Environment API for Runtimes

- Environment Factories
- Creating a New Environment Factory
- ModuleRunner
- ModuleRunnerOptions
- ModuleEvaluator
- ModuleRunnerTransport

:::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. :::

This page is for runtime providers, authors who integrate a JavaScript runtime with Vite. A runtime here is the JavaScript engine where transformed code executes, such as Node.js, the browser, Cloudflare's workerd, or a Worker thread. A runtime provider packages the integration for one of these runtimes, so framework authors and end users (the developers building an app) don't have to set it up themselves.

Environment Factories

Environment factories are intended to be implemented by runtime providers, not by end users. Environment factories return an `EnvironmentOptions` for the most common case of using the target runtime for both dev and build environments. The default environment options can also be set so the user doesn't need to do it.

```
function createWorkerdEnvironment(
  userConfig: EnvironmentOptions,
): EnvironmentOptions {
  return mergeConfig(
    {
      resolve: {
        conditions: [/*...*/],
      },
      dev: {
        createEnvironment(name, config) {
          return createWorkerdDevEnvironment(name, config, {
            hot: true,
            transport: customHotChannel(),
          })
        },
      },
      build: {
        createEnvironment(name, config) {
          return createWorkerdBuildEnvironment(name, config)
        },
      },
    },
    userConfig,
  )
}
```

Then the config file can be written as:

```
import { createWorkerdEnvironment } from 'vite-environment-workerd'

export default {
  environments: {
    ssr: createWorkerdEnvironment({
      build: {
        outDir: '/dist/ssr',
      },
```

```
    }),
    rsc: createWorkerdEnvironment({
      build: {
        outDir: '/dist/rsc',
      },
    }),
  },
}
```

and frameworks can use an environment with the workerd runtime to do SSR using:

```
const ssrEnvironment = server.environments.ssr
```

Creating a New Environment Factory

A Vite dev server exposes two environments by default: a `client` environment and an `ssr` environment. The client environment is a browser environment by default, and the module runner is implemented by importing the virtual module `/@vite/client` to client apps. The SSR environment runs in the same Node runtime as the Vite server by default and allows application servers to be used to render requests during dev with full HMR support.

The transformed source code is called a module, and the relationships between the modules processed in each environment are kept in a module graph. The transformed code for these modules is sent to the runtimes associated with each environment to be executed. When a module is evaluated in the runtime, its imported modules will be requested triggering the processing of a section of the module graph.

A Vite Module Runner allows running any code by processing it with Vite plugins first. It is different from `server.ssrLoadModule` because the runner implementation is decoupled from the server. This allows library and framework authors to implement their layer of communication between the Vite server and the runner. The browser communicates with its corresponding environment using the server WebSocket and through HTTP requests. The Node Module runner can directly do function calls to process modules as it is running in the same process. Other environments could run modules connecting to a JS runtime like workerd, or a Worker Thread as Vitest does.

```
digraph module_runner {
  rankdir=LR
  node [shape=box style="rounded,filled" fontname="Arial" fontsize=11 margin="0.2,0.1"
fontcolor="${#3c3c43|#ffffff}" color="${#c2c2c4|#3c3f44}"]
```

```
  edge [color="${#67676c|#98989f}" fontname="Arial" fontsize=10
fontcolor="${#67676c|#98989f}"]
  bgcolor="transparent"
  compound=true

  subgraph cluster_server {
    label="Vite Dev Server (Node.js)" labeljust=l fontname="Arial" fontsize=12
    style="rounded,filled" fillcolor="${#f6f6f7|#1a1a1f}" color="${#c2c2c4|#3c3f44}"
    fontcolor="${#3c3c43|#ffffff}"

    subgraph cluster_env {
      label="DevEnvironment" labeljust=l fontname="Arial" fontsize=11
      style="rounded,filled" fillcolor="${#f2ecfc|#2c273e}" color="${#c2c2c4|#3c3f44}"
      fontcolor="${#3c3c43|#ffffff}"

      plugins [label="Plugin\nPipeline" fillcolor="${#e9eaff|#222541}"]
      mg [label="Module\nGraph" fillcolor="${#e9eaff|#222541}"]
      hot [label="HotChannel" fillcolor="${#fcf4dc|#38301a}"]

      plugins -> mg [dir=both]
      mg -> hot [style=invis]
    }
  }

  subgraph cluster_runtime {
    label="Target Runtime" labeljust=l fontname="Arial" fontsize=12
    style="rounded,filled" fillcolor="${#f0fdf4|#131b15}" color="${#c2c2c4|#3c3f44}"
    fontcolor="${#3c3c43|#ffffff}"

    subgraph cluster_runner {
      label="ModuleRunner" labeljust=l fontname="Arial" fontsize=11
      style="rounded,filled" fillcolor="${#def5ed|#15312d}" color="${#c2c2c4|#3c3f44}"
      fontcolor="${#3c3c43|#ffffff}"

      evaluator [label="Module\nEvaluator" fillcolor="${#def5ed|#15312d}"]
      transport [label="Transport" fillcolor="${#fcf4dc|#38301a}"]
    }
  }

  hot -> transport [label="HMR / Module\nfetch & invoke" dir=both style=bold
```

```
color="${#6f42c1|#c8abfa}"]
}
```

One of the goals of this feature is to provide a customizable API to process and run code. Users can create new environment factories using the exposed primitives.

```
import { DevEnvironment, HotChannel } from 'vite'

function createWorkerdDevEnvironment(
  name: string,
  config: ResolvedConfig,
  context: DevEnvironmentContext
) {
  const connection = /* ... */
  const transport: HotChannel = {
    on: (listener) => { connection.on('message', listener) },
    send: (data) => connection.send(data),
  }

  const workerdDevEnvironment = new DevEnvironment(name, config, {
    options: {
      resolve: { conditions: ['custom'] },
      ...context.options,
    },
    hot: true,
    transport,
  })
  return workerdDevEnvironment
}
```

By default, `HotChannel` transports have `server.fs` restrictions applied, meaning only files within the allowed directories can be served. If your transport is not exposed over the network (e.g., it communicates via worker threads or in-process calls), you can set `skipFsCheck: true` on the `HotChannel` to bypass these restrictions.

There are multiple communication levels for the `DevEnvironment`. To make it easier for frameworks to write runtime agnostic code, we recommend to implement the most flexible communication level possible.

`ModuleRunner`

A module runner is instantiated in the target runtime. All APIs in the next section are imported from `vite/module-runner` unless stated otherwise. This export entry point is kept as lightweight as possible, only exporting the minimal needed to create module runners.

Type Signature:

```
export class ModuleRunner {
  constructor(
    public options: ModuleRunnerOptions,
    public evaluator: ModuleEvaluator = new ESModulesEvaluator(),
    private debug?: ModuleRunnerDebugger,
  ) {}
  /**
   * URL to execute.
   * Accepts file path, server path, or id relative to the root.
   */
  public async import<T = any>(url: string): Promise<T>
  /**
   * Clear all caches including HMR listeners.
   */
  public clearCache(): void
  /**
   * Clear all caches, remove all HMR listeners, reset sourcemap support.
   * This method doesn't stop the HMR connection.
   */
  public async close(): Promise<void>
  /**
   * Returns `true` if the runner has been closed by calling `close()`.
   */
  public isClosed(): boolean
}
```

The module evaluator in `ModuleRunner` is responsible for executing the code. Vite exports `ESModulesEvaluator` out of the box, it uses `new AsyncFunction` to evaluate the code. You can provide your own implementation if your JavaScript runtime doesn't support unsafe evaluation.

Module runner exposes `import` method. When Vite server triggers `full-reload` HMR event, all affected modules will be re-executed. Be aware that Module Runner doesn't update `exports` object when this happens (it overrides it), you would need to run `import` or get the module from `evaluatedModules` again if you rely on having the latest `exports` object.

Example Usage:

```
import {
  ModuleRunner,
  ESModulesEvaluator,
  createNodeImportMeta,
} from 'vite/module-runner'
import { transport } from './rpc-implementation.js'

const moduleRunner = new ModuleRunner(
  {
    transport,
    createImportMeta: createNodeImportMeta, // if the module runner runs in Node.js
  },
  new ESModulesEvaluator(),
)

await moduleRunner.import('/src/entry-point.js')
```

ModuleRunnerOptions

```
import type { Debug } from '@type-challenges/utils'
import type {
  InterceptorOptions as InterceptorOptionsRaw,
  ModuleRunnerHmr as ModuleRunnerHmrRaw,
  EvaluatedModules,
} from 'vite/module-runner'

type InterceptorOptions = Debug<InterceptorOptionsRaw>
type ModuleRunnerHmr = Debug<ModuleRunnerHmrRaw>
/** see below */
type ModuleRunnerTransport = unknown

// ---cut---
interface ModuleRunnerOptions {
  /**
   * A set of methods to communicate with the server.
   */
  transport: ModuleRunnerTransport
```

```
  /**
   * Configure how source maps are resolved.
   * Prefers `node` if `process.setSourceMapsEnabled` is available.
   * Otherwise it will use `prepareStackTrace` by default which overrides
   * `Error.prepareStackTrace` method.
   * You can provide an object to configure how file contents and
   * source maps are resolved for files that were not processed by Vite.
   */
  sourcemapInterceptor?:
    | false
    | 'node'
    | 'prepareStackTrace'
    | InterceptorOptions
  /**
   * Disable HMR or configure HMR options.
   *
   * @default true
   */
  hmr?: boolean | ModuleRunnerHmr
  /**
   * Custom module cache. If not provided, it creates a separate module
   * cache for each module runner instance.
   */
  evaluatedModules?: EvaluatedModules
}
```

ModuleEvaluator

Type Signature:

```
import type { Debug } from '@type-challenges/utils'
import type { ModuleRunnerContext as ModuleRunnerContextRaw } from 'vite/module-runner'

type ModuleRunnerContext = Debug<ModuleRunnerContextRaw>

// ---cut---
export interface ModuleEvaluator {
  /**
   * Number of prefixed lines in the transformed code.
```

```
   */
  startOffset?: number
  /**
   * Evaluate code that was transformed by Vite.
   * @param context Function context
   * @param code Transformed code
   * @param id ID that was used to fetch the module
   */
  runInlinedModule(
    context: ModuleRunnerContext,
    code: string,
    id: string,
  ): Promise<any>
  /**
   * evaluate externalized module.
   * @param file File URL to the external module
   */
  runExternalModule(file: string): Promise<any>
}
```

Vite exports `ESModulesEvaluator` that implements this interface by default. It uses `new AsyncFunction` to evaluate code, so if the code has inlined source map it should contain an offset of 2 lines to accommodate for new lines added. This is done automatically by the `ESModulesEvaluator`. Custom evaluators will not add additional lines.

ModuleRunnerTransport

Type Signature:

```
import type { ModuleRunnerTransportHandlers } from 'vite/module-runner'
/** an object */
type HotPayload = unknown
// ---cut---
interface ModuleRunnerTransport {
  connect?(handlers: ModuleRunnerTransportHandlers): Promise<void> | void
  disconnect?(): Promise<void> | void
  send?(data: HotPayload): Promise<void> | void
  invoke?(data: HotPayload): Promise<{ result: any } | { error: any }>
  timeout?: number
```

```
}
```

Transport object that communicates with the environment via an RPC or by directly calling the function. When `invoke` method is not implemented, the `send` method and `connect` method is required to be implemented. Vite will construct the `invoke` internally.

You need to couple it with the `HotChannel` instance on the server like in this example where module runner is created in the worker thread:

::: code-group

```
import { fileURLToPath } from 'node:url'
import { parentPort } from 'node:worker_threads'
import {
  ESModulesEvaluator,
  ModuleRunner,
  createNodeImportMeta,
} from 'vite/module-runner'

/** @type {import('vite/module-runner').ModuleRunnerTransport} */
const transport = {
  connect({ onMessage, onDisconnection }) {
    parentPort.on('message', onMessage)
    parentPort.on('close', onDisconnection)
  },
  send(data) {
    parentPort.postMessage(data)
  },
}

const runner = new ModuleRunner(
  {
    transport,
    createImportMeta: createNodeImportMeta,
  },
  new ESModulesEvaluator(),
)
```

```
import { BroadcastChannel } from 'node:worker_threads'
import { createServer, DevEnvironment } from 'vite'
```

```
function createWorkerEnvironment(name, config, context) {
  const worker = new Worker('./worker.js')
  const handlerToWorkerListener = new WeakMap()
  const client = {
    send(payload: HotPayload) {
      worker.postMessage(payload)
    },
  }

  const workerHotChannel = {
    // Worker threads post messages are not exposed over the network, skip server.fs checks
    skipFsCheck: true,
    send: (data) => worker.postMessage(data),
    on: (event, handler) => {
      // client is already connected
      if (event === 'vite:client:connect') return
      if (event === 'vite:client:disconnect') {
        const listener = () => {
          handler(undefined, client)
        }
        handlerToWorkerListener.set(handler, listener)
        worker.on('exit', listener)
        return
      }

      const listener = (value) => {
        if (value.type === 'custom' && value.event === event) {
          handler(value.data, client)
        }
      }
      handlerToWorkerListener.set(handler, listener)
      worker.on('message', listener)
    },
    off: (event, handler) => {
      if (event === 'vite:client:connect') return
      if (event === 'vite:client:disconnect') {
        const listener = handlerToWorkerListener.get(handler)
        if (listener) {
          worker.off('exit', listener)
          handlerToWorkerListener.delete(handler)
```

```
        }
        return
      }

      const listener = handlerToWorkerListener.get(handler)
      if (listener) {
        worker.off('message', listener)
        handlerToWorkerListener.delete(handler)
      }
    },
  }

  return new DevEnvironment(name, config, {
    transport: workerHotChannel,
  })
}

await createServer({
  environments: {
    worker: {
      dev: {
        createEnvironment: createWorkerEnvironment,
      },
    },
  },
})
```

:::

Make sure to implement the `vite:client:connect` / `vite:client:disconnect` events in the `on` / `off` methods when those methods exist. `vite:client:connect` event should be emitted when the connection is established, and `vite:client:disconnect` event should be emitted when the connection is closed. The `HotChannelClient` object passed to the event handler must have the same reference for the same connection.

A different example using an HTTP request to communicate between the runner and the server:

```
import { ESModulesEvaluator, ModuleRunner } from 'vite/module-runner'

export const runner = new ModuleRunner(
  {
```

```
    transport: {
      async invoke(data) {
        const response = await fetch(`http://my-vite-server/invoke`, {
          method: 'POST',
          body: JSON.stringify(data),
        })
        return response.json()
      },
    },
    hmr: false, // disable HMR as HMR requires transport.connect
  },
  new ESModulesEvaluator(),
)

await runner.import('/entry.js')
```

In this case, the `handleInvoke` method in the `NormalizedHotChannel` can be used:

```
const customEnvironment = new DevEnvironment(name, config, context)

server.onRequest((request: Request) => {
  const url = new URL(request.url)
  if (url.pathname === '/invoke') {
    const payload = (await request.json()) as HotPayload
    const result = customEnvironment.hot.handleInvoke(payload)
    return new Response(JSON.stringify(result))
  }
  return Response.error()
})
```

But note that for HMR support, `send` and `connect` methods are required. The `send` method is usually called when the custom event is triggered (like, `import.meta.hot.send("my-event")`).

For an SSR environment running in the same Node.js process as the Vite server, Vite exports `createServerHotChannel` as a ready-made `HotChannel`:

```
import { createServerHotChannel, DevEnvironment } from 'vite'

new DevEnvironment(name, config, {
  hot: true,
  transport: createServerHotChannel(),
})
```

□□□ #2
□ □□□ □□□ 2026-10-31 13:13:52 UTC
□ □□□ □□□ 2026-10-31 13:13:53 UTC