



Jonas Galvez

Happy Little Monoliths

Build and deploy **full stack**
Node.js applications
with **Fastify** and **Vite**.

1st edition

April 18, 2025

All future editions are included.

Examples and code snippets assume Unix-like environment.

Software and library versions used:

- Node v22
- Fastfiy v5
- Vite v6
- Vue v3
- React v19

Chapter 3

A Fastify and Vite Crash Course

Now that we've covered the state of modern JavaScript, it's time to dive into Fastify and Vite. We'll look first at Fastify, the most mature and robust web server for Node.js. A strong opinion, one would say. Hopefully after the following section you'll understand why.

Essential Fastify

As covered before, it's 2025 and the JavaScript runtime ecosystem has evolved substantially. Deno, Bun, Workerd — and many others in development — can make one question if Node.js is still the right choice. I, for one, am extremely eager to try new stuff, and have been an early adopter on more than one occasion. In this trade, things move fast and you have to adapt. If you ignore new things, chances are, your skills quickly become obsolete.

Hono is an example of new stuff I was eager to try. And I *love it*. Enough to start recommending it instead of Fastify? No — or perhaps, not yet. It's not about what I love the most, but what makes more *practical sense*.

As someone with a strong Python background from an earlier period in time, I can't help but see Fastify as an analog to **Flask** — something small, yet incredibly fast and reliable.

And still can't help but see Node.js, in practical senses, as the most stable and robust runtime available today, with the

biggest community, support and general resources available. Performance nowadays might lag behind Bun, for instance, for some applications, but the feeling that something has stood the test of time ends up weighing in the decision.

And the main reason I started to trust Fastify to begin with was the fact I realized it's written and maintained by folks who actively contribute to Node.js itself. The style of Fastify's code follows the style of Node.js itself, with the utmost care and precision. Fastify was built with performance in mind, and it deserves its name.

This section is intended to fast track you through all of Fastify's essentials, so rather than diving into **deeper details**, you'll be presented with a series of snippets illustrating its essential API features and characteristics, as well as best practices.

Before starting, install these packages:

```
% pnpm add fastify
% pnpm add fastify-plugin
% pnpm add @fastify/one-line-logger
```

sh

The basic setup code for a Fastify server is as follows:

```
import Fastify from 'fastify'

const server = Fastify()

server.get('/', (req, reply) => {
  reply.send('Hello world!')
})

await server.listen({ port: 8000 })
```

js

Defining routes is very straightforward in Fastify.

Like `server.get()`, all other HTTP methods are accounted for. If the request takes `PUT` or `POST` data, it becomes available as `req.body`.

If you post JSON with the correct headers, that is, `Content-Type` set to `application/json`, `req.body` will have the parsed JSON.

If you need to parse `x-www-form-urlencoded` bodies, you can do so by employing the `@fastify/formbody` plugin. If you need multipart bodies, use the `@fastify/multipart` package. If you need something, chances are Fastify has a plugin for it.

In practice though, you'd want to write:

js

```
import Fastify from 'fastify'

export async function getServer() {
  const server = Fastify()

  server.get('/', (req, reply) => {
    reply.send('Hello world!')
  })

  await server.ready()

  return server
}

if (process.argv[1] === import.meta.filename) {
  const server = await getServer()
  await server.listen({ port: 8000 })
}
```

In one file, you're able to either start the server, or export it for consumption by a test. Fastify is amazing for testing.

Notice in this scenario we explicitly await on `ready()`, to ensure all plugins have been loaded before we attempt injecting

mock requests. The [API used for mocking requests](#) is quite robust, and has been used to [automatically generate an internal API client based on Fastify routes](#).

With the `getServer()` function, we can write:

js

```
import { test } from 'node:test'
import { strictEqual } from 'node:assert'
import { getServer } from './basic-server.js'

test('Renders hello world', async () => {
  const server = await getServer()

  const response = await server.inject({
    method: 'GET',
    url: '/'
  })

  strictEqual(response.statusCode, 200)
  strictEqual(response.body, 'Hello world!')

  await server.close()
})
```

Going further, you'd probably define your routes as a Fastify plugin. Fastify plugins are functions that take up to three parameters. In its synchronous definition, the third parameter is a function that tells Fastify's bootstrap code that the plugin is done executing. Most commonly though, authors opt to use the asynchronous definition, which eliminates the third parameter altogether.

Create a `routes.js` file with the plugin code:

js

```
export default async function (fastify, options) {
  fastify.get('/', (_, reply) => {
```

```
    reply.send('Hello world!')
  })
  if (options.health) {
    fastify.get('/health', (_, reply) => {
      reply.send('ok')
    })
  }
}
```

And then modify `getServer()` to import it as follows:

```
import Fastify from 'fastify'
import routes from './routes.js'

export async function getServer() {
  const server = Fastify()

  await server.register(routes, {
    health: true,
  })

  await server.ready()

  return server
}
```

js

In Fastify, plugins create their own **encapsulation context** — that means anything you do in the scope of a plugin is valid for that plugin only, and completely unseen by all others.

Unless of course **you want** a plugin to be registered at the parent scope, in which case you'd use **fastify-plugin**, which basically annotates the plugin function with a symbol telling Fastify to skip creating a new encapsulation context for it.

```
import Fastify from 'fastify'
import fp from 'fastify-plugin'
import routes from './routes.js'

export async function getServer() {
  const server = Fastify()

  await server.register(fp(routes), {
    health: true,
  })
  await server.ready()

  return server
}
```

Fastify also has a peculiar method for *extending* the server, adding things to the Fastify server itself, or to Fastify's **Request** and **Reply** objects. It does so in a way that prevents messing up with V8's **inline caches**, which basically means making sure objects have a pre-defined shape (set of properties) at boot time and stay in that configuration at runtime.

This is a good moment to mention that Fastify wraps Node's **IncomingMessage** and **ServerResponse** instances in its **Request** and **Reply** abstractions, respectively — the **req** and **reply** parameters you see on route definitions.

In these wrappers, you still have access to the underlying native objects representing requests and responses via **req.raw** and **res.raw**, though there are only a handful of circumstances where it would make sense to touch them directly.

So to add something to the Fastify server instance, you do:

```
server.decorate('helper', function () {
  // this will have the Fastify server instance context
})
```


And then `server.helper()` becomes *safely* available, and by *safely* I mean in a way that doesn't cause V8 to create more hidden classes than it needs — as mentioned before.

Here's a more elaborate example:

js

```
let url
server.decorate('serverURL', { getter: () => url })
server.addHook('onListen', () => {
  const { port, address, family } = server.server.address()
  const protocol = server.https ? 'https' : 'http'
  if (family === 'IPv6') {
    url = `${protocol}://[${address}]:${port}`
  } else {
    url = `${protocol}://${address}:${port}`
  }
})
```

A lot to digest here: first, the `decorate()` call adds a `serverURL` **getter** to the Fastify server instance. Next, we register an `onListen` **hook** to set `url` after the server boots up and we can then call Fastify's underlying server's `address()` method.

Say you wanted to also add `req.fullURL` to join `server.serverURL` with the current request path. Here's how:

js

```
let url

server.decorate('serverURL', { getter: () => url })
server.decorateRequest('fullURL', null)

server.addHook('onRequest', (req, reply, done) => {
  req.fullURL = `${server.serverURL}${req.url}`
  done()
})
```

```

server.addHook('onListen', (done) => {
  const { port, address, family } = server.server.address()
  const protocol = server.https ? 'https' : 'http'
  if (family === 'IPv6') {
    url = `${protocol}://[${address}]:${port}`
  } else {
    url = `${protocol}://${address}:${port}`
  }
  done()
})

```

To extend Fastify's `Request` object, use `decorateRequest()`. For `Reply`, unsurprisingly, `decorateReply()`. Above you can also see an example of using Fastify's request-level hooks — in this case, `onRequest`, used to attach a value to the previously predefined `fullURL` request property. Notice how the decoration needs to happen before it even gets a value, just so V8 doesn't create an additional hidden class when it does get assigned.

That's why these decoration methods exist, essentially — they're not trying to make things more complicated, they're hiding away important optimizations required to completely eliminate, or at least reduce, any performance overhead your application might have.

There's a bit more to Fastify, some of which you'll still be presented with in later chapters, but one could also say this is the gist of it.

There's one thing we still need to cover here though:

Fastify's [server instance options](#).

This is what a real world app's server options may look like:

```

const server = Fastify({
  connectionTimeout: 45 * 1000,

```

js

```
logger: true,  
...process.stdout.isTTY && {  
  logger: {  
    transport: {  
      // Formatted logging for dev  
      target: '@fastify/one-line-logger',  
    }  
  }  
},  
bodyLimit: 100 * 1024 * 1024,  
})
```

In this snippet, we set a custom connection timeout (the default is zero), we condition the enabling of a simplified logger instance on the presence of a terminal and finally, we set the HTTP request body limit to **100mb**. Check out the [full reference](#).

This example is available in [examples/3/fastify](#).

To recap, you have seen how to:

- Create a Fastify server instance and start it
- Add routes, hooks, decorators and plugins to it
- Create a testable Fastify server factory
- Create a Fastify test using Node.js' test runner

With this, you're all set to move forward.

Essential Vite

Vite is one of the most *transformative* build tools to emerge in the JavaScript ecosystem in recent years, no pun intended.

Created by **Evan You** (the author of Vue), Vite has established itself as a core foundation for web development, with over **26 million** weekly downloads.

Vite's key characteristic is that, unlike others bundlers like Webpack, it leverages browsers' support for ES modules to deliver client code. There's no need to package everything in a single bundle beforehand, files are built and served isolated, both in development and production modes.

Vite has a powerful plugin system, based on Rollup's plugin interface with some custom extensions.

The main entry point of a Vite application is the `index.html` file. The mere existence of this file is enough for Vite, no configuration needed (yet). Let's give it a try:

```
% mkdir basic-vite
% pnpm add vite -D
```

sh

Now create `index.html`:

```
<!doctype html>
<h1></h1>
<script type="module" src="/index.js">
</script>
```

html

Did you know `<html>`, `<head>` and `<body>` are optional tags [as per the HTML5 spec?](#)

And `index.js`:

```
document.querySelector('h1').innerText = 'Hello world!'
```

js

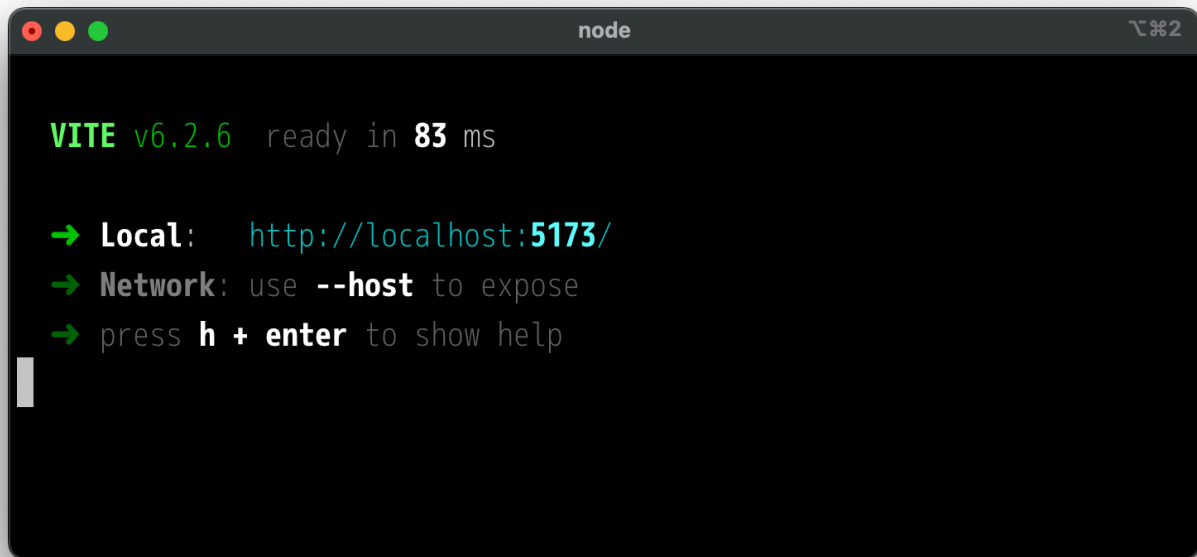
And then run Vite's development server:

```
npx vite dev
```

sh

A few things to note: as mentioned before, even without a configuration file, Vite is smart enough to build the module

loaded by `index.html`. With Vite's development server, you can do changes to `index.js` and see them appear in the browser nearly in real time.



```
node
VITE v6.2.6 ready in 83 ms
→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

It will only consider scripts that have `type` set to `module`.

You can have as many as you want — and that's why it's convenient to have `index.html` as the front and central entry point of your client application. If you include a non-module `<script>`:

```
<!doctype html>
<h1></h1>
<script type="module" src="/index.js">
</script>
<script src="/static.js">
</script>
```

html

The `build` command will give you a warning:

```
% npx vite build
vite v6.2.6 building for production...
```

sh


```
<script src="/static.js"> in "/index.html" can't be bundled without type="modu
✓ 3 modules transformed.
dist/index.html          0.12 kB | gzip: 0.12 kB
dist/assets/index-DCw9hIbG.js 0.76 kB | gzip: 0.43 kB
✓ built in 91ms
```

But if you create a **public/** folder and move **static.js** to it, it'll automatically be added to your bundle.

It's time to start playing with Vite's configuration options.

Create a **vite.config.js** file as follows:

```
import { defineConfig } from 'vite'

export default defineConfig({
  root: import.meta.dirname,
  base: '/custom',
  build: {
    outDir: 'build',
    assetsDir: 'app',
    minify: false,
  }
})
```

js

When you run **npx vite build**, you'll notice a couple of things. First, the generated bundle is now located at **./build**, and the assets dir is now **./build/app**. Open **./build/index.html** and you should see:

```
<!doctype html>
<script type="module" crossorigin src="/custom/app/index-DSadDZK9.js">
</script>
<h1></h1>
```

html

```
<script src="/custom/static.js">
</script>
```

Notice how we also explicitly set the Vite project's root to `import.meta.dirname`, and disable minification with `build.minify`.

This example is available in `examples/3/vite`.

*
**

You're now armed with enough Fastify and Vite skills to proceed.

Before we dive into `@fastify/vite`, let's explore what directly integrating Fastify and Vite looks like.

Direct Integration

To better understand what's involved in integrating Fastify and Vite, let's manually create an SSR server for a Vue app.

First install all required dependencies:

```
% pnpm add @fastify/middie
% pnpm add @fastify/one-line-logger
% pnpm add @fastify/static
% pnpm add fastify
% pnpm add vite
% pnpm add @vitejs/plugin-vue -D
```

sh

Now let's create the client code — and place it under a `client/` folder while we're at it, for better organization:

In `client/index.html`:

html

```
<!doctype html>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<div id="root"><!-- element --></div>
<script type="module" src="/mount.js"></script>
```

In **client/mount.js**:

js

```
import { createApp } from './base.js'

createApp().mount('#root')
```

In **client/base.vue**:

html

```
<template>
  <p>Hello world from Vue, Fastify and Vite!</p>
</template>
```

In **client/base.js**:

js

```
import { createSSRApp } from 'vue'
import base from './base.vue'

export function createApp () {
  return createSSRApp(base)
}
```

And finally, in **client/server.js**:

js

```
export { createApp } from './base.js'
```

This is the file Fastify will load for SSR.

What we want to happen is: when the user requests `/`, the server executes `createApp()` and places the output inside the HTML file, replacing the `<!-- element -->` placeholder.

First, let's also add `vite.config.js`:

js

```
import { join } from 'node:path'
import { defineConfig } from 'vite'
import vuePlugin from '@vitejs/plugin-vue'

export default defineConfig({
  root: join(import.meta.dirname, 'client'),
  plugins: [vuePlugin()],
  environments: {
    ssr: {
      build: {
        outDir: `dist/server`,
        ssr: true,
        rollupOptions: {
          input: {
            index: join(import.meta.dirname, 'client/server.js'),
          },
        },
      },
    },
  },
  builder: {
    async buildApp (builder) {
      await builder.build(builder.environments.client)
      await builder.build(builder.environments.ssr)
    }
  }
})
```

Quite a bit to unpack here. First thing to notice is the use of the Vue plugin, which allows Vite to compile `vue` files.

Then, we have an SSR environment definition, so we can have a separate build for the `client/server.js` file.

Vite 6 can have multiple build environments — with `client` and `ssr` being the most commonly used ones. The `client` environment is always implicitly available; that's what takes care of bundling all the code for the `index.html` file.

And finally, via `builder.buildApp()`, we're able to tell Vite to always build these two environments when `vite build` runs.

We can now write our Fastify server. First let's take care of serving pages in development mode, leveraging Vite's development server **from within** Fastify. Let's begin with the imports:

```
import { join } from 'node:path'
import Fastify from 'fastify'
import Middie from '@fastify/middie'
import { resolveConfig, createServer, createServerModuleRunner } from 'vite'
import { renderToString } from 'vue/server-renderer'
```

js

Next, let's write the basic boilerplate — where we set up the Fastify instance, and also make sure we have access to Vite's configuration file. For that we can use Vite's `resolveConfig()` function, which takes a **Vite inline configuration** object.

With it, we can determine the location of the configuration file we want to load. The function will assume the current working directory if you omit `configFile` in the inline configuration options.

```
export async function getServer (devFlag) {
  const dev = devFlag ?? process.argv.includes('--dev')
```

js


```

const server = Fastify({
  logger: true,
  ...(process.stdout.isTTY || devFlag) && {
    logger: {
      transport: {
        target: '@fastify/one-line-logger'
      }
    }
  },
})

const viteConfig = await resolveConfig({
  configFile: join(import.meta.dirname, 'vite.config.js')
})

const { root, base: baseOriginal, build } = viteConfig

// Ensure viteConfig.base has no trailing slashes if defined
const base = (baseOriginal ?? '/').replace(/(?<=\w+)\$/ , '')

// Route setup

return server
}

if (process.argv[1] === import.meta.filename) {
  const server = await getServer()
  await server.listen({ port: 3000 })
}

```

Now that we have the basic boilerplate for delivering `index.html` with our server-side rendered Vue component in development mode, we can add routing. First we'll need `Middie`, a Fastify plugin to provide support for Express-style middleware functions — which Vite uses.

```
await server.register(Middie)
```

Next, Vite's development server options, the Vite development server instance, and a Vite *runner* for the environment we need (in this case, SSR). What's neat about runners is that they provide an `import()` method that picks up any changes you make to your files while the server is running — of course you have to run it again.

```
const devServerOptions = {
  server: {
    middlewareMode: true,
    hmr: {
      server: server.server,
    },
  },
  appType: 'custom',
}

const devServer = await createServer(devServerOptions)
const devRunner = createServerModuleRunner(devServer.environments.ssr)

server.use(devServer.middlewares)
```

Next, the route registration.

```
server.get(base, async (req, reply) => {
  const indexPath = join(viteConfig.root, 'index.html')
  const indexHtml = await readFile(indexPath, 'utf8')
  const transformedHtml = await devServer.transformIndexHtml(
    req.url,
    indexHtml,
  )
  const { createApp } = await devRunner.import('/server.js')
```

```
const element = await renderToString(createApp())
const html = transformedHtml.replace('<!-- element -->', element)
reply.type('text/html')
reply.send(html)
})
```

Since this is development mode, we:

- Load the source `index.html` file every time;
- Run it through Vite's `transformIndexHtml()` function;
- And also load the `client/server.js` file every time.

And finally, we ensure Vite's dev server closes with Fastify.

```
server.addHook('onClose', () => devServer.close())
```

js

To test it, you can run `node server.js --dev`.

And you'll also notice it will only work with the `--dev` flag for the moment. To **enable production mode**, we have to make some additions. First, bypass registering `Middie` and register `@fastify/static` static delivery routes instead:

```
if (dev) {
  await server.register(Middie)
} else {
  await server.register(async function assetFiles (scope) {
    await scope.register(FastifyStatic, {
      root,
      prefix: join(base || '/', build.assetsDir)
    })
  })
}

await server.register(async function publicFiles(scope) {
  await scope.register(FastifyStatic, {
    root,
```

js

```

    index: false,
    wildcard: false
  })
})
}

```

The reason we scope the `@fastify/static` module registrations in their own plugins is to avoid having conflicting options.

Next, let's register a different route handler for production:

```

if (dev) {
  // Omitted for brevity
} else {
  const bundlePath = join(viteConfig.root, viteConfig.outDir ?? 'dist')
  const indexHtml = readFileSync(join(bundlePath, 'index.html'), 'utf8')
  const { createApp } = await import(join(bundlePath, 'server', 'index.js'))

  server.get(base, async (_, reply) => {
    const element = await renderToString(createApp())
    const html = indexHtml.replace('<!-- element -->', element)
    reply.type('text/html')
    reply.send(html)
  })
}

```

In the end, this is the directory layout we have:

```

direct/
├─ package.json
├─ server.js
├─ client/
│   ├─ base.vue
│   ├─ base.js
│   └─ index.html

```

```
| ┌ mount.js
| └ vite.config.js
```

This example is available in [examples/3/fastify-vite-direct](#).

You might think — ***This is great. This is simple, I can understand all moving parts, I don't need no frameworks!***

However, there are a few problems:

- Vite's package needs to be available in production, even though it's not part of the bundle delivered to the client (obviously), it's still one extra dependency we have to load just to be able to load Vite's configuration file.
- If you try to run the server in production mode without building the app first (running `vite build`) you'll get an error, this scenario is completely unhandled.
- You have to manually register development and production routes for each of the client routes that you have. In this simple root page example, it's definitely ok, but for multiple routes you'll want to modularize this process.
- If you want to have your SSR module be responsible for setting up other aspects of the route handler, like additional `<head>` tags, you have to manually set it up for every route.

That's where we arrive at [@fastify/vite](#).

Essential [@fastify/vite](#)

[@fastify/vite](#) is the official Fastify plugin for Vite integration. It takes care of much of what has been shown here so far. In addition to eliminating the need for all the boilerplate you've seen, it also has some other advantages:

- Vite's package **does not need** to be available in production, **@fastify/vite** will cache all relevant config options it needs as part of the bundle.
- It provides friendlier messages for cases where, for instance, you're attempting to launch a server in production mode and have forgotten to build the client first.
- It takes care of registering routes automatically, and also allows you to cleanly define which *client* routes need to be registered at the server-level for SSR.
- It loads your Vite configuration file automatically, no matter the format it's in — **@fastify/vite** will recognize **js**, **mjs** and **ts** extensions. Needless to say, your TypeScript code is also automatically supported by Vite. Not the Fastify server code, but all the code under Vite's project root.

Let's move our direct integration example to **@fastify/vite**.

First, update **package.json**:

js

```
{
  "type": "module",
  "scripts": {
    "dev": "node server.js --dev",
    "start": "node server.js",
    "build": "vite build"
  },
  "dependencies": {
    "@fastify/vite": "^8.0.3",
    "fastify": "^5.3.0"
  },
  "devDependencies": {
    "vite": "^6.2.6",
    "@vitejs/plugin-vue": "^5.2.3"
  }
}
```

```
}  
}
```

`@fastify/vite` includes `@fastify/middie` and `@fastify/static`, so they can be removed. We can also move `vite` to `devDependencies` now.

Next, update `vite.config.js`:

```
import { join } from 'node:path'  
import { defineConfig } from 'vite'  
import vuePlugin from '@vitejs/plugin-vue'  
import fastifyVitePlugin from '@fastify/vite/plugin'  
  
export default defineConfig({  
  root: join(import.meta.dirname, 'client'),  
  plugins: [fastifyVitePlugin(), vuePlugin()],  
})
```

js

Next, rename `client/server.js` to `client/index.js`. `@fastify/vite` automatically recognizes the `index.js` file in your Vite project's root to be the SSR module. In earlier versions of `@fastify/vite`, it was known as the ***client module***. In the latest release of `@fastify/vite`, the client module is now created from all Vite environment entry points (instead of being assumed to be the SSR environment's entry point), which are automatically loaded and made available in the first parameter passed to the `prepareClient()` hook. See its default definition below:

```
async function prepareClient(entries, scope, config) {  
  const clientModule = entries.ssr  
  if (!clientModule) {  
    return null  
  }  
  const routes =
```

js

```
    typeof clientModule.routes === 'function'
      ? await clientModule.routes()
      : clientModule.routes
  return Object.assign({}, clientModule, { routes })
}
```

@fastify/vite/plugin also takes care of setting up your build environments properly, and also **builder.buildApp()**.

Next, update **server.js** as follows:

```
import Fastify from 'fastify'
import FastifyVite from '@fastify/vite'
import { renderToString } from 'vue/server-renderer'

export async function getServer (dev) {
  const server = Fastify()

  await server.register(FastifyVite, {
    root: import.meta.url,
    dev: dev ?? process.argv.includes('--dev'),
    async createRenderFunction ({ createApp }) {
      return async () => ({
        element: await renderToString(createApp())
      })
    }
  })

  server.get('/', (_, reply) => {
    return reply.html()
  })

  await server.vite.ready()
}
```

js

```
    return server
  }
```

First things to notice:

```
server.get('/', (_, reply) => {
  return reply.html()
})

await server.vite.ready()
```

js

We must always await on `server.vite.ready()`, no matter if in production or development mode, to make sure all client modules are loaded. And for this example, we have to manually register a route handler and use `reply.html()` to send the HTML page.

`@fastify/vite` works by breaking down the entire process we've seen in the direct integration example into a series of **well defined steps**, which are defined as hook functions that can be overridden:

```
└─ prepareClient()
  └─ createHtmlTemplateFunction()
    └─ createHtmlFunction()
      └─ createRenderFunction()
        └─ createRouteHandler()
          └─ createErrorHandler()
            └─ createRoute()
```

Here's a rundown of what each of them do:

- `prepareClient(entries, server, config)`

As soon as all Vite build environments are loaded, they are

passed to the `prepareClient()` hook, the return of which will be considered the `client` module.

- `createHtmlTemplateFunction(source)`
This hook creates a templating function based on HTML input — it's used to turn `index.html` into a templating function. Every segment like `<!-- element -->` is transformed into a string template literal.
- `createHtmlFunction(source, server, config)`
This hook creates the `reply.html()` method based on the function created by the `createHtmlTemplateFunction()` hook.
- `createRenderFunction(client, server, config)`
This hook creates a `render()` method that is *decorated* into all Fastify `Reply` objects, i.e., makes `reply.render()` available.
- `createRouteHandler({ client, route }, server, config)`
This hook creates the default route handler for registering Fastify routes based on the client module's `routes` array.

More on this in a bit!

- `createErrorHandler({ client, route }, server, config)`
This hook creates the default error handler for all the Fastify routes registered based on the client module's `routes` array.
- `createRoute({ handler, errorHandler, route }, server, config)`
This hook is responsible for actually registering an individual Fastify route for each of your client-level routes, defined by a `routes` array export.

For this integration, we just need `createRenderFunction()` — notice how the client module, *prepared* from your specified Vite build environments via `prepareClient()`, is passed as the first parameter giving you access to `createApp`:

```

async createRenderFunction ({ createApp }) {
  return async () => ({
    element: await renderToString(createApp())
  })
}

```

That's because `@fastify/vite`'s default definition for `createHtmlFunction()` is as follows:

```

async function createHtmlFunction(source, scope, config) {
  const indexHtmlTemplate = await config.createHtmlTemplateFunction(source)
  if (config.spa) {
    return function () {
      this.type('text/html')
      this.send(indexHtmlTemplate({ element: '' }))
      return this
    }
  }
  if (config.hasRenderFunction) {
    return async function (ctx) {
      this.type('text/html')
      this.send(await indexHtmlTemplate(ctx ?? (await this.render(ctx))))
      return this
    }
  }
  return async function (ctx) {
    this.type('text/html')
    this.send(await indexHtmlTemplate(ctx))
    return this
  }
}

```

Although you can arrange it any way you like, the default behavior is to pass the result of the `reply.render()` function to

the `reply.html()` function, which is what can be seen happening above.

In production, both `reply.render()` and `reply.html()` stay working as expected, but instead of relying on Vite's development server, they use the production bundle.

This example is available in [examples/3/fastify-vite](#).

Universal Routing

`@fastify/vite` also makes it easy to register multiple routes for SSR based on your client code. In fact, the same route configuration used for application client-side (for SPA navigation) is reused by Fastify. Let's try it out with a new project.

First, create a few view files:

In `client/views/index.vue`:

```
<template>
  <p>
    <router-link to="/other">
      Go to another page
    </router-link>
  </p>
</template>
```

html

In `client/views/other.vue`:

```
<template>
  <p>This page is just for demonstrating client-side navigation.</p>
  <router-link to="/">
    Go back to index
```

html

```
</router-link>
</template>
```

These will be our *route modules*.

Next, let's add our core application files:

In `client/base.js`:

```
import { createSSRApp } from 'vue'
import {
  createRouter,
  createMemoryHistory,
  createWebHistory
} from 'vue-router'

import base from './base.vue'
import routes from './routes.js'

export async function createApp (ctx, url) {
  const instance = createSSRApp(base)
  const history = import.meta.env.SSR
    ? createMemoryHistory()
    : createWebHistory()
  const router = createRouter({ history, routes })

  instance.use(router)

  if (url) {
    router.push(url)
    await router.isReady()
  }

  return { ctx, router, instance }
}
```

js

A pretty standard Vue Router setup, with the appropriate history manager for SSR and client environments and awaiting on `isReady()` before returning the Vue instance.

In `client/routes.js`:

js

```
export default [
  {
    path: '/',
    component: () => import('./views/index.vue'),
  },
  {
    path: '/other',
    component: () => import('./views/other.vue')
  }
]
```

In `client/base.vue`:

html

```
<template>
  <router-view v-slot="{ Component }">
    <Suspense>
      <component
        :is="Component"
        :key="$route.path"
      />
    </Suspense>
  </router-view>
</template>
```

In `client/mount.js`:

js

```
import { createApp } from './base.js'

createApp(window.hydration)
```

```
.then(({ instance, router }) => {
  router.isReady().then(() => {
    instance.mount('root')
  })
})
```

This setup enables SSR in such a manner that the prerendered markup appears instantly on the client, but it continues to operate as SPA — that is, if you navigate to other URLs using `<router-link>`, there should be no full page reload, which is the default behavior of Next.js and Nuxt applications.

Finally in the `client/` folder, create `index.js`:

```
export { createApp } from './base.js'
export { default as routes } from './routes.js'
```

js

This is what we need to make available to the server for SSR.

Next, let's create our `@fastify/vite` renderer options.

In `renderer.js`:

```
import { renderToString } from 'vue/server-renderer'

export function createRoute ({ handler, errorHandler, route }, server) {
  server.route({
    url: route.path,
    method: 'GET',
    handler,
    errorHandler,
  })
}

export function createRenderFunction ({ createApp }) {
  return async function ({ app: server, req, reply }) {
```

js

```

const app = await createApp({ server, req, reply }, req.raw.url)
const element = await renderToString(app.instance, app.ctx)
return { element }
}
}

```

`@fastify/vite` will run the `createRoute()` hook for every route object present in the array exported by the `client/routes.js` file, **automatically**. Putting it all together:

```

import Fastify from 'fastify'
import FastifyVite from '@fastify/vite'
import * as renderer from './renderer.js'

export async function getServer (devFlag) {
  const dev = devFlag ?? process.argv.includes('--dev')

  const server = Fastify({
    logger: true,
    ...(process.stdout.isTTY || dev) && {
      logger: {
        transport: {
          target: '@fastify/one-line-logger'
        }
      }
    },
  })

  await server.register(FastifyVite, {
    root: import.meta.url,
    dev,
    renderer,
  })

  await server.vite.ready()
}

```

js

```
    return server
  }

  if (process.argv[1] === import.meta.filename) {
    const server = await getServer()
    await server.listen({ port: 3000 })
  }
}
```

Let's see our server-side rendered `/` route:

```
% curl http://localhost:3000/ sh
<!doctype html>
<script type="module" crossorigin src="/assets/index-DdvG5ZXm.js"></script>
<link rel="stylesheet" crossorigin href="/assets/index-Cc-1_JzF.css">
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<div id="root"><p><a href="/other" class=""> Go to another page </a></p></div>
```

On the client, if you click the link to `/other`, there will be no full page reload as it will be going through Vue Router. You can of course hit `/other` directly for its SSR version, in which case the same will happen: the rendered page stays working as a SPA.

This example is available in [examples/3/fastify-vite-routing](#).