Environment API for Frameworks

- DevEnvironment Communication Levels
  - RunnableDevEnvironment
  - FetchableDevEnvironment
  - raw DevEnvironment
- Environments During Build
  - Configuring the app build with builder.buildApp
  - The buildApp plugin hook
  - Building programmatically with createBuilder
- Environment Agnostic Code

:::info Release Candidate The Environment API is generally in the release candidate phase. We'll maintain stability in the APIs between major releases to allow the ecosystem to experiment and build upon them. However, note that some specific APIs are still considered experimental.

We plan to stabilize these new APIs (with potential breaking changes) in a future major release once downstream projects have had time to experiment with the new features and validate them.

Resources:

- Feedback discussion where we are gathering feedback about the new APIs.
- Environment API PR where the new APIs were implemented and reviewed.

Please share your feedback with us. :::

DevEnvironment Communication Levels

Since environments may run in different runtimes, communication against the environment may have constraints depending on the runtime. To allow frameworks to write runtime agnostic code easily, the Environment API provides three kinds of communication levels.

`RunnableDevEnvironment`

`RunnableDevEnvironment` is an environment that can communicate arbitrary JavaScript values with your application code. Importing a module returns its real, live exports (functions, class instances, and any other values), so frameworks can run their server entries directly. The implicit `ssr` environment and other non-client environments use a `RunnableDevEnvironment` by default during dev. You can guard access to the runner with the `isRunnableDevEnvironment` function.

Its `runner` is a `ModuleRunner`. You import modules through it with `runner.import(url)`, which fetches, transforms, and evaluates a module from the Vite module graph (the `url` accepts a file path, server path, or id relative to the root) and returns the instantiated module with full HMR support. It is the modern replacement for `server.ssrLoadModule`, so frameworks can migrate to it to enable HMR for their SSR dev story.

:::info Why it can communicate arbitrary values A `RunnableDevEnvironment` evaluates modules in the same runtime as the Vite server, so values cross the boundary in-process instead of being serialized. This is what distinguishes it from `FetchableDevEnvironment`, which can only communicate through serialized `Request`/`Response` objects over the Fetch API. As a result, using a `RunnableDevEnvironment` requires the runner's runtime to be the same as the one the Vite server is running in. :::

```
export class RunnableDevEnvironment extends DevEnvironment {
  public readonly runner: ModuleRunner
}

class ModuleRunner {
  /**
   * URL to execute.
   * Accepts file path, server path, or id relative to the root.
   * Returns an instantiated module (same as in ssrLoadModule)
   */
  public async import(url: string): Promise<Record<string, any>>
  /**
   * Other ModuleRunner methods...
   */
}

if (isRunnableDevEnvironment(server.environments.ssr)) {
```

```
  await server.environments.ssr.runner.import('/entry-point.js')
}
```

:::warning The `runner` is evaluated lazily only when it's accessed for the first time. Beware that Vite enables source map support when the `runner` is created by calling `process.setSourceMapsEnabled` or by overriding `Error.prepareStackTrace` if it's not available. :::

Given a Vite server configured in middleware mode as described by the SSR setup guide, let's implement the SSR middleware using the environment API. Remember that it doesn't have to be called `ssr`, so we'll name it `server` in this example. Error handling is omitted.

```
import fs from 'node:fs'
import path from 'node:path'
import { createServer } from 'vite'

const viteServer = await createServer({
  server: { middlewareMode: true },
  appType: 'custom',
  environments: {
    server: {
      // by default, modules are run in the same process as the vite server
    },
  },
})

// You might need to cast this to RunnableDevEnvironment in TypeScript or
// use isRunnableDevEnvironment to guard the access to the runner
const serverEnvironment = viteServer.environments.server

app.use('*', async (req, res, next) => {
  const url = req.originalUrl

  // 1. Read index.html
  const indexHtmlPath = path.resolve(import.meta.dirname, 'index.html')
  let template = fs.readFileSync(indexHtmlPath, 'utf-8')

  // 2. Apply Vite HTML transforms. This injects the Vite HMR client,
  //    and also applies HTML transforms from Vite plugins, e.g. global
  //    preambles from @vitejs/plugin-react
  template = await viteServer.transformIndexHtml(url, template)
```

```
  // 3. Load the server entry. import(url) automatically transforms
  //    ESM source code to be usable in Node.js! There is no bundling
  //    required, and provides full HMR support.
  const { render } = await serverEnvironment.runner.import(
    '/src/entry-server.js',
  )

  // 4. render the app HTML. This assumes entry-server.js's exported
  //     `render` function calls appropriate framework SSR APIs,
  //    e.g. ReactDOMServer.renderToString()
  const appHtml = await render(url)

  // 5. Inject the app-rendered HTML into the template.
  const html = template.replace(`<!--ssr-outlet-->`, appHtml)

  // 6. Send the rendered HTML back.
  res.status(200).set({ 'Content-Type': 'text/html' }).end(html)
})
```

When using environments that support HMR (such as `RunnableDevEnvironment`), you should add `import.meta.hot.accept()` in your server entry file for optimal behavior. Without this, server file changes will invalidate the entire server module graph:

```
// src/entry-server.js
export function render(...) { ... }

if (import.meta.hot) {
  import.meta.hot.accept()
}
```

`FetchableDevEnvironment`

:::info

We are looking for feedback on the `FetchableDevEnvironment` proposal.

:::

`FetchableDevEnvironment` is an environment that can communicate with its runtime via the Fetch API interface. Since the `RunnableDevEnvironment` is only possible to implement in a limited set of runtimes, we recommend to use the `FetchableDevEnvironment` instead of the `RunnableDevEnvironment`

.

A common reason to reach for it is a framework that wants to support a runtime that can't run Vite directly (e.g. Cloudflare Workers). A `RunnableDevEnvironment` can't be used there, since it requires the runner to share the Vite server's runtime so values can cross the boundary in-process. Standardizing on the Fetch API lets the framework keep a single request-handling path across all of its target runtimes: its dev middleware forwards each incoming browser request as a `Request` and sends the returned `Response` back to the browser, mirroring how the app handles requests in production.

This environment provides a standardized way of handling requests via the `handleRequest` method:

```
import {
  createServer,
  createFetchableDevEnvironment,
  isFetchableDevEnvironment,
} from 'vite'

const server = await createServer({
  server: { middlewareMode: true },
  appType: 'custom',
  environments: {
    custom: {
      dev: {
        createEnvironment(name, config) {
          return createFetchableDevEnvironment(name, config, {
            handleRequest(request: Request): Promise<Response> | Response {
              // handle Request and return a Response
            },
          })
        },
      },
    },
  },
})

// Any consumer of the environment API can now call `dispatchFetch`
if (isFetchableDevEnvironment(server.environments.custom)) {
  const response: Response = await server.environments.custom.dispatchFetch(
    new Request('http://example.com/request-to-handle'),
  )
```

```
}
```

:::warning Vite validates the input and output of the `dispatchFetch` method: the request must be an instance of the global `Request` class and the response must be the instance of the global `Response` class. Vite will throw a `TypeError` if this is not the case.

Note that although the `FetchableDevEnvironment` is implemented as a class, it is considered an implementation detail by the Vite team and might change at any moment. :::

raw `DevEnvironment`

If the environment does not implement the `RunnableDevEnvironment` or `FetchableDevEnvironment` interfaces, you need to set up the communication manually.

If your code can run in the same runtime as the user modules (i.e., it does not rely on Node.js-specific APIs), you can use a virtual module. This approach eliminates the need to access the value from the code using Vite's APIs.

```
// code using the Vite's APIs
import { createServer } from 'vite'

const server = await createServer({
  plugins: [
    // a plugin that handles `virtual:entrypoint`
    {
      name: 'virtual-module',
      /* plugin implementation */
    },
  ],
})
const ssrEnvironment = server.environments.ssr
const input = {}

// use exposed functions by each environment factories that runs the code
// check for each environment factories what they provide
if (ssrEnvironment instanceof CustomDevEnvironment) {
  ssrEnvironment.runEntrypoint('virtual:entrypoint')
} else {
  throw new Error(`Unsupported runtime for ${ssrEnvironment.name}`)
}
```

```
// ------------------------------------
// virtual:entrypoint
const { createHandler } = await import('./entrypoint.js')
const handler = createHandler(input)
const response = handler(new Request('http://example.com/'))

// ------------------------------------
// ./entrypoint.js
export function createHandler(input) {
  return function handler(req) {
    return new Response('hello')
  }
}
```

For example, to call `transformIndexHtml` on the user module, the following plugin can be used:

```
function vitePluginVirtualIndexHtml(): Plugin {
  let server: ViteDevServer | undefined
  return {
    name: vitePluginVirtualIndexHtml.name,
    configureServer(server_) {
      server = server_
    },
    resolveId(source) {
      return source === 'virtual:index-html' ? '\0' + source : undefined
    },
    async load(id) {
      if (id === '\0' + 'virtual:index-html') {
        let html: string
        if (server) {
          this.addWatchFile('index.html')
          html = fs.readFileSync('index.html', 'utf-8')
          html = await server.transformIndexHtml('/', html)
        } else {
          html = fs.readFileSync('dist/client/index.html', 'utf-8')
        }
        return `export default ${JSON.stringify(html)}`
      }
      return
    },
```

```
  }
}
```

If your code requires Node.js APIs, you can use `hot.send` to communicate with the code that uses Vite's APIs from the user modules. However, be aware that this approach may not work the same way after the build process.

```
// code using the Vite's APIs
import { createServer } from 'vite'

const server = await createServer({
  plugins: [
    // a plugin that handles `virtual:entrypoint`
    {
      name: 'virtual-module',
      /* plugin implementation */
    },
  ],
})
const ssrEnvironment = server.environments.ssr
const input = {}

// use exposed functions by each environment factories that runs the code
// check for each environment factories what they provide
if (ssrEnvironment instanceof RunnableDevEnvironment) {
  await ssrEnvironment.runner.import('virtual:entrypoint')
} else if (ssrEnvironment instanceof CustomDevEnvironment) {
  ssrEnvironment.runEntrypoint('virtual:entrypoint')
} else {
  throw new Error(`Unsupported runtime for ${ssrEnvironment.name}`)
}

const req = new Request('http://example.com/')

const uniqueId = 'a-unique-id'
ssrEnvironment.hot.send('request', serialize({ req, uniqueId }))
const response = await new Promise((resolve) => {
  ssrEnvironment.hot.on('response', (data) => {
    data = deserialize(data)
    if (data.uniqueId === uniqueId) {
```

```
      resolve(data.res)
    }
  })
})

// ------------------------------------
// virtual:entrypoint
const { createHandler } = await import('./entrypoint.js')
const handler = createHandler(input)

import.meta.hot.on('request', (data) => {
  const { req, uniqueId } = deserialize(data)
  const res = handler(req)
  import.meta.hot.send('response', serialize({ res: res, uniqueId }))
})

// ------------------------------------
// ./entrypoint.js
export function createHandler(input) {
  return function handler(req) {
    return new Response('hello')
  }
}
```

Environments During Build

In the CLI, calling `vite build` and `vite build --ssr` will still build the client only and ssr only environments for backward compatibility.

When the `builder` option is set (even to an empty object `{}`, which is what `vite build --app` does), `vite build` opts in to building the entire app instead. This will become the default in a future major. In this mode, Vite creates a `ViteBuilder` instance (the build-time equivalent of a `ViteDevServer`) and uses it to build all configured environments for production. By default, environments are built in series, following the order of the `environments` record.

Configuring the app build with `builder.buildApp`

A framework or user can control how the environments are built through the `builder.buildApp` option. It receives the `ViteBuilder` instance (named `builder` in the example below) and is responsible for building each environment; for instance, to build some of them in parallel:

```
import { defineConfig } from 'vite'

export default defineConfig({
  builder: {
    buildApp: async (builder) => {
      const environments = Object.values(builder.environments)
      await Promise.all(
        environments.map((environment) => builder.build(environment)),
      )
    },
  },
})
```

The `buildApp` plugin hook

- **Type:** `(this: MinimalPluginContextWithoutEnvironment, builder: ViteBuilder) => Promise<void>`
- **Kind:** `async`, `sequential`
- **Scope:** Global

Besides the `builder.buildApp` config option, plugins can define a `buildApp` hook to participate in the app build. The config option and the plugin hooks run in a defined order: hooks with order `'pre'` or `null` run first, then the configured `builder.buildApp`, then hooks with order `'post'`. Within a hook, `environment.isBuilt` tells you whether an environment has already been built, which lets a plugin avoid building it twice.

Building programmatically with `createBuilder`

To trigger an app build from your own code, use `createBuilder` instead of the standalone `build` function. `createBuilder` is the build-time equivalent of `createServer`: it resolves the config and returns a `ViteBuilder`, whose `buildApp` method builds every configured environment. You can also build a single environment with `builder.build(environment)`.

```
import { createBuilder } from 'vite'
```

```
const builder = await createBuilder()
await builder.buildApp()
```

`createBuilder` supersedes the standalone `build` function for environment-aware builds. `build` still works as the simple entry point for the legacy client-only and ssr-only builds described above, but it cannot build arbitrary environments. Running `builder.buildApp()` is the programmatic equivalent of `vite build --app`.

Environment Agnostic Code

Most of the time, the current `environment` instance will be available as part of the context of the code being run so the need to access them through `server.environments` should be rare. For example, inside plugin hooks the environment is exposed as part of the `PluginContext`, so it can be accessed using `this.environment`. See Environment API for Plugins to learn about how to build environment aware plugins.

□□□ #2
□ □□□ □□□ 2026-10-31 13:13:43 UTC
□ □□□ □□□ 2026-10-31 13:13:45 UTC