Server Directory

Nuxt automatically scans files inside these directories to register API and server handlers with HMR support:

- `~/server/api`
- `~/server/routes`
- `~/server/middleware`

Each file should export a default function defined with `defineEventHandler()` or `eventHandler()` (alias).

The handler can directly return JSON data, a `Promise`, or use `event.node.res.end()` to send a response.

Example: Create the `/api/hello` route with `server/api/hello.ts` file:

server/api/hello.ts

```ts
export default defineEventHandler((event) => {
  return {
    hello: 'world'
  }
})
```

You can now universally call this API in your pages and components:

pages/index.vue

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

<template>
  <pre>{{ data }}</pre>
</template>
```

Note that h3 utilities are auto-imported.

👉 Read more in Nitro Route Handling Docs.

Server Routes

Files inside the `~/server/api` are automatically prefixed with `/api` in their route.

To add server routes without `/api` prefix, put them into `~/server/routes` directory.

Example:

server/routes/hello.ts

```
export default defineEventHandler(() => 'Hello World!')
```

Given the example above, the `/hello` route will be accessible at http://localhost:3000/hello.

💡 Note that currently server routes do not support the full functionality of dynamic routes as pages do.

Server Middleware

Nuxt will automatically read in any file in the `~/server/middleware` to create server middleware for your project.

Middleware handlers will run on every request before any other server route to add or check headers, log requests, or extend the event's request object.

Middleware handlers should not return anything (nor close or respond to the request) and only inspect or extend the request context or throw an error.

Examples:

server/middleware/log.ts

```
export default defineEventHandler((event) => {
  console.log('New request: ' + getRequestURL(event))
})
```

server/middleware/auth.ts

```
export default defineEventHandler((event) => {
```

```
  event.context.auth = { user: 123 }
})
```

Server Plugins

Nuxt will automatically read any files in the `~/server/plugins` directory and register them as Nitro plugins. This allows extending Nitro's runtime behavior and hooking into lifecycle events.

Example:

server/plugins/nitroPlugin.ts

```ts
export default defineNitroPlugin((nitroApp) => {
  console.log('Nitro plugin', nitroApp)
})
```

👉 Read more in Nitro Plugins.

Server Utilities

Server routes are powered by unjs/h3 which comes with a handy set of helpers.

👉 Read more in Available H3 Request Helpers.

You can add more helpers yourself inside the `~/server/utils` directory.

For example, you can define a custom handler utility that wraps the original handler and performs additional operations before returning the final response.

Example:

server/utils/handler.ts

```ts
import type { EventHandler, EventHandlerRequest } from 'h3'

export const defineWrappedResponseHandler = <T extends EventHandlerRequest, D> (
  handler: EventHandler<T, D>
): EventHandler<T, D> =>
  defineEventHandler<T>(async event => {
    try {
```

```
    // do something before the route handler
    const response = await handler(event)
    // do something after the route handler
    return { response }
  } catch (err) {
    // Error handling
    return { err }
  }
})
```

Server Types

This feature is available from Nuxt >= 3.5

To improve clarity within your IDE between the auto-imports from 'nitro' and 'vue', you can add a `~/server/tsconfig.json` with the following content:

server/tsconfig.json

```
{
  "extends": "../.nuxt/tsconfig.server.json"
}
```

Although right now these values won't be respected when type checking (`nuxi typecheck`), you should get better type hints in your IDE.

Usage Examples

Matching Route Parameters

Server routes can use dynamic parameters within brackets in the file name like `/api/hello/[name].ts` and be accessed via `event.context.params` .

Example:

server/api/hello/[name].ts

```
export default defineEventHandler((event) => {
  const name = getRouterParam(event, 'name')
```

```
  return `Hello, ${name}!`
})
```

You can now universally call this API using `await $fetch('/api/hello/nuxt')` and get `Hello, nuxt!`.

Matching HTTP Method

Handle file names can be suffixed with `.get`, `.post`, `.put`, `.delete`, ... to match request's HTTP Method.

server/api/test.get.ts

```
export default defineEventHandler(() => 'Test get handler')
```

server/api/test.post.ts

```
export default defineEventHandler(() => 'Test post handler')
```

Given the example above, fetching `/test` with:

- **GET** method: Returns `Test get handler`
- **POST** method: Returns `Test post handler`
- Any other method: Returns 405 error

You can also use `index.[method].ts` inside a directory for structuring your code differently.

Example:

server/api/foo/index.ts

```
export default defineEventHandler((event) => {
  // handle the `api/foo` endpoint
})
```

This is useful to create API namespaces.

Examples:

server/api/foo/index.get.ts

```
export default defineEventHandler((event) => {
  // handle GET requests for the `api/foo` endpoint
})
```

server/api/foo/index.post.ts

```
export default defineEventHandler((event) => {
```

```
  // handle POST requests for the `api/foo` endpoint
})
```

server/api/foo/bar.get.ts

```ts
export default defineEventHandler((event) => {
  // handle GET requests for the `api/foo/bar` endpoint
})
```

Catch-all Route

Catch-all routes are helpful for fallback route handling. For example, creating a file named `~/server/api/foo/[...].ts` will register a catch-all route for all requests that do not match any route handler, such as `/api/foo/bar/baz` .

Examples:

server/api/foo/[...].ts

```ts
export default defineEventHandler(() => `Default foo handler`)
```

server/api/[...].ts

```ts
export default defineEventHandler(() => `Default api handler`)
```

Handling Requests with Body

server/api/submit.post.ts

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

You can now universally call this API using `$fetch('/api/submit', { method: 'post', body: { test: 123 } })` .

We are using `submit.post.ts` in the filename only to match requests with `POST` method that can accept the request body. When using `readBody` within a GET request, `readBody` will throw a `405 Method Not Allowed` HTTP error.

Handling Requests With Query Parameters

Sample query `/api/query?param1=a&param2=b`

server/api/query.get.ts

```ts
export default defineEventHandler((event) => {
  const query = getQuery(event)
  return { a: query.param1, b: query.param2 }
})
```

Error handling

If no errors are thrown, a status code of `200 OK` will be returned. Any uncaught errors will return a `500 Internal Server Error` HTTP Error.

To return other error codes, throw an exception with `createError`

server/api/validation/[id].ts

```ts
export default defineEventHandler((event) => {
  const id = parseInt(event.context.params.id) as number
  if (!Number.isInteger(id)) {
    throw createError({
      statusCode: 400,
      statusMessage: 'ID should be an integer',
    })
  }
  return 'All good'
})
```

Returning other status codes

To return other status codes, you can use the `setResponseStatus` utility.

For example, to return `202 Accepted`

server/api/validation/[id].ts

```ts
export default defineEventHandler((event) => {
  setResponseStatus(event, 202)
})
```

Accessing Runtime Config

server/api/foo.ts

```ts
export default defineEventHandler((event) => {
  const config = useRuntimeConfig()
  return { key: config.KEY }
})
```

Accessing Request Cookies

```ts
export default defineEventHandler((event) => {
  const cookies = parseCookies(event)
  return { cookies }
})
```

Advanced Usage Examples

Nitro Configuration

You can use `nitro` key in `nuxt.config` to directly set Nitro configuration.

This is an advanced option. Custom config can affect production deployments, as the configuration interface might change over time when Nitro is upgraded in semver-minor versions of Nuxt.

nuxt.config.ts

```ts
export default defineNuxtConfig({
  // https://nitro.unjs.io/config
  nitro: {}
})
```

Using a Nested Router

server/api/hello/[...slug].ts

```ts
import { createRouter, defineEventHandler, useBase } from 'h3'

const router = createRouter()

router.get('/test', defineEventHandler(() => 'Hello World'))

export default useBase('/api/hello', router.handler)
```

Sending Streams (Experimental)

Note: This is an experimental feature and is only available within Node.js environments.

server/api/foo.get.ts

```ts
import fs from 'node:fs'
import { sendStream } from 'h3'

export default defineEventHandler((event) => {
  return sendStream(event, fs.createReadStream('/path/to/file'))
})
```

Sending Redirect

server/api/foo.get.ts

```ts
export default defineEventHandler(async (event) => {
  await sendRedirect(event, '/path/redirect/to', 302)
})
```

Return a Legacy Handler or Middleware

server/api/legacy.ts

```ts
export default fromNodeMiddleware((req, res) => {
  res.end('Legacy handler')
})
```

Legacy support is possible using unjs/h3, but it is advised to avoid legacy handlers as much as you can.

server/middleware/legacy.ts

```ts
export default fromNodeMiddleware((req, res, next) => {
  console.log('Legacy middleware')
  next()
})
```

Never combine `next()` callback with a legacy middleware that is `async` or returns a `Promise` !

Server Storage

Nitro provides a cross-platform storage layer. In order to configure additional storage mount points, you can use `nitro.storage` , or server plugins.

Example: Using Redis

Using `nitro.storage` :

nuxt.config.ts

```ts
export default defineNuxtConfig({
  nitro: {
    storage: {
      'redis': {
        driver: 'redis',
        /* redis connector options */
        port: 6379, // Redis port
        host: "127.0.0.1", // Redis host
        username: "", // needs Redis >= 6
        password: "",
        db: 0, // Defaults to 0
        tls: {} // tls/ssl
      }
    }
  }
})
```

Alternatively, using server plugins:

server/plugins/storage.ts nuxt.config.ts

server/plugins/storage.ts

```ts
import redisDriver from 'unstorage/drivers/redis'

export default defineNitroPlugin(() => {
  const storage = useStorage()

  // Dynamically pass in credentials from runtime configuration, or other sources
  const driver = redisDriver({
      base: 'redis',
      host: useRuntimeConfig().redis.host,
      port: useRuntimeConfig().redis.port,
      /* other redis connector options */
    })

  // Mount driver
  storage.mount('redis', driver)
})
```

Create a new file in `server/api/test.post.ts` :

server/api/test.post.ts

```ts
export default defineEventHandler(async (event) => {
  const body = await readBody(event)
  await useStorage().setItem('redis:test', body)
  return 'Data is set'
})
```

Create a new file in `server/api/test.get.ts` :

server/api/test.get.ts

```ts
export default defineEventHandler(async (event) => {
  const data = await useStorage().getItem('redis:test')
  return data
})
```

Create a new file in `app.vue` :

app.vue

```vue
<script setup lang="ts">
  const { data: resDataSuccess } = await useFetch('/api/test', {
      method: 'post',
      body: { text: 'Nuxt is Awesome!' }
  })
  const { data: resData } = await useFetch('/api/test')
</script>
```

```vue
<template>
  <div>
    <div>Post state: {{ resDataSuccess }}</div>
    <div>Get Data: {{ resData.text }}</div>
  </div>
</template>
```

👉 Read more in Docs > Guide > Directory Structure > Server.

Edit on Github