

Powering Multi-Tenant Applications with Nuxt

Adam DeHaven



Adam DeHaven

Senior Staff Engineer @ Kong



@adamdehaven

What is Multi-Tenancy?

Build Once, Serve Many

What is Multi-Tenancy?



Shared codebase and infrastructure

The app is built and deployed utilizing a single codebase, drastically reducing maintenance overhead and ensuring all tenants benefit from new features and bug fixes simultaneously.



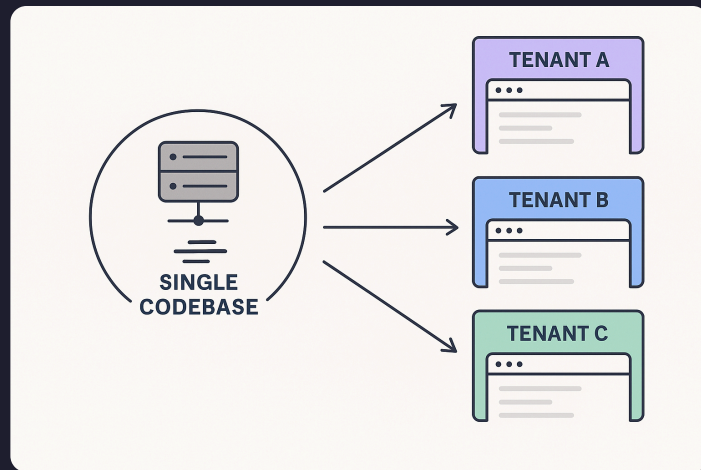
Isolated tenant data

Typically achieved through a combination of storage partitioning and application-level access controls.



Content and UX customizable per-tenant

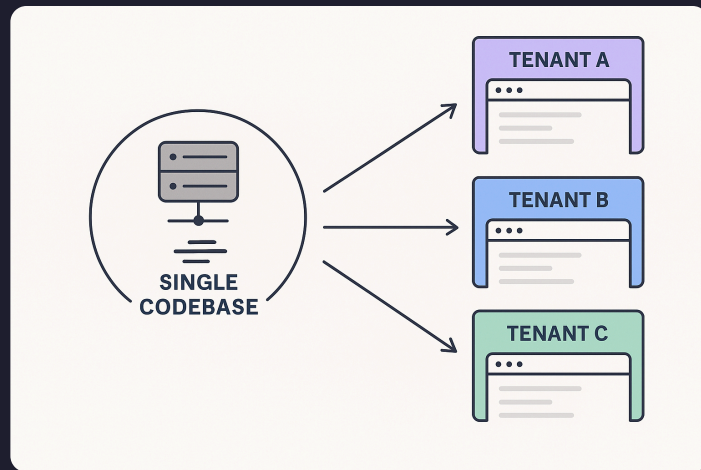
Tenants can fully customize the content, layout, and visual elements of the app, creating unique branded experiences for their users.



What is Multi-Tenancy?

More Tenants, More Complexity

- Handling tenant-specific content and configurations.
- Maintaining separate branding and UX per tenant.
- Managing different authentication strategies and authorization across tenants.
- Ensuring performance with varying traffic loads.



Many teams end up with separate codebases or end up performing separate builds for each tenant which can quickly become unmanageable and very expensive.



Nuxt Makes It Easy

Nuxt Makes Multi-Tenancy Easy



Runtime Configuration Capabilities

Allows for dynamic loading of tenant-specific settings and features.



Nitro Server Engine

Powerful plugins, middleware, and event handlers enable Nuxt to seamlessly read and write to storage providers, proxy requests to external APIs, and implement tenant-aware request handling and caching strategies.



Nuxt Module Ecosystem

Easily extend functionality and add tenant-specific features based on dynamic configuration.



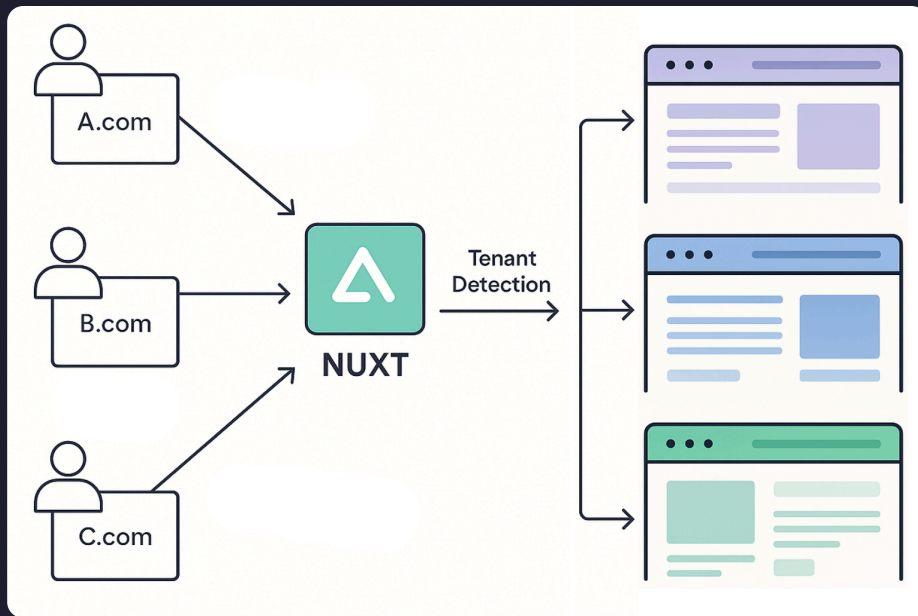
SSR & SEO

Server-side rendering and built-in head management for tenant content and metadata.



Edge Deployment Compatibility

Nuxt apps can be deployed at the edge, dramatically reducing latency while enabling powerful caching strategies that scale automatically.



Tenant Detection

Powered by ⚡ Nitro

Tenant Detection



To get started, we will create a few helpers to detect the requested tenant:



Custom \$fetch instance

Make API requests from the client to our Nitro server while automatically attaching the request headers



Custom useFetch composable

Replace usage of separate `$fetch` and `useAsyncData` calls with a simpler composable



Nitro Plugin

Extract the tenant info from the custom headers and add it to the event context to make it available to all server routes and middleware

Tenant Detection

Custom \$fetch plugin

```
export default defineNuxtPlugin((nuxtApp) => {
  const requestURL = useRequestURL()
  const requestHeaders = useRequestHeaders()

  const api = $fetch.create({
    onRequest({ request, options, error }) {
      options.headers = {
        ...(options.headers || {}),
        // Always include the original client request headers
        ...requestHeaders,
        // Add custom tenant-specific headers to all requests that go through our $fetch instance
        ...{ 'x-tenant-hostname': requestURL.hostname },
      }
    }
  })

  return {
    provide: { api }, // Expose helper to useNuxtApp().$api
  }
})
```

Tenant Detection

Custom \$fetch

With our Nuxt plugin registered, `$api` is exposed from `useNuxtApp()` to make API calls directly from our Vue components and other app code.

We can also wrap `$api` with `useAsyncData` to avoid double-fetching data during server-side rendering (server & client on hydration).

▼ app/pages/[...page].vue

```
<script setup lang="ts">
const { $api } = useNuxtApp()
const route = useRoute()
const pageKey = computed(() => `page-${route.path.replace(/\\/g, '-')}`)

const { data: pageData } = await useAsyncData(
  pageKey,
  // Use our `api` instance to ✨automatically✨ attach our custom tenant headers
  () => $api(`/api/pages/${encodeURIComponent(route.path)}`),
)
</script>
```

Tenant Detection

Custom useFetch composable

Instead of using `useAsyncData` and `$api` directly, we can also create a custom composable that wraps `useFetch` which can be used in our components and pages.

TS app/composables/useApi.ts

```
import type { UseFetchOptions } from 'nuxt/app'

export function useApi<T>(  
  url: string | (() => string),  
  options?: UseFetchOptions<T>,  
) {  
  return useFetch(url, {  
    ...options,  
    $fetch: useNuxtApp().$api as typeof $fetch  
  })  
}
```

Tenant Detection

Custom useFetch composable

Let's refactor our previous code to utilize our new `useApi` composable.

```
<script setup lang="ts">
const route = useRoute()
const pagePath = computed(() => encodeURIComponent(route.path))

const { data: pageData } = await useApi(() => `/api/pages/${pagePath.value}`)
</script>
```

Tenant Detection

Nitro Plugin

Now that we have the original client request headers, we can access them on the Nitro server to identify the tenant and attach the tenant info to the `event.context`.

```
// server/plugins/00.tenant-context.ts

export default defineNitroPlugin((nitroApp) => {
  nitroApp.hooks.hook('request', async (event) => {
    // Get the custom tenant client headers along with the host header for dynamic request routing.
    const xTenantHostname = getRequestHeader(event, 'x-tenant-hostname')
    const hostHeader = getRequestHeader(event, 'host')

    const hostname = String(xTenantHostname || hostHeader || '')
      .replace(/\$/ , '').split(':')[0] || ''
    const id = await useStorage('kv').getItem<string>(`tenant:id:${tenant.hostname}`)

    // Attach tenant data to the event context to make available to all server routes and middleware.
    event.context.tenant = { id, hostname, /* etc... */ }
  })
})
```

Tenant Detection



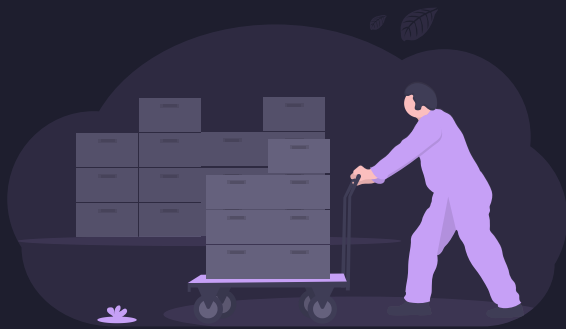
We can also augment the `H3EventContext`` interface to include our new ``tenant`` properties for TypeScript support throughout our server files.

TS server/event-context.d.ts

```
import type { H3EventContext } from 'h3'

declare module 'h3' {
  interface H3EventContext {
    tenant: {
      /** The tenant ID. */
      id: string
      /** The tenant client hostname. */
      hostname: string
    }
  }
}

export {}
```



Accessing Tenant Data

Tenant-Aware Storage

Accessing Tenant Data



Let's say that we want to fetch the tenant's theme name from their configuration data and add it to the `<head>` of our application with a Nuxt plugin.

The attribute can then be utilized in our CSS to apply the style rules for the tenant's theme.

```
:root[data-theme="midnight"] {  
  --color-background: #181825;  
  --color-text: #fff;  
}  
  
:root[data-theme="default"] {  
  --color-background: #fff;  
  --color-text: #111;  
}
```

Accessing Tenant Data

server/api/config.get.ts

```
export default defineCachedEventHandler(async (event) => {
  const { id, hostname } = event.context.tenant
  // Retrieve tenant configuration from KV storage using the tenant's ID as the key
  const config = await useStorage('kv').getItem<TenantConfig>(`tenant:config:${id}`)

  if (!config) {
    // Alternatively, you could fallback to fetching the config from an API
    throw createError({
      statusCode: 404,
      statusMessage: `Tenant config not found for '${hostname}'`
    })
  }
  return config
}, {
  getKey: (event: H3Event) => `tenant:config:${event.context.tenant.id}`, // Unique cache key
  swr: true, // Serve a stale cached response while asynchronously revalidating it
  maxAge: 60 * 60, // Cache for 1 hour
  // Important: Prevent discarding tenant-specific headers
  // An array of request headers to be considered for the cache
  varies: ['host', 'x-forwarded-host'],
})
```



Proxying Requests

Tenant-Aware Request Routing

Proxying Requests

server/api/external/[...].ts

```
import { joinURL } from 'ufo'

export default defineEventHandler(async (event) => {
  // Remove the `/external/api` prefix from the path
  const targetPath = event.path.replace(/^\/api\/external\/$/, '')
  // Set the request target utilizing our external API's base URL and the tenant ID
  const target = joinURL('https://api.app.com', event.context.tenant.id, targetPath)
  // Determine the request body when applicable
  const requestBody = ['PATCH', 'POST', 'PUT', 'DELETE'].includes(event.method)
    ? await readRawBody(event, false) : undefined

  return sendProxy(event, target, {
    headers: {
      // ...Add necessary request headers as needed, e.g. `Cookie`, `Accept`, `Content-Type`, etc.
    },
    fetchOptions: {
      method: event.method,
      body: requestBody,
    },
  })
})
```

Proxying Requests



Now that we have set up our proxy route, we can use it in our pages and components to fetch tenant-specific data from our external API.

▼ app/pages/users.vue

```
<template>
  <UserCard
    v-for="user in allUsers"
    :key="user.id"
    :user="user"
  />
</template>

<script setup lang="ts">
// Request is proxied to `https://api.app.com/<tenant_id>/users`
const { data: allUsers } = await useApi('/api/external/users')
</script>
```



App Customization

Runtime Configuration and Integrations

Runtime Configuration



Nuxt's modular architecture and runtime configuration capabilities are a perfect fit for customizing multi-tenant applications.



SEO Optimizations

Open Graph metadata, images, and crawler assets such as sitemaps & robots.txt.



Performance & Analytics Integrations

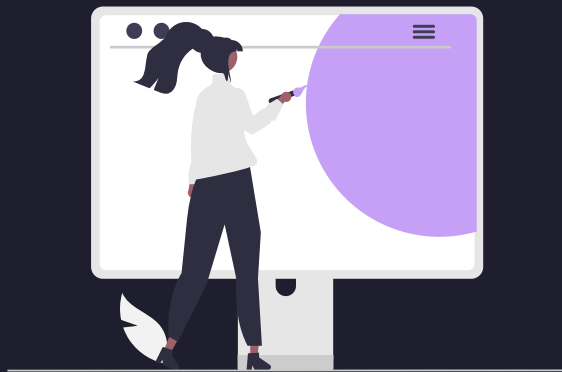
Seamlessly integrate 3rd party tools without needing to build or deploy.



Dynamic Feature Enablement

Enable or disable features based on tenant preferences, subscription level, A/B testing, or product-driven feature flags.





Brand Customization

Tenant-Specific Look & Feel

Tenant Branding



We can update our existing ``tenant-theme`` plugin to apply the tenant's custom stylesheets, including CSS Custom Properties, fonts, and colors, to customize the app's look and feel.

```
export default defineNuxtPlugin({
  name: 'tenant-theme',
  async setup() {
    const { data: tenantConfig } = await useApi('/api/config')

    useHead({
      // Add a `data-theme` attribute to the root <html> element
      htmlAttrs: { 'data-theme': tenantConfig.value?.theme?.name || 'default' },
      style: [{
        innerHTML: () => sanitizeCustomCss(tenantConfig.value?.css),
        tagPosition: 'head',
      }],
    })
  },
})
```

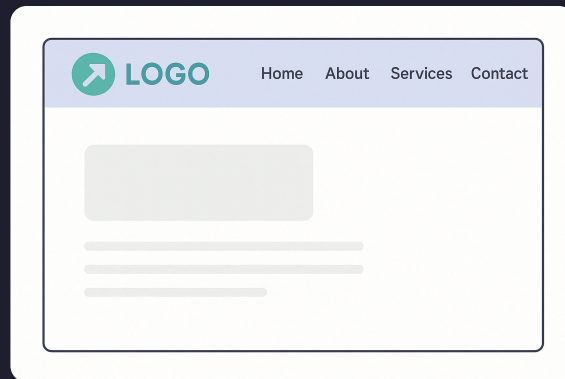
Tenant Branding

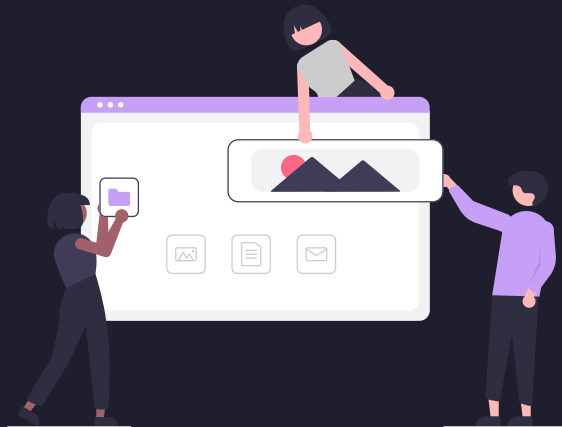


For other brand elements, such as the logo, we can easily fetch assets from dedicated, tenant-aware endpoints at runtime.

✓ app/components/page/PageHeader.vue

```
<template>
  <header>
    <NuxtLink to="/">
      
    </NuxtLink>
  </header>
</template>
```





Custom Content

Dynamic Pages, Routes, and Components

Custom Content Requirements



Intuitive content management

Page content must be editable by the tenant in an easy-to-use format.



Dynamic content organization

Page paths must be fully dynamic, empowering tenants to design intuitive navigation systems that best serve their users.



Highly-customizable components & page layouts

The app must offer components that allow for building complex page layouts that seamlessly align with their brand identity.



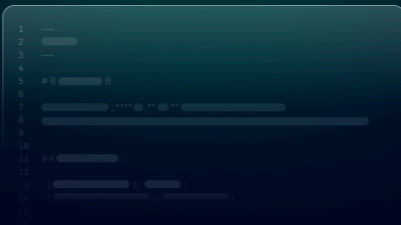


The git-based CMS for Nuxt projects.

Write content in Markdown, YAML, or JSON files, query and display them in your Nuxt application.

Nuxt Content

Close, but not quite what we need



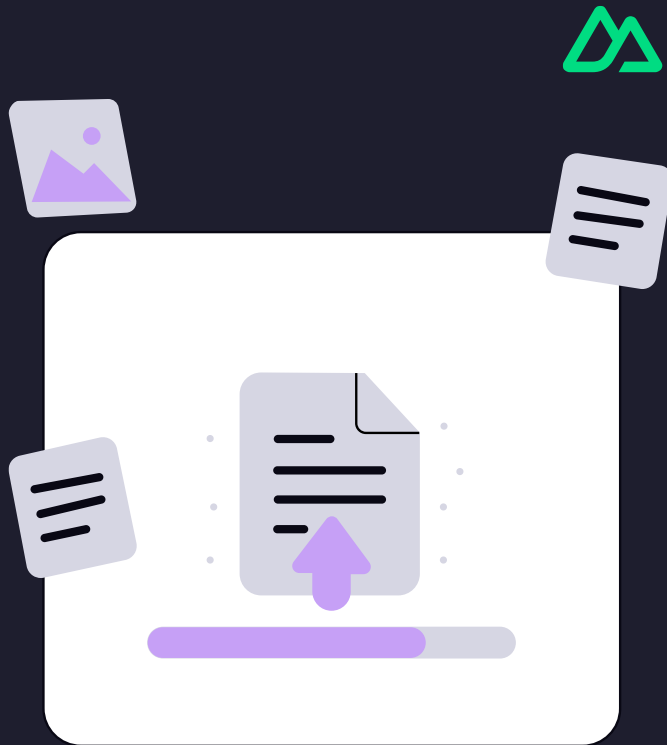
Supercharge regular Markdown to write documents
interacting deeply with any Vue component.

Intuitive content management

Markdown & MDC Syntax

Dynamic content organization

- Provide API endpoints for tenants to manage their MDC pages in our storage provider.
- Tenants will push their pages to our API, mapping the file structure in their directory to path-based logic that will render the page in the app:
 - A file at ``about/team.md`` would be served at ``/about/team`` in the application.
- We store the page document under a tenant-specific key (e.g. Cloudflare R2):
``<tenant_id>:<page_path>``



Dynamic content organization



```
<template>
  <MDCRenderer v-if="ast?.body" :body="ast.body" :data="ast.data" />
</template>

<script setup lang="ts">
import { parseMarkdown } from '@nuxtjs/mdc/runtime'

const route = useRoute()
const pagePath = computed(() => encodeURIComponent(route.path))
const { data: pageData, error: pageError } = await useApi(() => `/api/pages/${pagePath.value}`)

if (pageError.value) {
  throw createError({ statusCode: 404, statusMessage: 'Not Found', fatal: true })
}

const { data: ast } = await useAsyncData(computed(() => `parsed-page-${route.path}`),
  () => parseMarkdown(pageData.value),
  { immediate: !!pageData.value, watch: [pageData] },
)
</script>
```



Highly-customizable components & page layouts

Markdown & MDC Syntax

Highly-customizable components & page layouts



```
<template>
  <component :is="tag" class="page-container">
    <MDCSlot name="default" />
  </component>
</template>

<script setup lang="ts">
const {
  tag = 'div', color, background, margin, /* ...etc. */
} = defineProps<PageContainerProps>()
</script>

<style scoped>
.page-container {
  background: v-bind('background');
  color: v-bind('color');
  margin: v-bind('margin');
  /* ...etc. */
}
</style>
```



Highly-customizable components & page layouts



```
::page-container
```

```
---
```

```
color: "var(--color-text-secondary)"  
background: "var(--color-background)"  
border: "1px solid var(--color-border)"  
border-radius: "8px"  
padding: "20px"
```

```
---
```

```
### Welcome to MadVue 2025!
```

```
Join us for a full day of talks, activities, and  
networking with the Vue.js community!
```

```
::
```

Welcome to MadVue 2025!

Join us for a full day of talks, activities, and networking with the Vue.js community!

Highly-customizable components & page layouts



```
:: page-hero
---
image: "https://images.unsplash.com/photo-1543783207-ec64e4
color: "var(--color-text-inverse)"
padding: "100px 20px"
text-align: "center"
---
#title
Welcome to
[Mad]{style="color: var(--color-text-primary)}"
[Vue]{style="color: var(--color-text-tertiary)}"
2025!

#description
Join us for a full day of talks, activities, and
networking with the Vue.js community!

#actions
::: button{ appearance="primary" }
Get your ticket →
:::

::: button{ appearance="secondary" }
Sponsors
:::
::
```

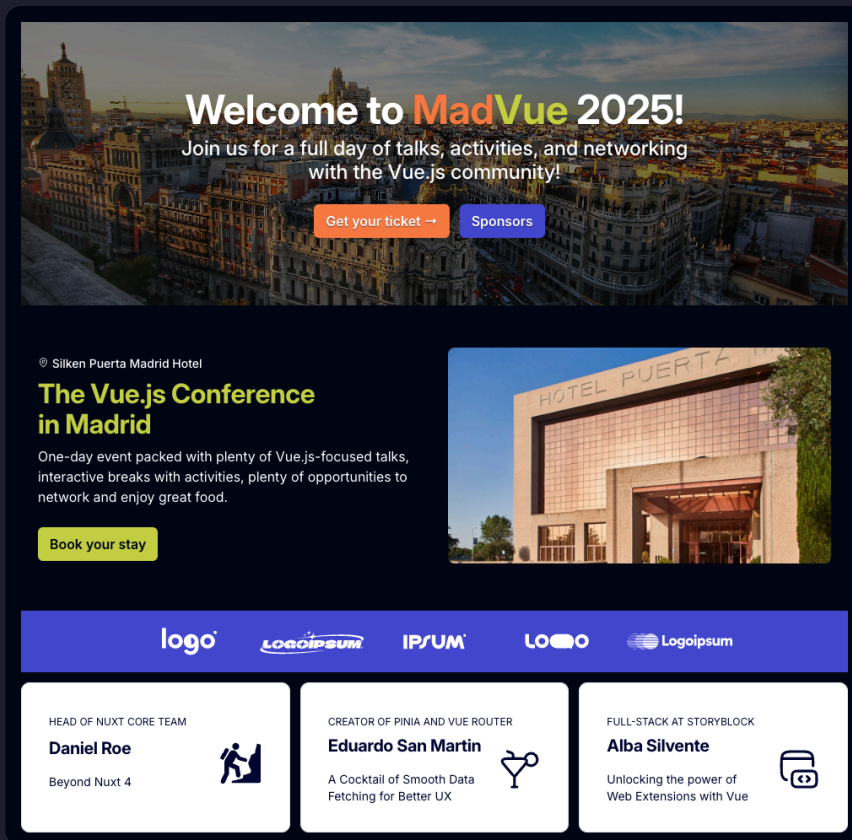


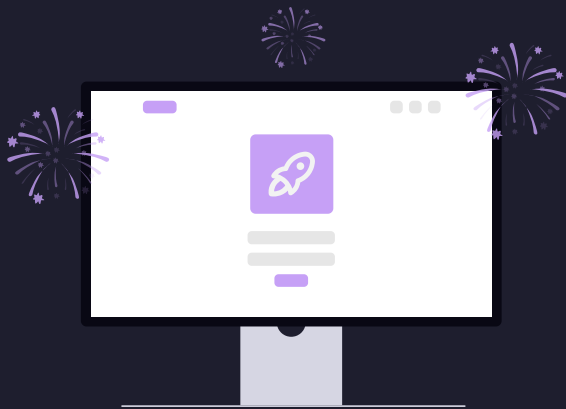
Highly-customizable components & page layouts



You can create a library of components that cover a wide range of use-cases that are fully customizable by your tenants.

Since each MDC component is customizable, every tenant app can have a unique look and feel, while relying on the same core components.





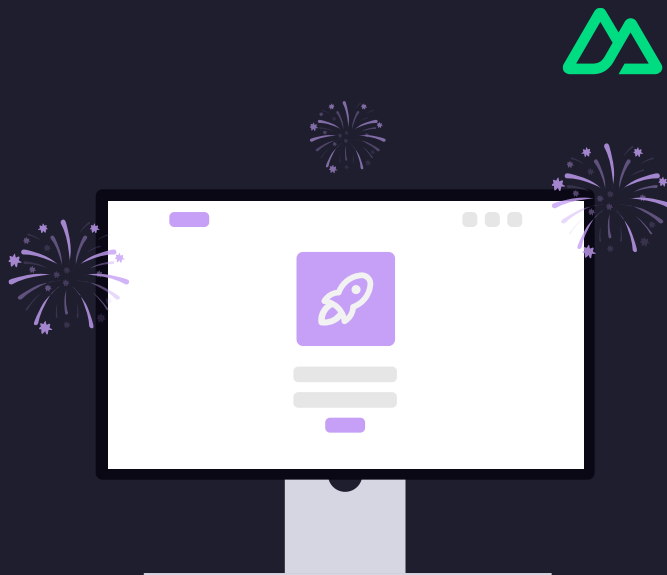
Key Takeaways

Key Takeaways

- Nuxt's **server-driven architecture** provides an ideal foundation for multi-tenancy, enabling a single codebase to serve unlimited tenant organizations.
- **Runtime configuration capabilities** allow for dynamic tenant customization without separate builds or deployments.
- The **MDC module alongside tenant-aware data retrieval** can deliver fully-customizable content and components while maintaining a unified codebase.

Multi-tenancy doesn't have to be complex when you leverage Nuxt's architecture and ecosystem.

Build once, deploy once, **customize infinitely**.



Adam DeHaven

Thanks for having me, Madrid!

bit.ly/madvue-2025



@adamdehaven