reactive state in Nuxt 4 apps.
- Install and register Pinia via plugin.
- Define stores with state and actions.
- Use stores inside components by importing and calling.
- Modular stores help keep code clean and scalable.

Ready to proceed to **Chapter 9: Server API and Server Routes**?

You said:

yes

ChatGPT said:

Here’s **Chapter 9: Server API and Server Routes** in detail with examples.

Chapter 9: Server API and Server Routes

What are Server Routes in Nuxt 4?

Nuxt 4 supports building backend API routes **inside your project** in the server/ directory. This allows you to write backend logic (APIs, server middleware) without needing a separate backend app.

---

1. Server Directory Overview

- The server/ folder contains server code.
- server/api/ holds API endpoints.
- server/middleware/ holds server middleware.
- Supports request handlers with minimal code.

---

2. Creating API Endpoints

Create a file like server/api/hello.get.ts:

ts

CopyEdit

```
export default defineEventHandler(() => {
  return { message: 'Hello from Nuxt API!' }
})
```

- The .get.ts suffix indicates this handles GET requests.
- Other HTTP methods like .post.ts are supported.

---

3. Handling Requests and Responses

You can access request data and respond accordingly.

Example: server/api/user.post.ts

ts

CopyEdit

```
import { readBody } from 'h3'

export default async (event) => {
  const body = await readBody(event)  // get JSON post data
  return { received: body }
}
```

---

4. Middleware on Server Routes

You can write server middleware to intercept requests:

server/middleware/auth.ts

ts

CopyEdit

```
export default defineEventHandler((event) => {
  const authHeader = event.req.headers.authorization
  if (!authHeader) {
    throw createError({ statusCode: 401, statusMessage: 'Unauthorized' })
  }
})
```

Middleware applies to all server requests unless scoped otherwise.

---

5. Using Server Routes for Backend Logic

- Fetch from these API routes inside your Nuxt app with useFetch or fetch().
- Use server routes to connect to databases, handle authentication, or process forms.

Example: Simple API Route

server/api/time.get.ts

ts

CopyEdit

```
export default () => {
  return { time: new Date().toISOString() }
}
```

Fetch data in a component:

ts

CopyEdit

```
const { data } = await useFetch('/api/time')
```

Summary:

- server/ folder lets you build backend APIs inside Nuxt 4.
- API endpoints go in server/api/ with HTTP method suffixes.
- Access request data with helpers like readBody().
- Use middleware in server/middleware/ to control API access.
- Integrate frontend and backend seamlessly in one project.

Ready for **Chapter 10: Data Fetching and Async Data**?

You said:

yes

ChatGPT said:

Here’s **Chapter 10: Data Fetching and Async Data** in detail with examples.

Chapter 10: Data Fetching and Async Data

Why Data Fetching?

In Nuxt apps, you often need to **load data** from APIs or backend before rendering pages or components. Nuxt provides reactive composables to help with this, working well with server-side rendering (SSR) and client-side rendering.

---

1. useFetch

useFetch is a composable that lets you fetch data from an API reactively.

Example:

ts

CopyEdit

```
const { data, pending, error } = await useFetch('/api/hello')
```

- data holds the fetched data (reactive).
- pending is true while loading.
- error contains any error.

You can use this in <script setup>:

vue

CopyEdit

```
<script setup lang="ts">
const { data, pending, error } = await useFetch('/api/hello')
</script>

<template>
```

```
  <div v-if="pending">Loading...</div>
  <div v-else-if="error">Error: {{ error.message }}</div>
  <div v-else>{{ data.message }}</div>
</template>
```

2. useAsyncData

Similar to useFetch but more general and flexible.

Example:

ts

CopyEdit

```
const { data, pending, error } = await useAsyncData('uniqueKey', () =>
  $fetch('/api/data')
)
```

- useAsyncData caches data based on a key (uniqueKey).
- Useful for server-side fetched data.

3. Server-side Data Fetching

Both useFetch and useAsyncData work during SSR to prefetch data so the HTML sent to the client is already populated.

4. Client-side Data Fetching

You can call these composables inside lifecycle hooks (like onMounted) to fetch data only on client side.

5. Handling Loading and Errors

Use the reactive pending and error properties to:

- Show loading spinners or skeletons.
- Display error messages.

6. Example: Fetch User Data

vue

CopyEdit

```
<script setup lang="ts">
const { data: user, pending, error } = await useFetch('/api/user')
</script>

<template>
  <div v-if="pending">Loading user info...</div>
  <div v-else-if="error">Failed to load user: {{ error.message }}</div>
  <div v-else>
    <h1>{{ user.name }}</h1>
    <p>{{ user.email }}</p>
  </div>
</template>
```

Summary:

- Use useFetch and useAsyncData to load data reactively.
- Works both client-side and server-side with SSR.
- Handle loading and errors gracefully.
- Use keys in useAsyncData for caching and unique data calls.

Ready for **Chapter 11: Nuxt Configuration**?

You said:

yes

ChatGPT said:

Here’s **Chapter 11: Nuxt Configuration** explained in detail with examples.

Chapter 11: Nuxt Configuration

What is nuxt.config.ts?

The nuxt.config.ts file is the **main configuration file** for your Nuxt 4 app. It controls how Nuxt builds, runs, and behaves.

---

1. Basic Structure

ts

CopyEdit

```
export default defineNuxtConfig({
  // Configuration options here
})
```

---

2. Common Configuration Options

a) CSS

Add global CSS files or libraries:

ts

CopyEdit

```
css: [
  '~/assets/css/main.css',
  'some-package/styles.css'
],
```

b) Modules

Add Nuxt modules to extend functionality:

ts

CopyEdit

```
modules: [
  '@pinia/nuxt',
  '@nuxtjs/i18n'
],
```

c) Runtime Configuration

Separate config into **public** (client + server) and **private** (server-only):

ts

CopyEdit

```
runtimeConfig: {
  public: {
    apiBase: '/api'
  },
  apiSecret: 'my-secret' // only on server
}
```

Access in code:

ts

CopyEdit

```
const config = useRuntimeConfig()
console.log(config.public.apiBase)
```

d) Plugins

You can manually register plugins here (usually auto-imported):

ts

CopyEdit

```
plugins: [
  '~/plugins/myPlugin.ts'
]
```

e) Build Configuration

Control how your app is built:

ts

CopyEdit

```
build: {
  transpile: ['some-lib'],
  postcss: true
}
```

---

3. Example Full Config

ts

CopyEdit

```
export default defineNuxtConfig({
  css: ['~/assets/css/main.css'],
  modules: ['@pinia/nuxt', '@nuxtjs/i18n'],
  runtimeConfig: {
    public: {
      apiBase: '/api'
    },
    apiSecret: process.env.API_SECRET
  },
  build: {
    transpile: ['some-lib']
  }
})
```

---

4. Extending Config with Environment Variables

Use .env files to manage environment-specific settings, then access in runtimeConfig.

Summary:

- nuxt.config.ts controls app behavior and build.
- Configure CSS, modules, plugins, runtime config, and build options here.

- Use runtime config for environment variables.
- Modular and extendable for different project needs.

Ready for **Chapter 12: Auto-imports and Composables**?

You said:

yes

ChatGPT said:

Here's **Chapter 12: Auto-imports and Composables** explained in detail with examples.

Chapter 12: Auto-imports and Composables

1. What are Auto-imports?

Nuxt 4 automatically imports many Vue and Nuxt APIs so you don't have to manually import them in every file. This **makes your code cleaner and faster to write**.

Auto-imported APIs Examples:

- Vue APIs: ref, reactive, computed, watch, onMounted, etc.
- Nuxt composables: useRouter, useRoute, useFetch, useAsyncData, useRuntimeConfig, etc.

Example:

vue

CopyEdit

```
<script setup>
const count = ref(0)  // no need to import ref
</script>
```

2. What are Composables?

Composables are **reusable functions** that encapsulate reactive state and logic using Vue's Composition API. They help you **reuse code across components**.

3. Creating Your Own Composables

Create a file in the composables/ directory, e.g., composables/useCounter.ts

ts

CopyEdit

```
import { ref } from 'vue'

export function useCounter() {
  const count = ref(0)
  function increment() {
    count.value++
  }
  return { count, increment }
}
```

4. Using Composables in Components

vue

CopyEdit

```
<script setup>
const { count, increment } = useCounter()
</script>

<template>
  <button @click="increment">Count: {{ count }}</button>
</template>
```

5. Lifecycle Hooks in Nuxt

Nuxt supports Vue lifecycle hooks automatically, e.g.,

ts

CopyEdit

```
onMounted(() => {
  console.log('Component mounted!')
})
```

No need to import onMounted manually because of auto-imports.

Summary:

- Nuxt auto-imports Vue and Nuxt APIs, so you write less import code.
- Composables are reusable functions for state and logic.
- Create composables in the composables/ folder.
- Use lifecycle hooks like onMounted without manual import.
- Makes code modular, clean, and maintainable.

Ready for **Chapter 13: Deployment**?

You said:

yes

ChatGPT said:

Here's **Chapter 13: Deployment** explained in detail with examples.

Chapter 13: Deployment

What is Deployment?

Deployment means making your Nuxt 4 application **live on the internet** so users can access it.

---

1. Deployment Types in Nuxt 4

Nuxt 4 supports multiple deployment options:

- **Static Site Generation (SSG):** Pre-render pages as static HTML files, served from a CDN or static host (good for blogs, docs).
- **Server-Side Rendering (SSR):** Run the app on a Node.js server that dynamically renders pages on request.
- **Serverless Deployment:** Run your app as serverless functions on platforms like Vercel, Netlify.
- **Hybrid:** Use a mix of static and SSR pages.

---

2. Static Site Generation (SSG)

Generate static files with:

bash

CopyEdit

```
nuxi build

nuxi generate
```

Or configure in nuxt.config.ts:

ts

CopyEdit

```
export default defineNuxtConfig({
  nitro: {
    preset: 'static'
  }
})
```

Upload generated files (.output/public) to any static host like Netlify, GitHub Pages.

---

3. Server-Side Rendering (SSR)

Run your Nuxt app on a Node.js server:

bash

CopyEdit

```
npm run build
npm run start
```

You can deploy this on any Node server (AWS, DigitalOcean, Heroku, etc.)

4. Serverless Deployment

Nuxt supports serverless platforms like Vercel and Netlify:

- Deploy using their CLI or Git integrations.
- Nuxt builds server functions automatically.
- Fast scaling and zero maintenance.

5. Environment Variables

Use .env files and runtime config:

- Create .env file for secrets and configs.
- Access using useRuntimeConfig() in your app.

Example .env:

ini

CopyEdit

```
API_SECRET=mysecretkey
```

In nuxt.config.ts:

ts

CopyEdit

```
runtimeConfig: {
  apiSecret: process.env.API_SECRET
}
```

6. Example Deployment Flow

1. Develop your app locally.
2. Build the app (nuxi build).
3. For static: generate (nuxi generate) and upload.

4. For SSR/serverless: deploy server with hosting provider.

Summary:

- Nuxt 4 supports static, SSR, and serverless deployment modes.
- Choose based on your app needs (static for speed, SSR for dynamic content).
- Use environment variables for config and secrets.
- Deployment steps differ based on chosen mode.

Ready for **Chapter 14: TypeScript Support**?

You said:

yes

ChatGPT said:

Here's **Chapter 14: TypeScript Support** explained in detail with examples.

Chapter 14: TypeScript Support

What is TypeScript?

TypeScript (TS) is a **typed superset of JavaScript** that adds static types to your code, helping catch errors early and improving developer experience with better autocomplete and documentation.

---

1. Setting Up TypeScript in Nuxt 4

Nuxt 4 has **built-in TypeScript support**, so starting a new project with TypeScript is straightforward.

You can create a new project with TypeScript:

bash

CopyEdit

```
npx nuxi init my-nuxt-app --ts

cd my-nuxt-app
```

```
npm install
```

Or add TypeScript to an existing project:

bash

CopyEdit

```
npm install --save-dev typescript
npm install --save-dev @types/node
```

Create a tsconfig.json file if not already present.

---

2. Type Safety in Components

Use .vue files with <script setup lang="ts">:

vue

CopyEdit

```
<script setup lang="ts">
const msg: string = 'Hello, Nuxt + TS!'
</script>

<template>
  <p>{{ msg }}</p>
</template>
```

---

3. Types in Pinia Stores

Pinia works well with TypeScript:

ts

CopyEdit

```
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0 as number
  }),
  actions: {
    increment() {
      this.count++
    }
  }
})
```

---

4. TypeScript in API Routes

Define event handlers with explicit types:

ts

CopyEdit

```
import type { H3Event } from 'h3'

export default defineEventHandler((event: H3Event) => {
  // Your logic here
  return { message: 'Hello from typed API!' }
})
```

---

5. Benefits of Using TypeScript

- Catch type errors before running code
- Better editor autocomplete and hints
- Easier maintenance and refactoring
- Better collaboration in teams

---

6. Common Tips

- Use lang="ts" in Vue script tags.

- Define types for props, data, and methods.
- Use interfaces and types for complex data.
- Take advantage of Nuxt's typed composables and helpers.

Summary:

- Nuxt 4 supports TypeScript out of the box.
- Use .vue with <script setup lang="ts"> for typed components.
- Pinia and server API routes work smoothly with TS.
- TypeScript improves reliability and developer experience.

Ready for **Chapter 15: Meta Tags, SEO, and Head Management**?

You said:

yes

ChatGPT said:

Here’s **Chapter 15: Meta Tags, SEO, and Head Management** explained in detail with examples.

Chapter 15: Meta Tags, SEO, and Head Management

Why Meta Tags and SEO Matter?

Meta tags help define page information (title, description, keywords) that search engines use to index your site and improve ranking. Proper SEO helps your site appear in search results and improves user experience.

---

1. Managing Head Tags in Nuxt 4

Nuxt 4 uses the useHead composable to manage <head> tags reactively in your components and pages.

---

2. Basic Example of useHead

vue

CopyEdit

```
<script setup>
useHead({
  title: 'My Nuxt App',
  meta: [
    { name: 'description', content: 'This is my Nuxt 4 app' },
    { name: 'keywords', content: 'nuxt, vue, seo' }
  ]
})
</script>
```

3. Dynamic Meta Tags

You can set dynamic meta based on data or props:

vue

CopyEdit

```
<script setup>
const postTitle = 'Nuxt SEO Guide'
useHead({
  title: postTitle,
  meta: [
    { name: 'description', content: `Read about ${postTitle}` }
  ]
})
</script>
```

4. Open Graph and Twitter Cards

For social media sharing, add Open Graph (OG) and Twitter meta tags:

vue

CopyEdit

```vue
<script setup>
useHead({
  meta: [
    { property: 'og:title', content: 'My Nuxt App' },
    { property: 'og:description', content: 'Great Nuxt app' },
    { property: 'og:image', content: '/social-image.png' },
    { name: 'twitter:card', content: 'summary_large_image' }
  ]
})
</script>
```

---

5. SEO Best Practices

- Use unique, descriptive titles for each page.
- Provide clear meta descriptions.
- Include relevant keywords naturally.
- Use canonical URLs to avoid duplicate content.
- Use semantic HTML tags for content structure.

---

6. Using head in Page Components

You can also define head inside page components using the definePageMeta or head property.

Example with definePageMeta:

ts

CopyEdit

```ts
export default definePageMeta({
  head: {
    title: 'Page Title',
    meta: [{ name: 'description', content: 'Page description' }]
  }
})
```

Summary:

- Use useHead to set page title and meta tags reactively.
- Manage SEO-friendly meta tags like descriptions, keywords, and social sharing tags.
- Dynamically set meta tags based on page data.
- Follow SEO best practices to improve search ranking.

Ready for **Chapter 16: Dynamic Routing and Navigation Guards**?

You said:

yes

ChatGPT said:

Here’s **Chapter 16: Dynamic Routing and Navigation Guards** explained in detail with examples.

Chapter 16: Dynamic Routing and Navigation Guards

---

1. Dynamic Routes with Parameters

Nuxt 4 supports **file-based routing** with dynamic route parameters.

- Create dynamic routes by adding brackets around the filename:
  Example: pages/user/[id].vue
- The [id] is a **route parameter** accessible in the page.

Example: pages/user/[id].vue

vue

CopyEdit

```
<script setup>
const route = useRoute()
const userId = route.params.id
</script>


<template>
  <h1>User ID: {{ userId }}</h1>
</template>
```

If you visit /user/123, userId will be '123'.

---

2. Query Parameters

Query params are accessible via useRoute():

Example: URL /search?term=vue

ts

CopyEdit

```
const route = useRoute()
const searchTerm = route.query.term
```

---

3. Navigation Guards using Middleware

Middleware acts as **navigation guards** to control access.

- Middleware runs before route enters.
- Can redirect, allow, or block navigation.

Example: Simple auth middleware (middleware/auth.ts):

ts

CopyEdit

```
export default defineNuxtRouteMiddleware((to, from) => {
  const isLoggedIn = false // Replace with auth logic
  if (!isLoggedIn && to.name !== 'login') {
    return navigateTo('/login')
  }
})
```

Apply middleware to pages:

ts

CopyEdit

```
definePageMeta({
  middleware: 'auth'
})
```

---

4. Redirects and Route Protection

You can **redirect users** based on route conditions.

Example:

ts

CopyEdit

```
export default defineNuxtRouteMiddleware((to) => {
  if (to.path === '/admin' && !userIsAdmin()) {
    return navigateTo('/not-authorized')
  }
})
```

Summary:

- Dynamic routes use [param] syntax in filenames.
- Access params via useRoute().params and query via useRoute().query.
- Middleware functions as navigation guards to protect routes.
- Middleware can redirect or block navigation based on conditions.

Ready for **Chapter 17: Transitions and Animations**?

You said:

yes

ChatGPT said:

Here's **Chapter 17: Transitions and Animations** explained in detail with examples.

Chapter 17: Transitions and Animations

---

1. Page Transitions

Nuxt 4 allows you to add smooth transitions when navigating between pages.

- Define transition in app.vue or on individual pages.
- Use the page-transition property or <Transition> component.

Example: Simple fade transition in app.vue

vue

CopyEdit

```
<template>
  <NuxtPage :key="$route.fullPath" v-slot="{ Component }">
    <Transition name="fade" mode="out-in">
      <component :is="Component" />
    </Transition>
  </NuxtPage>
</template>


<style>
.fade-enter-active, .fade-leave-active {
  transition: opacity 0.5s;
}
.fade-enter-from, .fade-leave-to {
  opacity: 0;
}
</style>
```

---

2. Component Animations

You can animate individual components with Vue’s <Transition> and <TransitionGroup>.

Example: Animate list items entering and leaving:

vue

CopyEdit

```
<template>
  <TransitionGroup name="list" tag="ul">
    <li v-for="item in items" :key="item.id">{{ item.name }}</li>
  </TransitionGroup>
</template>

<style>
.list-enter-active, .list-leave-active {
  transition: all 0.3s ease;
}
.list-enter-from, .list-leave-to {
  opacity: 0;
  transform: translateY(20px);
}
</style>
```

---

3. Using Vue Transition and TransitionGroup

- <Transition> works for a single element/component.
- <TransitionGroup> works for lists with multiple elements.
- You can customize animations using CSS or JavaScript hooks.

---

4. Advanced Animations

You can combine Vue transitions with animation libraries like **GSAP** or **Animate.css** for richer effects.

Summary:

- Use <Transition> for page or component entry/exit animations.
- Use <TransitionGroup> for list animations.
- Customize animation timing and styles with CSS.
- Combine with libraries for advanced effects.

Ready for **Chapter 18: Internationalization (i18n)**?

You said:

yes

ChatGPT said:

Here’s **Chapter 18: Internationalization (i18n)** explained in detail with examples.

Chapter 18: Internationalization (i18n)

What is Internationalization?

Internationalization (i18n) means making your Nuxt app **support multiple languages** so users from different countries can use it comfortably.

---

1. Setting up Multilingual Support

Nuxt 4 uses the **nuxt-i18n module** to add multilingual capabilities easily.

---

2. Installing nuxt-i18n

Add the module via:

bash

CopyEdit

```
npm install @nuxtjs/i18n
```

Then add it in nuxt.config.ts:

ts

CopyEdit

```
export default defineNuxtConfig({
  modules: ['@nuxtjs/i18n'],
  i18n: {
    locales: [
      { code: 'en', name: 'English' },
      { code: 'fr', name: 'Français' }
    ],
    defaultLocale: 'en',
    vueI18n: {
      legacy: false,
      locale: 'en',
      messages: {
        en: { welcome: 'Welcome' },
        fr: { welcome: 'Bienvenue' }
      }
    }
  }
})
```

---

3. Using Translations in Templates

Use the $t function in templates:

vue

CopyEdit

```
<template>
  <h1>{{ $t('welcome') }}</h1>
</template>
```

It will display “Welcome” or “Bienvenue” based on the selected language.

4. Switching Languages

You can switch languages programmatically:

ts

CopyEdit

```
const { locale } = useI18n()
locale.value = 'fr'  // switch to French
```

Or with UI buttons:

vue

CopyEdit

```
<button @click="locale = 'en'">English</button>
<button @click="locale = 'fr'">French</button>
```

5. Lazy Loading Language Files

For large apps, load language files only when needed using lazy loading options in nuxt-i18n.

Summary:

- Use nuxt-i18n to support multiple languages easily.
- Define locales and messages in nuxt.config.ts.
- Use $t() to display translated strings.
- Switch locales dynamically in code or UI.
- Supports lazy loading for performance.

Ready for **Chapter 19: Testing Nuxt Applications**?

You said:

yes

ChatGPT said:

Here’s **Chapter 19: Testing Nuxt Applications** explained in detail with examples.

Chapter 19: Testing Nuxt Applications

Why Test?

Testing helps ensure your Nuxt app works correctly, catches bugs early, and improves code quality and maintainability.

---

1. Unit Testing Components

Unit tests check individual components in isolation.

- Use testing frameworks like **Vitest** or **Jest**.
- Use Vue Test Utils for mounting and interacting with components.

Example: Simple component test with Vitest

ts

CopyEdit

```
import { mount } from '@vue/test-utils'
import HelloWorld from '~/components/HelloWorld.vue'
import { describe, it, expect } from 'vitest'

describe('HelloWorld', () => {
  it('renders properly', () => {
    const wrapper = mount(HelloWorld)
    expect(wrapper.text()).toContain('Hello World')
  })
})
```

2. End-to-End (E2E) Testing

E2E tests simulate real user interactions with the full app.

- Tools: **Cypress**, **Playwright**.
- Tests user flows like logging in, navigation, form submission.

3. Testing Server Routes (API)

Test server API routes using tools like **Supertest** to simulate HTTP requests.

Example:

ts

CopyEdit

```
import request from 'supertest'
import { setupServer } from 'some-server-setup'

describe('API tests', () => {
  it('GET /api/hello returns 200', async () => {
    const server = setupServer()
    const res = await request(server).get('/api/hello')
    expect(res.statusCode).toBe(200)
    expect(res.body.message).toBe('Hello')
  })
})
```

4. Setup Testing Environment

- Install testing dependencies:

bash

CopyEdit

```
npm install -D vitest @vue/test-utils
```

- Configure Vitest in vite.config.ts or package.json.

5. Running Tests

Run tests via:

bash

CopyEdit

```
npm run test
```

or

bash

CopyEdit

```
npx vitest
```

Summary:

- Write unit tests to verify component behavior.
- Use E2E tests to simulate user flows.
- Test server routes separately.
- Use modern testing tools like Vitest and Cypress.
- Automated tests improve code reliability.

Ready for **Chapter 20: Performance Optimization and Best Practices**?

You said:

yes

ChatGPT said:

Here’s **Chapter 20: Performance Optimization and Best Practices** explained in detail with examples.

Chapter 20: Performance Optimization and Best Practices

1. Lazy Loading Components and Routes

Lazy loading helps load only what’s needed, improving initial load time.

Lazy load components:

vue

CopyEdit

```
<script setup>
const LazyComponent = defineAsyncComponent(() => import('~/components/MyComponent.vue'))
</script>

<template>
  <LazyComponent />
</template>
```

Lazy load routes: Nuxt automatically code-splits pages, so each route loads on demand.

---

2. Code Splitting

Nuxt 4 automatically splits your JavaScript into smaller chunks, loaded as needed, which speeds up the app.

You can customize splitting in nuxt.config.ts if needed.

---

3. Using Caching and CDN

- Use CDN (Content Delivery Network) to serve static assets faster globally.
- Enable HTTP caching headers for API and assets.
- Leverage browser caching to reduce repeated downloads.

---

4. Security Tips

- Sanitize user input to avoid XSS attacks.
- Use HTTPS to encrypt traffic.
- Protect sensitive environment variables using runtime config.

- Use middleware to protect routes.
- Keep dependencies up to date.

---

5. Other Best Practices

- Use defineProps and defineEmits for better type inference.
- Avoid heavy computations on the client; move them to server or build time.
- Use modern image formats (like WebP) and optimize images.
- Minimize use of large third-party libraries.
- Monitor performance with tools like Lighthouse.

Summary:

- Lazy load components and routes for faster load times.
- Use CDN and caching for efficient asset delivery.
- Follow security best practices to protect your app.
- Write clean, maintainable, and optimized code.