🎤 **Nuxt Nation is coming** - The largest online Nuxt Conference is happening on October 18-19, 2023.

Register ↗

Data fetching

Nuxt comes with two composables and a built-in library to perform data-fetching in browser or server environments: `useFetch`, `useAsyncData` and `$fetch`. In a nutshell:

- `useFetch` is the most straightforward way to handle data fetching in a component setup function.
- `$fetch` is great to make network requests based on user interaction.
- `useAsyncData`, combined with `$fetch`, offers more fine-grained control.

Both `useFetch` and `useAsyncData` share a common set of options and patterns that we will detail in the last sections.

Before that, it's imperative to know why these composables exist in the first place.

Why using specific composables?

When using a framework like Nuxt that can perform calls and render pages on both client and server environments, some challenges must be addressed. This is why Nuxt provides composables to wrap your queries, instead of letting the developer rely on `$fetch` calls alone.

Network calls duplication

The `useFetch` and `useAsyncData` composables ensure that once an API call is made on the server, the data is properly forwarded to the client in the payload.

The payload is a JavaScript object accessible through `useNuxtApp().payload`. It is used on the client to avoid refetching the same data when the code is executed in the browser.

⚙️ Use the Nuxt DevTools to inspect this data in the payload tab.

Suspense

Nuxt uses Vue's `<Suspense>` component under the hood to prevent navigation before every async data is available to the view. The data fetching composables can help you leverage this feature and use what suits best on a per-calls basis.

👉 These composables are auto-imported and can be used in `setup` functions or lifecycle hooks

useFetch

The `useFetch` composable is the most straightforward way to perform data fetching.

app.vue

```vue
<script setup lang="ts">
const { data: count } = await useFetch('/api/count')
</script>

<template>
  Page visits: {{ count }}
</template>
```

This composable is a wrapper around the `useAsyncData` composable and `$fetch` utility.

👉 Read more in Docs > API > Composables > Use Fetch.

 Read and edit a live example in Docs > Examples > Features > Data Fetching.

$fetch

Nuxt includes the `ofetch` library, and is auto-imported as the `$fetch` alias globally across your application. It's what `useFetch` uses behind the scenes.

```
const users = await $fetch('/api/users').catch((error) => error.data)
```

Beware that using only `$fetch` will not provide network calls de-duplication and navigation prevention. It is recommended to use `$fetch` when posting data to an event handler, when doing client-side only logic, or combined with `useAsyncData`.

The `ofetch` library is built on top of the `fetch` API and adds handy features to it:

- Works the same way in browser, Node or worker environments
- Automatic response parsing
- Error handling
- Auto-retry
- Interceptors

📘 Read the full documentation of ofetch

👉 Read more in Docs > API > Utils > Dollarfetch.

useAsyncData

The `useAsyncData` composable is responsible for wrapping async logic and returning the result once it is resolved.

Indeed, `useFetch(url)` is nearly equivalent to `useAsyncData(url, () => $fetch(url))` - it's developer experience sugar for the most common use case.

There are some cases when using the `useFetch` composable is not appropriate, for example when a CMS or a third-party provide their own query layer. In this case, you can use `useAsyncData` to wrap your calls and still keep the benefits provided by the composable.

The first argument of `useAsyncData` is the unique key used to cache the response of the second argument, the querying function. This argument can be ignored by directly passing the querying function. In that case, it will be auto-generated.

```
const { data, error } = await useAsyncData('users', () => myGetFunction('users'))
```

Since the autogenerated key only takes into account the file and line where `useAsyncData` is invoked, is recommended to always create your own key to avoid unwanted behavior, if you are creating your own custom composable that is wrapping `useAsyncData` .

```
const id = ref(1)

const { data, error } = await useAsyncData(`user:${id.value}`, () => {
  return myGetFunction('users', { id: id.value })
})
```

The `useAsyncData` composable is a great way to wrap and wait for multiple `useFetch` to be done, and then retrieve the results of each.

```
const { data: discounts, pending } = await useAsyncData('cart-discount', async () => {
  const [coupons, offers] = await Promise.all([$fetch('/cart/coupons'), $fetch('/cart/offers')])

  return {
    coupons,
    offers
  }
})
```

👉 Read more in Docs > API > Composables > Use Async Data.

Options

`useAsyncData` and `useFetch` return the same object type and accept a common set of options as their last argument. They can help you control the composables behavior, such as navigation blocking, caching or execution.

Lazy

By default, data fetching composables will wait for the resolution of their asynchronous function before navigating to a new page by using Vue's Suspense. This feature can be ignored on client-side navigation with the `lazy` option. In that case, you will have to manually handle loading state using the `pending` value.

app.vue

```vue
<script setup lang="ts">
const { pending, data: posts } = useFetch('/api/posts', {
  lazy: true
})
</script>

<template>
  <!-- you will need to handle a loading state -->
  <div v-if="pending">
    Loading ...
  </div>
  <div v-else>
    <div v-for="post in posts">
      <!-- do something -->
    </div>
  </div>
</template>
```

You can alternatively use `useLazyFetch` and `useLazyAsyncData` as convenient methods to perform the same.

```ts
const { pending, data: posts } = useLazyFetch('/api/posts')
```

👉 Read more in Docs > API > Composables > Use Lazy Fetch.

👉 Read more in Docs > API > Composables > Use Lazy Async Data.

Client-only fetching

By default, data fetching composables will perform their asynchronous function on both client and server environments. Set the `server` option to `false` to only perform the call on the client-side. On initial load, the data will not be fetched before hydration is complete so you have to handle a pending state, though on subsequent client-side navigation the data will be awaited before loading the page.

Combined with the `lazy` option, this can be useful for data that is not needed on the first render (for example, non-SEO sensitive data).

```ts
/* This call is performed before hydration */
```

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

/* This call will only be performed on the client */
const { pending, data: posts } = useFetch('/api/comments', {
  lazy: true,
  server: false
})
```

The `useFetch` composable is meant to be invoked in setup method or called directly at the top level of a function in lifecycle hooks, otherwise you should use `$fetch` method.

Minimize payload size

The `pick` option helps you to minimize the payload size stored in your HTML document by only selecting the fields that you want returned from the composables.

```
<script setup lang="ts">
/* only pick the fields used in your template */
const { data: mountain } = await useFetch('/api/mountains/everest', { pick: ['title', 'description
</script>

<template>
  <h1>{{ mountain.title }}</h1>
  <p>{{ mountain.description }}</p>
</template>
```

If you need more control or map over several objects, you can use the `transform` function to alter the result of the query.

```
const { data: mountains } = await useFetch('/api/mountains', {
  transform: (mountains) => {
    return mountains.map(mountain => ({ title: mountain.title, description: mountain.description }
  }
})
```

Both `pick` and `transform` don't prevent the unwanted data from being fetched initially. But they will prevent unwanted data from being added to the payload transferred from server to client.

Caching and refetching

Keys

`useFetch` and `useAsyncData` use keys to prevent refetching the same data.

- `useFetch` uses the provided URL as a key. Alternatively, a `key` value can be provided in the `options` object passed as a last argument.
- `useAsyncData` uses its first argument as a key if it is a string. If the first argument is the handler function that performs the query, then a key that is unique to the file name and line number of the instance of `useAsyncData` will be generated for you.

> 📘 To get the cached data by key, you can use `useNuxtData`

Refresh and execute

If you want to fetch or refresh data manually, use the `execute` or `refresh` function provided by the composables.

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

<template>
  <div>
    <p>{{ data }}</p>
    <button @click="refresh">Refresh data</button>
  </div>
</template>
```

The `execute` function is an alias for `refresh` that works in exactly the same way but is more semantic for cases when the fetch is not immediate.

> 📘 To globally refetch or invalidate cached data, see `clearNuxtData` and `refreshNuxtData`.

Watch

To re-run your fetching function each time other reactive values in your application change, use the `watch` option. You can use it for one or multiple *watchable* elements.

```
const id = ref(1)

const { data, error, refresh } = await useFetch('/api/users', {
  /* Changing the id will trigger a refetch */
  watch: [id]
})
```

Not immediate

The `useFetch` composable will start fetching data the moment is invoked. You may prevent this by setting `immediate: false`, for example, to wait for user interaction.

With that, you will need both the `status` to handle the fetch lifecycle, and `execute` to start the data fetch.

```
<script setup lang="ts">
const { data, error, execute, pending, status } = await useLazyFetch('/api/comments')
</script>

<template>
  <div v-if="status === 'idle'">
    <button @click="execute">Get data</button>
  </div>

  <div v-else-if="pending">
    Loading comments...
  </div>

  <div v-else>
    {{ data }}
  </div>
</template>
```

For finer control, the `status` variable can be:

- `idle` when the fetch hasn't started
- `pending` when a fetch has started but not yet completed

- `error` when the fetch fails
- `success` when the fetch is completed successfully

Passing Headers and cookies

When we call `$fetch` in the browser, user headers like `cookie` will be directly sent to the API. But during server-side-rendering, since the `$fetch` request takes place 'internally' within the server, it doesn't include the user's browser cookies, nor does it pass on cookies from the fetch response.

Pass Client Headers to the API

We can use `useRequestHeaders` to access and proxy cookies to the API from server-side.

The example below adds the request headers to an isomorphic `$fetch` call to ensure that the API endpoint has access to the same `cookie` header originally sent by the user.

```
<script setup lang="ts">
const headers = useRequestHeaders(['cookie'])

const { data } = await useFetch('/api/me', { headers })
</script>
```

Be very careful before proxying headers to an external API and just include headers that you need. Not all headers are safe to be bypassed and might introduce unwanted behavior. Here is a list of common headers that are NOT to be proxied:

- `host` , `accept`
- `content-length` , `content-md5` , `content-type`
- `x-forwarded-host` , `x-forwarded-port` , `x-forwarded-proto`
- `cf-connecting-ip` , `cf-ray`

Pass Cookies From Server-side API Calls on SSR Response

If you want to pass on/proxy cookies in the other direction, from an internal request back to the client, you will need to handle this yourself.

composables/fetch.ts

```ts
import { appendResponseHeader, H3Event } from 'h3'

export const fetchWithCookie = async (event: H3Event, url: string) => {
  /* Get the response from the server endpoint */
  const res = await $fetch.raw(url)
  /* Get the cookies from the response */
  const cookies = (res.headers.get('set-cookie') || '').split(',')
  /* Attach each cookie to our incoming Request */
  for (const cookie of cookies) {
    appendResponseHeader(event, 'set-cookie', cookie)
  }
  /* Return the data of the response */
  return res._data
}
```

```vue
<script setup lang="ts">
// This composable will automatically pass cookies to the client
const event = useRequestEvent()

const result = await fetchWithCookie(event, '/api/with-cookie')

onMounted(() => console.log(document.cookie))
</script>
```

Options API support

Nuxt 3 provides a way to perform `asyncData` fetching within the Options API. You must wrap your component definition within `defineNuxtComponent` for this to work.

```vue
<script>
export default defineNuxtComponent({
  /* Use the fetchKey option to provide a unique key */
  fetchKey: 'hello',
  async asyncData () {
    return {
      hello: await $fetch('/api/hello')
    }
```

```
  }
})
</script>
```

Using `<script setup lang="ts">` is the recommended way of declaring Vue components in Nuxt 3.

👉 Read more in Docs > API > Utils > Define Nuxt Component.

Serialization

When fetching data from the `server` directory, the response is serialized using `JSON.stringify`. However, since serialization is limited to only JavaScript primitive types, Nuxt does its best to convert the return type of `$fetch` and `useFetch` to match the actual value.

👉 You can learn more about `JSON.stringify` limitations here.

Example

server/api/foo.ts

```ts
export default defineEventHandler(() => {
  return new Date()
})
```

app.vue

```vue
<script setup lang="ts">
// Type of `data` is inferred as string even though we returned a Date object
const { data } = await useFetch('/api/foo')
</script>
```

Custom serializer function

To customize the serialization behavior, you can define a `toJSON` function on your returned object. If you define a `toJSON` method, Nuxt will respect the return type of the function and will not try to convert the types.

server/api/bar.ts

```ts
export default defineEventHandler(() => {
  const data = {
    createdAt: new Date(),

    toJSON() {
      return {
        createdAt: {
          year: this.createdAt.getFullYear(),
          month: this.createdAt.getMonth(),
          day: this.createdAt.getDate(),
        },
      }
    },
  }
  return data
})
```

app.vue

```vue
<script setup lang="ts">
// Type of `data` is inferred as
// {
//   createdAt: {
//     year: number
//     month: number
//     day: number
//   }
// }
const { data } = await useFetch('/api/bar')
</script>
```

Using an alternative serializer

Nuxt does not currently support an alternative serializer to `JSON.stringify`. However, you can return your payload as a normal string and utilize the `toJSON` method to maintain type safety.

In the example below, we use superjson as our serializer.

server/api/superjson.ts

```ts
import superjson from 'superjson'

export default defineEventHandler(() => {
  const data = {
```

```
    createdAt: new Date(),

    // Workaround the type conversion
    toJSON() {
      return this
    }
  }

  // Serialize the output to string, using superjson
  return superjson.stringify(data) as unknown as typeof data
})
```

app.vue

```
<script setup lang="ts">
import superjson from 'superjson'

// `date` is inferred as { createdAt: Date } and you can safely use the Date object methods
const { data } = await useFetch('/api/superjson', {
  transform: (value) => {
    return superjson.parse(value as unknown as string)
  },
})
</script>
```

Edit on Github

© 2016-2023 Nuxt - MIT License