useNuxtApp

`useNuxtApp` is a built-in composable that provides a way to access shared runtime context of Nuxt, which is available on both client and server side. It helps you access the Vue app instance, runtime hooks, runtime config variables and internal states, such as `ssrContext` and `payload`.

You can use `useNuxtApp()` within composables, plugins and components.

app.vue

```vue
<script setup lang="ts">
const nuxtApp = useNuxtApp()
</script>
```

Methods

provide (name, value)

`nuxtApp` is a runtime context that you can extend using Nuxt plugins. Use the `provide` function to create Nuxt plugins to make values and helper methods available in your Nuxt application across all composables and components.

`provide` function accepts `name` and `value` parameters.

Example:

```ts
const nuxtApp = useNuxtApp()
nuxtApp.provide('hello', (name) => `Hello ${name}!`)

// Prints "Hello name!"
```

```
console.log(nuxtApp.$hello('name'))
```

As you can see in the example above, `$hello` has become the new and custom part of `nuxtApp` context and it is available in all places where `nuxtApp` is accessible.

hook(name, cb)

Hooks available in `nuxtApp` allows you to customize the runtime aspects of your Nuxt application. You can use runtime hooks in Vue.js composables and Nuxt plugins to hook into the rendering lifecycle.

`hook` function is useful for adding custom logic by hooking into the rendering lifecycle at a specific point.
`hook` function is mostly used when creating Nuxt plugins.

See Runtime Hooks for available runtime hooks called by Nuxt.

plugins/test.ts

```
export default defineNuxtPlugin((nuxtApp) => {
  nuxtApp.hook('page:start', () => {
    /* your code goes here */
  })
  nuxtApp.hook('vue:error', (..._args) => {
    console.log('vue:error')
    // if (process.client) {
    //   console.log(..._args)
    // }
  })
})
```

callHook(name, ...args)

`callHook` returns a promise when called with any of the existing hooks.

```
await nuxtApp.callHook('my-plugin:init')
```

Properties

`useNuxtApp()` exposes the following properties that you can use to extend and customize your app and share state, data and variables.

vueApp

`vueApp` is the global Vue.js application instance that you can access through `nuxtApp`. Some useful methods:

- **component()** - Registers a global component if passing both a name string and a component definition, or retrieves an already registered one if only the name is passed.
- **directive()** - Registers a global custom directive if passing both a name string and a directive definition, or retrieves an already registered one if only the name is passed(example).
- **use()** - Installs a Vue.js Plugin (example).

> 👉 Read more in https://vuejs.org/api/application.html#application-api.

ssrContext

`ssrContext` is generated during server-side rendering and it is only available on the server side. Nuxt exposes the following properties through `ssrContext`:

- `url` (string) - Current request url.
- `event` (unjs/h3 request event) - Access to `req` and `res` objects for the current request.
- `payload` (object) - NuxtApp payload object.

payload

`payload` exposes data and state variables from server side to client side. The following keys will be available on the client after they have been passed from the server side:

- **serverRendered** (boolean) - Indicates if response is server-side-rendered.
- **data** (object) - When you fetch the data from an API endpoint using either `useFetch` or `useAsyncData`, resulting payload can be accessed from the `payload.data`. This data is cached and helps you prevent fetching the same data in case an identical request is made more than once.

app.vue

```vue
<script setup lang="ts">
const { data } = await useAsyncData('count', () => $fetch('/api/count'))
</script>
```

After fetching the value of `count` using `useAsyncData` in the example above, if you access `payload.data`, you will see `{ count: 1 }` recorded there. The value of `count` is updated whenever the page count increases.

When accessing the same `payload.data` from ssrcontext, you can access the same value on the server side as well.

- **state** (object) - When you use `useState` composable in Nuxt to set shared state, this state data is accessed through `payload.state.[name-of-your-state]`.

plugins/my-plugin.ts

```ts
export const useColor = () => useState<string>('color', () => 'pink')

export default defineNuxtPlugin((nuxtApp) => {
  if (process.server) {
    const color = useColor()
  }
})
```

It is also possible to use more advanced types, such as `ref`, `reactive`, `shallowRef`, `shallowReactive` and `NuxtError`.

You can also add your own types, with a special plugin helper:

plugins/custom-payload.ts

```ts
  /**
   * This kind of plugin runs very early in the Nuxt lifecycle, before we revive the payload.
   * You will not have access to the router or other Nuxt-injected properties.
   */
export default definePayloadPlugin((nuxtApp) => {
  definePayloadReducer('BlinkingText', data => data === '<blink>' && '_')
  definePayloadReviver('BlinkingText', () => '<blink>')
})
```

isHydrating

Use `nuxtApp.isHydrating` (boolean) to check if the Nuxt app is hydrating on the client side.

Example:

components/nuxt-error-boundary.ts

```ts
export default defineComponent({
  setup (_props, { slots, emit }) {
    const nuxtApp = useNuxtApp()
    onErrorCaptured((err) => {
      if (process.client && !nuxtApp.isHydrating) {
        // ...
      }
    })
  }
})
```

runWithContext

You are likely here because you got a "Nuxt instance unavailable" message. Please use this method sparingly, and report examples that are causing issues, so that it can ultimately be solved at the framework level.

The `runWithContext` method is meant to be used to call a function and give it an explicit Nuxt context. Typically, the Nuxt context is passed around implicitly and you do not need to worry about this. However, when working with complex `async / await` scenarios in middleware/plugins, you can run into instances where the current instance has been unset after an async call.

```ts
export default defineNuxtRouteMiddleware(async (to, from) => {
  const nuxtApp = useNuxtApp()
  let user
  try {
    user = await fetchUser()
    // the Vue/Nuxt compiler loses context here because of the try/catch block.
  } catch (e) {
    user = null
  }
  if (!user) {
    // apply the correct Nuxt context to our `navigateTo` call.
    return nuxtApp.runWithContext(() => navigateTo('/auth'))
  }
})
```

Usage

```
const result = nuxtApp.runWithContext(() => functionWithContext())
```

`functionWithContext`

Any function that requires the context of the current Nuxt application. This context will be correctly applied automatically.

Return value

`runWithContext` will return whatever is returned by `functionWithContext` .

A Deeper Explanation of Context

Background

Vue.js Composition API (and Nuxt composables similarly) work by depending on an implicit context. During the lifecycle, Vue sets the temporary instance of the current component (and Nuxt temporary instance of nuxtApp) to a global variable and unsets it in same tick. When rendering on the server side, there are multiple requests from different users and nuxtApp running in a same global context. Because of this, Nuxt and Vue immediately unset this global instance to avoid leaking a shared reference between two users or components.

What it does mean? The Composition API and Nuxt Composables are only available during lifecycle and in same tick before any async operation:

```
// --- Vue internal ---
const _vueInstance = null
const getCurrentInstance = () => _vueInstance
// ---

// Vue / Nuxt sets a global variable referencing to current component in _vueInstance when calling
async function setup() {
  getCurrentInstance() // Works
  await someAsyncOperation() // Vue unsets the context in same tick before async operation!
```

```
  getCurrentInstance() // null
}
```

The classic solution to this, is caching the current instance on first call to a local variable like `const instance = getCurrentInstance()` and use it in the next composable call but the issue is that any nested composable calls now needs to explicitly accept the instance as an argument and not depend on the implicit context of composition-api. This is design limitation with composables and not an issue per-se.

To overcome this limitation, Vue does some behind the scenes work when compiling our application code and restores context after each call for `<script setup>` :

```
const __instance = getCurrentInstance() // Generated by Vue compiler
getCurrentInstance() // Works!
await someAsyncOperation() // Vue unsets the context
__restoreInstance(__instance) // Generated by Vue compiler
getCurrentInstance() // Still works!
```

For a better description of what Vue actually does, see unjs/unctx#2 (comment).

Solution

This is where `runWithContext` can be used to restore context, similarly to how `<script setup>` works.

Nuxt 3 internally uses unjs/unctx to support composables similar to Vue for plugins and middleware. This enables composables like `navigateTo()` to work without directly passing `nuxtApp` to them - bringing the DX and performance benefits of Composition API to the whole Nuxt framework.

Nuxt composables have the same design as the Vue Composition API and therefore need a similar solution to magically do this transform. Check out unjs/unctx#2 (proposal), unjs/unctx#4 (transform implementation), and nuxt/framework#3884 (Integration to Nuxt).

Vue currently only supports async context restoration for `<script setup>` for async/await usage. In Nuxt 3, the transform support for `defineNuxtPlugin()` and `defineNuxtRouteMiddleware()` was added, which means when you use them Nuxt automatically transforms them with context restoration.

Remaining Issues

The `unjs/unctx` transformation to automatically restore context seems buggy with `try/catch` statements containing `await` which ultimately needs to be solved in order to remove the requirement of the workaround suggested above.

Native Async Context

Using a new experimental feature, it is possible to enable native async context support using **Node.js** `AsyncLocalStorage` and new unctx support to make async context available **natively** to **any nested async composable** without needing a transform or manual passing/calling with context.

Native async context support works currently in Bun and Node.

👉 Read more in Docs > Guide > Going Further > Experimental Features #asynccontext.

Edit on Github