```
---
sidebar: false
editLink: false
outline: false
---

<script setup>
import BlogIndex from './.vitepress/theme/components/BlogIndex.vue'
</script>

# Latest From the Vite Blog

<BlogIndex/>
```

---
title: Configuring Vite

# Configuring Vite

When running `vite` from the command line, Vite will automatically try to resolve a config file named `vite.config.js` inside [project root](/guide/#index-html-and-project-root) (other JS and TS extensions are also supported).

The most basic config file looks like this:

```js
// vite.config.js
export default {
  // config options
}
```

Note Vite supports using ES modules syntax in the config file even if the project is not using native Node ESM, e.g. `type: "module"` in `package.json`. In this case, the config file is auto pre-processed before load.

You can also explicitly specify a config file to use with the `--config` CLI option (resolved relative to `cwd`):

```bash
vite --config my-config.js
```

## Config Intellisense

Since Vite ships with TypeScript typings, you can leverage your IDE's intellisense with jsdoc type hints:

```js
/** @type {import('vite').UserConfig} */
export default {
  // ...
}
```

Alternatively, you can use the `defineConfig` helper which should provide intellisense without the need for jsdoc annotations:

```js
import { defineConfig } from 'vite'

export default defineConfig({
  // ...
})
```

Vite also supports TypeScript config files. You can use `vite.config.ts` with the `defineConfig` helper function above, or with the `satisfies` operator:

```ts
import type { UserConfig } from 'vite'

export default {
  // ...
} satisfies UserConfig
```

## Conditional Config

If the config needs to conditionally determine options based on the command (`serve` or `build`), the [mode](/guide/env-and-mode) being used, if it's an SSR build (`isSsrBuild`), or is previewing the build (`isPreview`), it can export a function instead:

```js twoslash
import { defineConfig } from 'vite'
// ---cut---
export default defineConfig(({ command, mode, isSsrBuild, isPreview }) => {
  if (command === 'serve') {
    return {
      // dev specific config
    }
  } else {
    // command === 'build'
    return {
      // build specific config
    }
  }
})
```

It is important to note that in Vite's API the `command` value is `serve` during dev (in the cli [`vite`](/guide/cli#vite), `vite dev`, and `vite serve` are aliases), and `build` when building for production ([`vite build`](/guide/cli#vite-build)).

`isSsrBuild` and `isPreview` are additional optional flags to differentiate the kind of `build` and `serve` commands respectively. Some tools that load the Vite config may not support these flags and will pass `undefined` instead. Hence, it's recommended to use explicit comparison against `true` and `false`.

## Async Config

If the config needs to call async functions, it can export an async function instead. And this async function can also be passed through `defineConfig` for improved intellisense support:

```js twoslash
import { defineConfig } from 'vite'
// ---cut---
export default defineConfig(async ({ command, mode }) => {
  const data = await asyncFunction()
  return {
    // vite config
  }
})
```

## Using Environment Variables in Config

Environmental Variables can be obtained from `process.env` as usual.

Note that Vite doesn't load `.env` files by default as the files to load can only be determined after evaluating the Vite config, for example, the `root` and `envDir` options affect the loading behaviour. However, you can use the exported `loadEnv` helper to load the specific `.env` file if needed.

```js twoslash
import { defineConfig, loadEnv } from 'vite'

export default defineConfig(({ command, mode }) => {
  // Load env file based on `mode` in the current working directory.
  // Set the third parameter to '' to load all env regardless of the `VITE_` prefix.
  const env = loadEnv(mode, process.cwd(), '')
  return {
    // vite config
    define: {
      __APP_ENV__: JSON.stringify(env.APP_ENV),
    },
  }
})
```

```yaml
---
title: Announcing Vite 2.0
author:
  - name: The Vite Team
sidebar: false
date: 2021-02-16
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 2.0
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite2
  - - meta
    - property: og:description
      content: Vite 2 Release Announcement
---
```

# Announcing Vite 2.0

February 16, 2021 - Check out the [Vite 3.0 announcement](./announcing-vite3.md)

```html
<p style="text-align:center">
  <img src="/logo.svg" style="height:200px">
</p>
```

Today we are excited to announce the official release of Vite 2.0!

Vite (French word for "fast", pronounced `/vit/`) is a new kind of build tool for frontend web development. Think a pre-configured dev server + bundler combo, but leaner and faster. It leverages browser's [native ES modules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules) support and tools written in compile-to-native languages like [esbuild](https://esbuild.github.io/) to deliver a snappy and modern development experience.

To get a sense of how fast Vite is, check out [this video comparison](https://twitter.com/amasad/status/1355379680275128321) of booting up a React application on Repl.it using Vite vs. `create-react-app` (CRA).

If you've never heard of Vite before and would love to learn more about it, check out [the rationale behind the project](https://vitejs.dev/guide/why.html). If you are interested in how Vite differs from other similar tools, check out the [comparisons](https://vitejs.dev/guide/comparisons.html).

## What's New in 2.0

Since we decided to completely refactor the internals before 1.0 got out of RC, this is in fact the first stable release of Vite. That said, Vite 2.0 brings about many big improvements over its previous incarnation:

### Framework Agnostic Core

The original idea of Vite started as a [hacky prototype that serves Vue single-file components over native ESM](https://github.com/vuejs/vue-dev-server). Vite 1 was a continuation of that idea with HMR implemented on top.

Vite 2.0 takes what we learned along the way and is redesigned from scratch with a more robust internal architecture. It is now completely framework agnostic, and all framework-specific support is delegated to plugins. There are now [official templates for Vue, React, Preact, Lit Element](https://github.com/vitejs/vite/tree/main/packages/create-vite), and ongoing community efforts for Svelte integration.

### New Plugin Format and API

Inspired by [WMR](https://github.com/preactjs/wmr), the new plugin system extends Rollup's plugin interface and is [compatible with many Rollup plugins](https://vite-rollup-plugins.patak.dev/) out of the box. Plugins can use Rollup-compatible hooks, with additional Vite-specific hooks and properties to adjust Vite-only behavior (e.g. differentiating dev vs. build or custom handling of HMR).

The [programmatic API](https://vitejs.dev/guide/api-javascript.html) has also been greatly improved to facilitate higher level tools / frameworks built on top of Vite.

### esbuild Powered Dep Pre-Bundling

Since Vite is a native ESM dev server, it pre-bundles dependencies to reduce the number browser requests and handle CommonJS to ESM conversion. Previously Vite did this using Rollup, and in 2.0 it now uses `esbuild` which results in 10-100x faster dependency pre-bundling. As a reference, cold-booting a test app with heavy dependencies like React Material UI previously took 28 seconds on an M1-powered MacBook Pro and now takes ~1.5 seconds. Expect similar improvements if you are switching from a traditional bundler based setup.

### First-class CSS Support

Vite treats CSS as a first-class citizen of the module graph and supports the following out of the box:

- **Resolver enhancement**: `@import` and `url()` paths in CSS are enhanced with Vite's resolver to respect aliases and npm dependencies.
- **URL rebasing**: `url()` paths are automatically rebased regardless of where the file is imported from.

- **CSS code splitting**: a code-split JS chunk also emits a corresponding CSS file, which is automatically loaded in parallel with the JS chunk when requested.

### Server-Side Rendering (SSR) Support

Vite 2.0 ships with [experimental SSR support](https://vitejs.dev/guide/ssr.html). Vite provides APIs to efficiently load and update ESM-based source code in Node.js during development (almost like server-side HMR), and automatically externalizes CommonJS-compatible dependencies to improve development and SSR build speed. The production server can be completely decoupled from Vite, and the same setup can be easily adapted to perform pre-rendering / SSG.

Vite SSR is provided as a low-level feature and we are expecting to see higher level frameworks leveraging it under the hood.

### Opt-in Legacy Browser Support

Vite targets modern browsers with native ESM support by default, but you can also opt-in to support legacy browsers via the official [@vitejs/plugin-legacy](https://github.com/vitejs/vite/tree/main/packages/plugin-legacy). The plugin automatically generates dual modern/legacy bundles, and delivers the right bundle based on browser feature detection, ensuring more efficient code in modern browsers that support them.

## Give it a Try!

That was a lot of features, but getting started with Vite is simple! You can spin up a Vite-powered app literally in a minute, starting with the following command (make sure you have Node.js >=12):

```bash
npm init @vitejs/app
```

Then, check out [the guide](https://vitejs.dev/guide/) to see what Vite provides out of the box. You can also check out the source code on [GitHub](https://github.com/vitejs/vite), follow updates on [Twitter](https://twitter.com/vite_js), or join discussions with other Vite users on our [Discord chat server](http://chat.vitejs.dev/).

# Dep Optimization Options

- **Related:** [Dependency Pre-Bundling](/guide/dep-pre-bundling)

## optimizeDeps.entries

- **Type:** `string | string[]`

By default, Vite will crawl all your `.html` files to detect dependencies that need to be pre-bundled (ignoring `node_modules`, `build.outDir`, `__tests__` and `coverage`). If `build.rollupOptions.input` is specified, Vite will crawl those entry points instead.

If neither of these fit your needs, you can specify custom entries using this option - the value should be a [fast-glob pattern](https://github.com/mrmlnc/fast-glob#basic-syntax) or array of patterns that are relative from Vite project root. This will overwrite default entries inference. Only `node_modules` and `build.outDir` folders will be ignored by default when `optimizeDeps.entries` is explicitly defined. If other folders need to be ignored, you can use an ignore pattern as part of the entries list, marked with an initial `!`. If you don't want to ignore `node_modules` and `build.outDir`, you can specify using literal string paths (without fast-glob patterns) instead.

## optimizeDeps.exclude

- **Type:** `string[]`

Dependencies to exclude from pre-bundling.

:::warning CommonJS
CommonJS dependencies should not be excluded from optimization. If an ESM dependency is excluded from optimization, but has a nested CommonJS dependency, the CommonJS dependency should be added to `optimizeDeps.include`. Example:

```js twoslash
import { defineConfig } from 'vite'
// ---cut---
export default defineConfig({
  optimizeDeps: {
    include: ['esm-dep > cjs-dep'],
  },
})
```

:::

## optimizeDeps.include

- **Type:** `string[]`

By default, linked packages not inside `node_modules` are not pre-bundled. Use this option to force a linked package to be pre-bundled.

Experimental: If you're using a library with many deep imports, you can also specify a trailing glob pattern to pre-bundle all deep imports at once. This will avoid constantly pre-bundling whenever a new deep import is used. [Give Feedback](https://github.com/vitejs/vite/discussions/15833). For example:

```js twoslash
import { defineConfig } from 'vite'
// ---cut---
export default defineConfig({
  optimizeDeps: {
    include: ['my-lib/components/**/*.vue'],
  },
})
```

## optimizeDeps.esbuildOptions

- **Type:** [`Omit`](https://www.typescriptlang.org/docs/handbook/utility-types.html#omittype-keys)`<`[`EsbuildBuildOptions`](https://esbuild.github.io/api/#simple-options)`,
| 'bundle'
| 'entryPoints'
| 'external'
| 'write'
| 'watch'
| 'outdir'
| 'outfile'
| 'outbase'
| 'outExtension'
| 'metafile'>`

Options to pass to esbuild during the dep scanning and optimization.

Certain options are omitted since changing them would not be compatible with Vite's dep optimization.

- `external` is also omitted, use Vite's `optimizeDeps.exclude` option
- `plugins` are merged with Vite's dep plugin

## optimizeDeps.force

- **Type:** `boolean`

Set to `true` to force dependency pre-bundling, ignoring previously cached optimized dependencies.

## optimizeDeps.holdUntilCrawlEnd

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/15834)
- **Type:** `boolean`
- **Default:** `true`

When enabled, it will hold the first optimized deps results until all static imports are crawled on cold start. This avoids the need for full-page reloads when new dependencies are discovered and they trigger the generation of new common chunks. If all dependencies are found by the scanner plus the explicitly defined ones in `include`, it is better to disable this option to let the browser process more requests in parallel.

## optimizeDeps.disabled

- **Deprecated**
- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/13839)
- **Type:** `boolean | 'build' | 'dev'`
- **Default:** `'build'`

This option is deprecated. As of Vite 5.1, pre-bundling of dependencies during build have been removed. Setting `optimizeDeps.disabled` to `true` or `'dev'` disables the optimizer, and configured to `false` or `'build'` leaves the optimizer during dev enabled.

To disable the optimizer completely, use `optimizeDeps.noDiscovery: true` to disallow automatic discovery of dependencies and leave `optimizeDeps.include` undefined or empty.

:::warning
Optimizing dependencies during build time was an **experimental** feature. Projects trying out this strategy also removed `@rollup/plugin-commonjs` using `build.commonjsOptions: { include: [] }`. If you did so, a warning will guide you to re-enable it to support CJS only packages while bundling.
:::

## optimizeDeps.needsInterop

- **Experimental**
- **Type:** `string[]`

Forces ESM interop when importing these dependencies. Vite is able to properly detect when a dependency needs interop, so this option isn't generally needed. However, different combinations of dependencies could cause some of them to be prebundled differently. Adding these packages to `needsInterop` can speed up cold start by avoiding full-page reloads. You'll receive a warning if this is the case for one of your dependencies, suggesting to add the package name to this array in your config.

```
---
title: Vite 4.3 is out!
author:
  name: The Vite Team
date: 2023-04-20
sidebar: false
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 4.3
  - - meta
    - property: og:image
      content: https://vitejs.dev/og-image-announcing-vite4-3.png
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite4-3
  - - meta
    - property: og:description
      content: Vite 4.3 Release Announcement
  - - meta
    - name: twitter:card
      content: summary_large_image
---

# Vite 4.3 is out!

_April 20, 2023_

![Vite 4.3 Announcement Cover Image](/og-image-announcing-vite4-3.png)

Quick links:

- Docs: [English](/), [ç®€ä½“ä¸æ–‡](https://cn.vitejs.dev/), [æ—¥æœ¬èªž](https://ja.vitejs.dev/), [EspaÃ±ol]
(https://es.vitejs.dev/), [PortuguÃªs](https://pt.vitejs.dev/)
- [Vite 4.3 Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#430-2023-04-20)

## Performance Improvements

In this minor, we focused on improving the dev server performance. The resolve logic got streamlined, improving hot paths and
implementing smarter caching for finding `package.json`, TS config files, and resolved URL in general.

You can read a detailed walkthrough of the performance work done in this blog post by one of Vite Contributors: [How we made
Vite 4.3 faaaaster ðŸš€](https://sun0day.github.io/blog/vite/why-vite4_3-is-faster.html).

This sprint resulted in speed improvements across the board compared to Vite 4.2.

These are the performance improvements as measured by [sapphi-red/performance-compare](https://github.com/sapphi-
red/performance-compare), which tests an app with 1000 React Components cold and warm dev server startup time as well as HMR
times for a root and a leaf component:

| **Vite (babel)**   |  Vite 4.2 | Vite 4.3 | Improvement |
| :----------------- | --------: | -------: | ----------: |
| **dev cold start** | 17249.0ms | 5132.4ms |      -70.2% |
| **dev warm start** |  6027.8ms | 4536.1ms |      -24.7% |
| **Root HMR**       |    46.8ms |   26.7ms |      -42.9% |
| **Leaf HMR**       |    27.0ms |   12.9ms |      -52.2% |

| **Vite (swc)**     |  Vite 4.2 | Vite 4.3 | Improvement |
| :----------------- | --------: | -------: | ----------: |
| **dev cold start** | 13552.5ms | 3201.0ms |      -76.4% |
| **dev warm start** |  4625.5ms | 2834.4ms |      -38.7% |
| **Root HMR**       |    30.5ms |   24.0ms |      -21.3% |
| **Leaf HMR**       |    16.9ms |   10.0ms |      -40.8% |

![Vite 4.3 vs 4.2 startup time comparison](/vite4-3-startup-time.png)

![Vite 4.3 vs 4.2 HMR time comparison](/vite4-3-hmr-time.png)

You can read more information about the benchmark [here](https://gist.github.com/sapphi-
red/25be97327ee64a3c1dce793444afdf6e). Specs and Versions for this performance run:

- CPU: Ryzen 9 5900X, Memory: DDR4-3600 32GB, SSD: WD Blue SN550 NVME SSD
- Windows 10 Pro 21H2 19044.2846
- Node.js 18.16.0
- Vite and React Plugin versions
  - Vite 4.2 (babel): Vite 4.2.1 + plugin-react 3.1.0
  - Vite 4.3 (babel): Vite 4.3.0 + plugin-react 4.0.0-beta.1
  - Vite 4.2 (swc): Vite 4.2.1 + plugin-react-swc 3.2.0
  - Vite 4.3 (swc): Vite 4.3.0 + plugin-react-swc 3.3.0
```

Early adopters have also reported seeing 1.5x-2x dev startup time improvement on real apps while testing the Vite 4.3 beta. We'd love to know the results for your apps.

## Profiling

We'll continue to work on Vite's performance. We're working on an official [Benchmark tool](https://github.com/vitejs/vite-benchmark) for Vite that let us get performance metrics for each Pull Request.

And [vite-plugin-inspect](https://github.com/antfu/vite-plugin-inspect) now has more performance-related features to help you identify which plugins or middlewares are the bottleneck for your applications.

Using `vite --profile` (and then pressing `p`) once the page loads will save a CPU profile of the dev server startup. You can open them in an app as [speedscope](https://www.speedscope.app/) to identify performance issues. And you can share your findings with the Vite Team in a [Discussion](https://github.com/vitejs/vite/discussions) or in [Vite's Discord](https://chat.vitejs.dev).

## Next Steps

We decided to do a single Vite Major this year aligning with the [EOL of Node.js 16](https://endoflife.date/nodejs) in September, dropping support for both Node.js 14 and 16 in it. If you would like to get involved, we started a [Vite 5 Discussion](https://github.com/vitejs/vite/discussions/12466) to gather early feedback.

# Build Options

## build.target

- **Type:** `string | string[]`
- **Default:** `'modules'`
- **Related:** [Browser Compatibility](/guide/build#browser-compatibility)

Browser compatibility target for the final bundle. The default value is a Vite special value, `'modules'`, which targets browsers with [native ES Modules](https://caniuse.com/es6-module), [native ESM dynamic import](https://caniuse.com/es6-module-dynamic-import), and [`import.meta`](https://caniuse.com/mdn-javascript_operators_import_meta) support. Vite will replace `'modules'` to `['es2020', 'edge88', 'firefox78', 'chrome87', 'safari14']`

Another special value is `'esnext'` - which assumes native dynamic imports support and will transpile as little as possible:

- If the [`build.minify`](#build-minify) option is `'terser'` and the installed Terser version is below 5.16.0, `'esnext'` will be forced down to `'es2021'`.
- In other cases, it will perform no transpilation at all.

The transform is performed with esbuild and the value should be a valid [esbuild target option](https://esbuild.github.io/api/#target). Custom targets can either be an ES version (e.g. `es2015`), a browser with version (e.g. `chrome58`), or an array of multiple target strings.

Note the build will fail if the code contains features that cannot be safely transpiled by esbuild. See [esbuild docs](https://esbuild.github.io/content-types/#javascript) for more details.

## build.modulePreload

- **Type:** `boolean | { polyfill?: boolean, resolveDependencies?: ResolveModulePreloadDependenciesFn }`
- **Default:** `{ polyfill: true }`

By default, a [module preload polyfill](https://guybedford.com/es-module-preloading-integrity#modulepreload-polyfill) is automatically injected. The polyfill is auto injected into the proxy module of each `index.html` entry. If the build is configured to use a non-HTML custom entry via `build.rollupOptions.input`, then it is necessary to manually import the polyfill in your custom entry:

```js
import 'vite/modulepreload-polyfill'
```

Note: the polyfill does **not** apply to [Library Mode](/guide/build#library-mode). If you need to support browsers without native dynamic import, you should probably avoid using it in your library.

The polyfill can be disabled using `{ polyfill: false }`.

The list of chunks to preload for each dynamic import is computed by Vite. By default, an absolute path including the `base` will be used when loading these dependencies. If the `base` is relative (`''` or `'./'`), `import.meta.url` is used at runtime to avoid absolute paths that depend on the final deployed base.

There is experimental support for fine grained control over the dependencies list and their paths using the `resolveDependencies` function. [Give Feedback](https://github.com/vitejs/vite/discussions/13841). It expects a function of type `ResolveModulePreloadDependenciesFn`:

```ts
type ResolveModulePreloadDependenciesFn = (
  url: string,
  deps: string[],
  context: {
    hostId: string
    hostType: 'html' | 'js'
  },
) => string[]
```

The `resolveDependencies` function will be called for each dynamic import with a list of the chunks it depends on, and it will also be called for each chunk imported in entry HTML files. A new dependencies array can be returned with these filtered or more dependencies injected, and their paths modified. The `deps` paths are relative to the `build.outDir`. The return value should be a relative path to the `build.outDir`.

```js twoslash
/** @type {import('vite').UserConfig} */
const config = {
  // prettier-ignore
  build: {
// ---cut-before---
modulePreload: {
  resolveDependencies: (filename, deps, { hostId, hostType }) => {
    return deps.filter(condition)
  },
},
// ---cut-after---
  },
}
```

```

The resolved dependency paths can be further modified using [`experimental.renderBuiltUrl`](../guide/build.md#advanced-base-options).

## build.polyfillModulePreload

- **Type:** `boolean`
- **Default:** `true`
- **Deprecated** use `build.modulePreload.polyfill` instead

Whether to automatically inject a [module preload polyfill](https://guybedford.com/es-module-preloading-integrity#modulepreload-polyfill).

## build.outDir

- **Type:** `string`
- **Default:** `dist`

Specify the output directory (relative to [project root](/guide/#index-html-and-project-root)).

## build.assetsDir

- **Type:** `string`
- **Default:** `assets`

Specify the directory to nest generated assets under (relative to `build.outDir`. This is not used in [Library Mode](/guide/build#library-mode)).

## build.assetsInlineLimit

- **Type:** `number` | `((filePath: string, content: Buffer) => boolean | undefined)`
- **Default:** `4096` (4 KiB)

Imported or referenced assets that are smaller than this threshold will be inlined as base64 URLs to avoid extra http requests. Set to `0` to disable inlining altogether.

If a callback is passed, a boolean can be returned to opt-in or opt-out. If nothing is returned the default logic applies.

Git LFS placeholders are automatically excluded from inlining because they do not contain the content of the file they represent.

::: tip Note
If you specify `build.lib`, `build.assetsInlineLimit` will be ignored and assets will always be inlined, regardless of file size or being a Git LFS placeholder.
:::

## build.cssCodeSplit

- **Type:** `boolean`
- **Default:** `true`

Enable/disable CSS code splitting. When enabled, CSS imported in async JS chunks will be preserved as chunks and fetched together when the chunk is fetched.

If disabled, all CSS in the entire project will be extracted into a single CSS file.

::: tip Note
If you specify `build.lib`, `build.cssCodeSplit` will be `false` as default.
:::

## build.cssTarget

- **Type:** `string | string[]`
- **Default:** the same as [`build.target`](#build-target)

This option allows users to set a different browser target for CSS minification from the one used for JavaScript transpilation.

It should only be used when you are targeting a non-mainstream browser.
One example is Android WeChat WebView, which supports most modern JavaScript features but not the [`#RGBA` hexadecimal color notation in CSS](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value#rgb_colors).
In this case, you need to set `build.cssTarget` to `chrome61` to prevent vite from transform `rgba()` colors into `#RGBA` hexadecimal notations.

## build.cssMinify

- **Type:** `boolean | 'esbuild' | 'lightningcss'`
- **Default:** the same as [`build.minify`](#build-minify)

This option allows users to override CSS minification specifically instead of defaulting to `build.minify`, so you can configure minification for JS and CSS separately. Vite uses `esbuild` by default to minify CSS. Set the option to `'lightningcss'` to use [Lightning CSS](https://lightningcss.dev/minification.html) instead. If selected, it can be
```

configured using [`css.lightningcss`](./shared-options.md#css-lightningcss).

## build.sourcemap

- **Type:** `boolean | 'inline' | 'hidden'`
- **Default:** `false`

Generate production source maps. If `true`, a separate sourcemap file will be created. If `'inline'`, the sourcemap will be appended to the resulting output file as a data URI. `'hidden'` works like `true` except that the corresponding sourcemap comments in the bundled files are suppressed.

## build.rollupOptions

- **Type:** [`RollupOptions`](https://rollupjs.org/configuration-options/)

Directly customize the underlying Rollup bundle. This is the same as options that can be exported from a Rollup config file and will be merged with Vite's internal Rollup options. See [Rollup options docs](https://rollupjs.org/configuration-options/) for more details.

## build.commonjsOptions

- **Type:** [`RollupCommonJSOptions`](https://github.com/rollup/plugins/tree/master/packages/commonjs#options)

Options to pass on to [@rollup/plugin-commonjs](https://github.com/rollup/plugins/tree/master/packages/commonjs).

## build.dynamicImportVarsOptions

- **Type:** [`RollupDynamicImportVarsOptions`](https://github.com/rollup/plugins/tree/master/packages/dynamic-import-vars#options)
- **Related:** [Dynamic Import](/guide/features#dynamic-import)

Options to pass on to [@rollup/plugin-dynamic-import-vars](https://github.com/rollup/plugins/tree/master/packages/dynamic-import-vars).

## build.lib

- **Type:** `{ entry: string | string[] | { [entryAlias: string]: string }, name?: string, formats?: ('es' | 'cjs' | 'umd' | 'iife')[], fileName?: string | ((format: ModuleFormat, entryName: string) => string) }`
- **Related:** [Library Mode](/guide/build#library-mode)

Build as a library. `entry` is required since the library cannot use HTML as entry. `name` is the exposed global variable and is required when `formats` includes `'umd'` or `'iife'`. Default `formats` are `['es', 'umd']`, or `['es', 'cjs']`, if multiple entries are used. `fileName` is the name of the package file output, default `fileName` is the name option of package.json, it can also be defined as function taking the `format` and `entryAlias` as arguments.

## build.manifest

- **Type:** `boolean | string`
- **Default:** `false`
- **Related:** [Backend Integration](/guide/backend-integration)

When set to `true`, the build will also generate a `.vite/manifest.json` file that contains a mapping of non-hashed asset filenames to their hashed versions, which can then be used by a server framework to render the correct asset links. When the value is a string, it will be used as the manifest file name.

## build.ssrManifest

- **Type:** `boolean | string`
- **Default:** `false`
- **Related:** [Server-Side Rendering](/guide/ssr)

When set to `true`, the build will also generate an SSR manifest for determining style links and asset preload directives in production. When the value is a string, it will be used as the manifest file name.

## build.ssr

- **Type:** `boolean | string`
- **Default:** `false`
- **Related:** [Server-Side Rendering](/guide/ssr)

Produce SSR-oriented build. The value can be a string to directly specify the SSR entry, or `true`, which requires specifying the SSR entry via `rollupOptions.input`.

## build.ssrEmitAssets

- **Type:** `boolean`
- **Default:** `false`

During the SSR build, static assets aren't emitted as it is assumed they would be emitted as part of the client build. This option allows frameworks to force emitting them in both the client and SSR build. It is responsibility of the framework to merge the assets with a post build step.

## build.minify

- **Type:** `boolean | 'terser' | 'esbuild'`
- **Default:** `'esbuild'` for client build, `false` for SSR build

Set to `false` to disable minification, or specify the minifier to use. The default is [esbuild](https://github.com/evanw/esbuild) which is 20 ~ 40x faster than terser and only 1 ~ 2% worse compression. [Benchmarks](https://github.com/privatenumber/minification-benchmarks)

Note the `build.minify` option does not minify whitespaces when using the `'es'` format in lib mode, as it removes pure annotations and breaks tree-shaking.

Terser must be installed when it is set to `'terser'`.

```sh
npm add -D terser
```

## build.terserOptions

- **Type:** `TerserOptions`

Additional [minify options](https://terser.org/docs/api-reference#minify-options) to pass on to Terser.

In addition, you can also pass a `maxWorkers: number` option to specify the max number of workers to spawn. Defaults to the number of CPUs minus 1.

## build.write

- **Type:** `boolean`
- **Default:** `true`

Set to `false` to disable writing the bundle to disk. This is mostly used in [programmatic `build()` calls](/guide/api-javascript#build) where further post processing of the bundle is needed before writing to disk.

## build.emptyOutDir

- **Type:** `boolean`
- **Default:** `true` if `outDir` is inside `root`

By default, Vite will empty the `outDir` on build if it is inside project root. It will emit a warning if `outDir` is outside of root to avoid accidentally removing important files. You can explicitly set this option to suppress the warning. This is also available via command line as `--emptyOutDir`.

## build.copyPublicDir

- **Type:** `boolean`
- **Default:** `true`

By default, Vite will copy files from the `publicDir` into the `outDir` on build. Set to `false` to disable this.

## build.reportCompressedSize

- **Type:** `boolean`
- **Default:** `true`

Enable/disable gzip-compressed size reporting. Compressing large output files can be slow, so disabling this may increase build performance for large projects.

## build.chunkSizeWarningLimit

- **Type:** `number`
- **Default:** `500`

Limit for chunk size warnings (in kB). It is compared against the uncompressed chunk size as the [JavaScript size itself is related to the execution time](https://v8.dev/blog/cost-of-javascript-2019).

## build.watch

- **Type:** [`WatcherOptions`](https://rollupjs.org/configuration-options/#watch)`| null`
- **Default:** `null`

Set to `{}` to enable rollup watcher. This is mostly used in cases that involve build-only plugins or integrations processes.

::: warning Using Vite on Windows Subsystem for Linux (WSL) 2

There are cases that file system watching does not work with WSL2.
See [`server.watch`](./server-options.md#server-watch) for more details.

:::

```
---
title: Vite 3.0 is out!
author:
  name: The Vite Team
date: 2022-07-23
sidebar: false
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 3
  - - meta
    - property: og:image
      content: https://vitejs.dev/og-image-announcing-vite3.png
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite3
  - - meta
    - property: og:description
      content: Vite 3 Release Announcement
  - - meta
    - name: twitter:card
      content: summary_large_image
---

# Vite 3.0 is out!

_July 23, 2022_ - Check out the [Vite 4.0 announcement](./announcing-vite4.md)

In February last year, [Evan You](https://twitter.com/youyuxi) released Vite 2. Since then, its adoption has grown non-stop,
reaching more than 1 million npm downloads per week. A sprawling ecosystem rapidly formed after the release. Vite is powering
a renewed innovation race in Web frameworks. [Nuxt 3](https://v3.nuxtjs.org/) uses Vite by default. [SvelteKit]
(https://kit.svelte.dev/), [Astro](https://astro.build/), [Hydrogen](https://hydrogen.shopify.dev/), and [SolidStart]
(https://docs.solidjs.com/quick-start) are all built with Vite. [Laravel has now decided to use Vite by default]
(https://laravel.com/docs/9.x/vite). [Vite Ruby](https://vite-ruby.netlify.app/) shows how Vite can improve Rails DX.
[Vitest](https://vitest.dev) is making strides as a Vite-native alternative to Jest. Vite is behind [Cypress]
(https://docs.cypress.io/guides/component-testing/writing-your-first-component-test) and [Playwright]
(https://playwright.dev/docs/test-components)'s new Component Testing features, Storybook has [Vite as an official builder]
(https://github.com/storybookjs/builder-vite). And [the list goes on](https://patak.dev/vite/ecosystem.html). Maintainers
from most of these projects got involved in improving the Vite core itself, working closely with the Vite [team]
(https://vitejs.dev/team) and other contributors.

![Vite 3 Announcement Cover Image](/og-image-announcing-vite3.png)

Today, 16 months from the v2 launch we are happy to announce the release of Vite 3. We decided to release a new Vite major at
least every year to align with [Node.js's EOL](https://nodejs.org/en/about/releases/), and take the opportunity to review
Vite's API regularly with a short migration path for projects in the ecosystem.

Quick links:

- [Docs](/)
- [Migration Guide](https://v3.vitejs.dev/guide/migration.html)
- [Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#300-2022-07-13)

If you are new to Vite, we recommend reading the [Why Vite Guide](https://vitejs.dev/guide/why.html). Then check out [the
Getting Started](https://vitejs.dev/guide/) and [Features guide](https://vitejs.dev/guide/features) to see what Vite provides
out of the box. As usual, contributions are welcome at [GitHub](https://github.com/vitejs/vite). More than [600
collaborators](https://github.com/vitejs/vite/graphs/contributors) have helped improve Vite so far. Follow the updates on
[Twitter](https://twitter.com/vite_js), or join discussions with other Vite users on our [Discord chat server]
(http://chat.vitejs.dev/).

## New Documentation

Go to [vitejs.dev](https://vitejs.dev) to enjoy the new v3 docs. Vite is now using the new [VitePress]
(https://vitepress.vuejs.org) default theme, with a stunning dark mode between other features.

[![Vite documentation frontpage](../images/v3-docs.png)](https://vitejs.dev)

Several projects in the ecosystem have already migrated to it (see [Vitest](https://vitest.dev), [vite-plugin-pwa]
(https://vite-plugin-pwa.netlify.app/), and [VitePress](https://vitepress.vuejs.org/) itself).

If you need to access the Vite 2 docs, they will remain online at [v2.vitejs.dev](https://v2.vitejs.dev). There is also a new
[main.vitejs.dev](https://main.vitejs.dev) subdomain, where each commit to Viteâ€™s main branch is auto deployed. This is
useful when testing beta versions or contributing to the coreâ€™s development.

There is also now an official Spanish translation, that has been added to the previous Chinese and Japanese translations:

- [ç®€ä½“ä¸æ–‡](https://cn.vitejs.dev/)
- [æ—¥æœ¬èªž](https://ja.vitejs.dev/)
- [EspaÃ±ol](https://es.vitejs.dev/)
```

## Create Vite Starter Templates

[create-vite](/guide/#trying-vite-online) templates have been a great tool to quickly test Vite with your favorite framework. In Vite 3, all of the templates got a new theme in line with the new docs. Open them online and start playing with Vite 3 now:

<div class="stackblitz-links">
<a target="_blank" href="https://vite.new"><img width="75" height="75" src="../images/vite.svg" alt="Vite logo"></a>
<a target="_blank" href="https://vite.new/vue"><img width="75" height="75" src="../images/vue.svg" alt="Vue logo"></a>
<a target="_blank" href="https://vite.new/svelte"><img width="60" height="60" src="../images/svelte.svg" alt="Svelte logo"></a>
<a target="_blank" href="https://vite.new/react"><img width="75" height="75" src="../images/react.svg" alt="React logo"></a>
<a target="_blank" href="https://vite.new/preact"><img width="65" height="65" src="../images/preact.svg" alt="Preact logo"></a>
<a target="_blank" href="https://vite.new/lit"><img width="60" height="60" src="../images/lit.svg" alt="Lit logo"></a>
</div>

<style>
.stackblitz-links {
  display: flex;
  width: 100%;
  justify-content: space-around;
  align-items: center;
}
@media screen and (max-width: 550px) {
  .stackblitz-links {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
    width: 100%;
    gap: 2rem;
    padding-left: 3rem;
    padding-right: 3rem;
  }
}
.stackblitz-links > a {
  width: 70px;
  height: 70px;
  display: grid;
  align-items: center;
  justify-items: center;
}
.stackblitz-links > a:hover {
  filter: drop-shadow(0 0 0.5em #646cffaa);
}
</style>

The theme is now shared by all templates. This should help better convey the scope for these starters as minimal templates to get started with Vite. For more complete solutions including linting, testing setup, and other features, there are official Vite-powered templates for some frameworks like [create-vue](https://github.com/vuejs/create-vue) and [create-svelte](https://github.com/sveltejs/kit). There is a community-maintained list of templates at [Awesome Vite](https://github.com/vitejs/awesome-vite#templates).

## Dev Improvements

### Vite CLI

<pre style="background-color: var(--vp-code-block-bg);padding:2em;border-radius:8px;max-width:100%;overflow-x:auto;">
  <span style="color:lightgreen"><b>VITE</b></span> <span style="color:lightgreen">v3.0.0</span> <span style="color:gray">ready in <b>320</b> ms</span>

  <span style="color:lightgreen"><b>âžœ</b></span> <span style="color:white"><b>Local</b>:</span>   <span style="color:cyan">http://127.0.0.1:5173/</span>
  <span style="color:green"><b>âžœ</b></span> <span style="color:gray"><b>Network</b>: use --host to expose</span>
</pre>

Apart from the CLIâ€™s aesthetics improvements, youâ€™ll notice that the default dev server port is now 5173 and the preview server listening at 4173. This change ensures Vite will avoid collisions with other tools.

### Improved WebSocket Connection Strategy

One of the pain points of Vite 2 was configuring the server when running behind a proxy. Vite 3 changes the default connection scheme so it works out of the box in most scenarios. All these setups are now tested as part of the Vite Ecosystem CI through [`vite-setup-catalogue`](https://github.com/sapphi-red/vite-setup-catalogue).

### Cold Start Improvements

Vite now avoids full reload during cold start when imports are injected by plugins while crawling the initial statically imported modules ([#8869](https://github.com/vitejs/vite/issues/8869)).

<details>
  <summary><b>Click to learn more</b></summary>

In Vite 2.9, both the scanner and optimizer were run in the background. In the best scenario, where the scanner would find every dependency, no reload was needed in cold start. But if the scanner missed a dependency, a new optimization phase and then a reload were needed. Vite was able to avoid some of these reloads in v2.9, as we detected if the new optimized chunks were compatible with the ones the browser had. But if there was a common dep, the sub-chunks could change and a reload was required to avoid duplicated state. In Vite 3, the optimized deps aren't handed to the browser until the crawling of static imports is done. A quick optimization phase is issued if there is a missing dep (for example, injected by a plugin), and only then, the bundled deps are sent. So, a page reload is no longer needed for these cases.

</details>

<img style="background-color: var(--vp-code-block-bg);padding:4%;border-radius:8px;" width="100%" height="auto" src="../images/vite-3-cold-start.svg" alt="Two graphs comparing Vite 2.9 and Vite 3 optimization strategy">

### import.meta.glob

`import.meta.glob` support was rewritten. Read about the new features in the [Glob Import Guide](/guide/features.html#glob-import):

[Multiple Patterns](/guide/features.html#multiple-patterns) can be passed as an array

```js
import.meta.glob(['./dir/*.js', './another/*.js'])
```

[Negative Patterns](/guide/features.html#negative-patterns) are now supported (prefixed with `!`) to ignore some specific files

```js
import.meta.glob(['./dir/*.js', '!**/bar.js'])
```

[Named Imports](/guide/features.html#named-imports) can be specified to improve tree-shaking

```js
import.meta.glob('./dir/*.js', { import: 'setup' })
```

[Custom Queries](/guide/features.html#custom-queries) can be passed to attach metadata

```js
import.meta.glob('./dir/*.js', { query: { custom: 'data' } })
```

[Eager Imports](/guide/features.html#glob-import) is now passed as a flag

```js
import.meta.glob('./dir/*.js', { eager: true })
```

### Aligning WASM Import with Future Standards

The WebAssembly import API has been revised to avoid collisions with future standards and to make it more flexible:

```js
import init from './example.wasm?init'

init().then((instance) => {
  instance.exports.test()
})
```

Learn more in the [WebAssembly guide](/guide/features.html#webassembly)

## Build Improvements

### ESM SSR Build by Default

Most SSR frameworks in the ecosystem were already using ESM builds. So, Vite 3 makes ESM the default format for SSR builds. This allows us to streamline previous [SSR externalization heuristics](https://vitejs.dev/guide/ssr.html#ssr-externals), externalizing dependencies by default.

### Improved Relative Base Support

Vite 3 now properly supports relative base (using `base: ''`), allowing built assets to be deployed to different bases without re-building. This is useful when the base isn't known at build time, for example when deploying to content-addressable networks like [IPFS](https://ipfs.io/).

## Experimental Features

### Built Asset Paths fine-grained Control (Experimental)

There are other deploy scenarios where this isn't enough. For example, if the generated hashed assets need to be deployed to a different CDN from the public files, then finer-grained control is required over path generation at build time. Vite 3 provides an experimental API to modify the built file paths. Check [Build Advanced Base Options](/guide/build.html#advanced-base-options) for more information.

### Esbuild Deps Optimization at Build Time (Experimental)

One of the main differences between dev and build time is how Vite handles dependencies. During build time, [`@rollup/plugin-commonjs`](https://github.com/rollup/plugins/tree/master/packages/commonjs) is used to allow importing CJS only dependencies (like React). When using the dev server, esbuild is used instead to pre-bundle and optimize dependencies, and an inline interop scheme is applied while transforming user code importing CJS deps. During the development of Vite 3, we introduced the changes needed to also allow the use of [esbuild to optimize dependencies during build time](https://v3.vitejs.dev/guide/migration.html#using-esbuild-deps-optimization-at-build-time). [`@rollup/plugin-commonjs`](https://github.com/rollup/plugins/tree/master/packages/commonjs) can then be avoided, making dev and build time work in the same way.

Given that Rollup v3 will be out in the next months, and we're going to follow up with another Vite major, we've decided to make this mode optional to reduce v3 scope and give Vite and the ecosystem more time to work out possible issues with the new CJS interop approach during build time. Frameworks may switch to using esbuild deps optimization during build time by default at their own pace before Vite 4.

### HMR Partial Accept (Experimental)

There is opt-in support for [HMR Partial Accept](https://github.com/vitejs/vite/pull/7324). This feature could unlock finer-grained HMR for framework components that export several bindings in the same module. You can learn more at [the discussion for this proposal](https://github.com/vitejs/vite/discussions/7309).

## Bundle Size Reduction

Vite cares about its publish and install footprint; a fast installation of a new app is a feature. Vite bundles most of its dependencies and tries to use modern lightweight alternatives where possible. Continuing with this ongoing goal, Vite 3 publish size is 30% smaller than v2.

	Publish Size	Install Size
Vite 2.9.14	4.38MB	19.1MB
Vite 3.0.0	3.05MB	17.8MB
Reduction	-30%	-7%

In part, this reduction was possible by making some dependencies that most users weren't needing optional. First, [Terser](https://github.com/terser/terser) is no longer installed by default. This dependency was no longer needed since we already made esbuild the default minifier for both JS and CSS in Vite 2. If you use `build.minify: 'terser'`, you'll need to install it (`npm add -D terser`). We also moved [node-forge](https://github.com/digitalbazaar/forge) out of the monorepo, implementing support for automatic https certificate generation as a new plugin: [`@vitejs/plugin-basic-ssl`](https://v3.vitejs.dev/guide/migration.html#automatic-https-certificate-generation). Since this feature only creates untrusted certificates that are not added to the local store, it didn't justify the added size.

## Bug Fixing

A triaging marathon was spearheaded by [@bluwyoo](https://twitter.com/bluwyoo), [@sapphi_red](https://twitter.com/sapphi_red), that recently joined the Vite team. During the past three months, the Vite open issues were reduced from 770 to 400. And this dive was achieved while the newly open PRs were at an all-time high. At the same time, [@haoqunjiang](https://twitter.com/haoqunjiang) had also curated a comprehensive [overview of Vite issues](https://github.com/vitejs/vite/discussions/8232).

[![Graph of open issues and pull requests in Vite](../images/v3-open-issues-and-PRs.png)](https://www.repotrends.com/vitejs/vite)

[![Graph of new issues and pull requests in Vite](../images/v3-new-open-issues-and-PRs.png)](https://www.repotrends.com/vitejs/vite)

## Compatibility Notes

- Vite no longer supports Node.js 12 / 13 / 15, which reached its EOL. Node.js 14.18+ / 16+ is now required.
- Vite is now published as ESM, with a CJS proxy to the ESM entry for compatibility.
- The Modern Browser Baseline now targets browsers which support the [native ES Modules](https://caniuse.com/es6-module), [native ESM dynamic import](https://caniuse.com/es6-module-dynamic-import), and [`import.meta`](https://caniuse.com/mdn-javascript_operators_import_meta) features.
- JS file extensions in SSR and library mode now use a valid extension (`js`, `mjs`, or `cjs`) for output JS entries and chunks based on their format and the package type.

Learn more in the [Migration Guide](https://v3.vitejs.dev/guide/migration.html).

## Upgrades to Vite Core

While working towards Vite 3, we also improved the contributing experience for collaborators to [Vite Core](https://github.com/vitejs/vite).

- Unit and E2E tests have been migrated to [Vitest](https://vitest.dev), providing a faster and more stable DX. This move also works as dog fooding for an important infrastructure project in the ecosystem.
- VitePress build is now tested as part of CI.
- Vite upgraded to [pnpm 7](https://pnpm.io/), following the rest of the ecosystem.
- Playgrounds have been moved to [`/playgrounds`](https://github.com/vitejs/vite/tree/main/playground) out of packages directory.
- The packages and playgrounds are now `"type": "module"`.
- Plugins are now bundled using [unbuild](https://github.com/unjs/unbuild), and [plugin-vue-jsx]

(https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue-jsx) and [plugin-legacy]
(https://github.com/vitejs/vite/tree/main/packages/plugin-legacy) were moved to TypeScript.

## The Ecosystem is Ready for v3

We have worked closely with projects in the ecosystem to ensure that frameworks powered by Vite are ready for Vite 3. [vite-ecosystem-ci](https://github.com/vitejs/vite-ecosystem-ci) allows us to run the CI's from the leading players in the ecosystem against Vite's main branch and receive timely reports before introducing a regression. Today's release should soon be compatible with most projects using Vite.

## Acknowledgments

Vite 3 is the result of the aggregate effort of members of the [Vite Team](/team) working together with ecosystem project maintainers and other collaborators to Vite core.

We want to thank everyone that have implemented features, and fixes, given feedback, and have been involved in Vite 3:

- Vite team members [@youyuxi](https://twitter.com/youyuxi), [@patak_dev](https://twitter.com/patak_dev), [@antfu7](https://twitter.com/antfu7), [@bluwyoo](https://twitter.com/bluwyoo), [@sapphi_red](https://twitter.com/sapphi_red), [@haoqunjiang](https://twitter.com/haoqunjiang), [@poyoho](https://github.com/poyoho), [@Shini_92](https://twitter.com/Shini_92), and [@retropragma](https://twitter.com/retropragma).
- [@benmccann](https://github.com/benmccann), [@danielcroe](https://twitter.com/danielcroe), [@brillout](https://twitter.com/brillout), [@sheremet_va](https://twitter.com/sheremet_va), [@userquin](https://twitter.com/userquin), [@enzoinnocenzi](https://twitter.com/enzoinnocenzi), [@maximomussini](https://twitter.com/maximomussini), [@IanVanSchooten](https://twitter.com/IanVanSchooten), the [Astro team](https://astro.build/), and all other maintainers of frameworks and plugins in the ecosystem in that helped shape v3.
- [@dominikg](https://github.com/dominikg) for his work on vite-ecosystem-ci.
- [@ZoltanKochan](https://twitter.com/ZoltanKochan) for his work on [pnpm](https://pnpm.io/), and for his responsiveness when we needed support with it.
- [@rixo](https://github.com/rixo) for HMR Partial Accept support.
- [@KiaKing85](https://twitter.com/KiaKing85) for getting the theme ready for the Vite 3 release, and [@_brc_dd](https://twitter.com/_brc_dd) for working on the VitePress internals.
- [@CodingWithCego](https://twitter.com/CodingWithCego) for the new Spanish translation, and [@ShenQingchuan](https://twitter.com/ShenQingchuan), [@hiro-lapis](https://github.com/hiro-lapis) and others in the Chinese and Japanese translations teams for keeping the translated docs up to date.

We also want to thank individuals and companies sponsoring the Vite team, and companies investing in Vite development: some of [@antfu7](https://twitter.com/antfu7)'s work on Vite and the ecosystem is part of his job at [Nuxt Labs](https://nuxtlabs.com/), and [StackBlitz](https://stackblitz.com/) hired [@patak_dev](https://twitter.com/patak_dev) to work full time on Vite.

## What's Next

We'll take the following months to ensure a smooth transition for all the projects built on top of Vite. So the first minors will be focused on continuing our triaging efforts with a focus on newly opened issues.

The Rollup team is [working on its next major](https://twitter.com/lukastaegert/status/1544186847399743488), to be released in the following months. Once the Rollup plugins ecosystem has time to update, we'll follow up with a new Vite major. This will give us another opportunity to introduce more significant changes this year, which we could take to stabilize some of the experimental features introduced in this release.

If you are interested in helping improve Vite, the best way to get on board is to help with triaging issues. Join [our Discord](https://chat.vitejs.dev) and look for the `#contributing` channel. Or get involved in our `#docs`, `#help` others, or create plugins. We are just getting started. There are many open ideas to keep improving Vite's DX.

# HMR API

:::tip Note
This is the client HMR API. For handling HMR update in plugins, see [handleHotUpdate](./api-plugin#handlehotupdate).

The manual HMR API is primarily intended for framework and tooling authors. As an end user, HMR is likely already handled for you in the framework specific starter templates.
:::

Vite exposes its manual HMR API via the special `import.meta.hot` object:

```ts twoslash
import type { ModuleNamespace } from 'vite/types/hot.d.ts'
import type { InferCustomEventPayload } from 'vite/types/customEvent.d.ts'

// ---cut---
interface ImportMeta {
  readonly hot?: ViteHotContext
}

interface ViteHotContext {
  readonly data: any

  accept(): void
  accept(cb: (mod: ModuleNamespace | undefined) => void): void
  accept(dep: string, cb: (mod: ModuleNamespace | undefined) => void): void
  accept(
    deps: readonly string[],
    cb: (mods: Array<ModuleNamespace | undefined>) => void,
  ): void

  dispose(cb: (data: any) => void): void
  prune(cb: (data: any) => void): void
  invalidate(message?: string): void

  on<T extends string>(
    event: T,
    cb: (payload: InferCustomEventPayload<T>) => void,
  ): void
  off<T extends string>(
    event: T,
    cb: (payload: InferCustomEventPayload<T>) => void,
  ): void
  send<T extends string>(event: T, data?: InferCustomEventPayload<T>): void
}
```

## Required Conditional Guard

First of all, make sure to guard all HMR API usage with a conditional block so that the code can be tree-shaken in production:

```js
if (import.meta.hot) {
  // HMR code
}
```

## IntelliSense for TypeScript

Vite provides type definitions for `import.meta.hot` in [`vite/client.d.ts`](https://github.com/vitejs/vite/blob/main/packages/vite/client.d.ts). You can create an `env.d.ts` in the `src` directory so TypeScript picks up the type definitions:

```ts
/// <reference types="vite/client" />
```

## `hot.accept(cb)`

For a module to self-accept, use `import.meta.hot.accept` with a callback which receives the updated module:

```js twoslash
import 'vite/client'
// ---cut---
export const count = 1

if (import.meta.hot) {
  import.meta.hot.accept((newModule) => {
    if (newModule) {
      // newModule is undefined when SyntaxError happened
      console.log('updated: count is now ', newModule.count)
    }
```

~~~markdown
  })
}
```

A module that "accepts" hot updates is considered an **HMR boundary**.

Vite's HMR does not actually swap the originally imported module: if an HMR boundary module re-exports imports from a dep,
then it is responsible for updating those re-exports (and these exports must be using `let`). In addition, importers up the
chain from the boundary module will not be notified of the change. This simplified HMR implementation is sufficient for most
dev use cases, while allowing us to skip the expensive work of generating proxy modules.

Vite requires that the call to this function appears as `import.meta.hot.accept(` (whitespace-sensitive) in the source code
in order for the module to accept update. This is a requirement of the static analysis that Vite does to enable HMR support
for a module.

## `hot.accept(deps, cb)`

A module can also accept updates from direct dependencies without reloading itself:

```js twoslash
// @filename: /foo.d.ts
export declare const foo: () => void

// @filename: /example.js
import 'vite/client'
// ---cut---
import { foo } from './foo.js'

foo()

if (import.meta.hot) {
  import.meta.hot.accept('./foo.js', (newFoo) => {
    // the callback receives the updated './foo.js' module
    newFoo?.foo()
  })

  // Can also accept an array of dep modules:
  import.meta.hot.accept(
    ['./foo.js', './bar.js'],
    ([newFooModule, newBarModule]) => {
      // The callback receives an array where only the updated module is
      // non null. If the update was not successful (syntax error for ex.),
      // the array is empty
    },
  )
}
```

## `hot.dispose(cb)`

A self-accepting module or a module that expects to be accepted by others can use `hot.dispose` to clean-up any persistent
side effects created by its updated copy:

```js twoslash
import 'vite/client'
// ---cut---
function setupSideEffect() {}

setupSideEffect()

if (import.meta.hot) {
  import.meta.hot.dispose((data) => {
    // cleanup side effect
  })
}
```

## `hot.prune(cb)`

Register a callback that will call when the module is no longer imported on the page. Compared to `hot.dispose`, this can be
used if the source code cleans up side-effects by itself on updates and you only need to clean-up when it's removed from the
page. Vite currently uses this for `.css` imports.

```js twoslash
import 'vite/client'
// ---cut---
function setupOrReuseSideEffect() {}

setupOrReuseSideEffect()

if (import.meta.hot) {
  import.meta.hot.prune((data) => {
    // cleanup side effect
~~~

  })
}

## `hot.data`

The `import.meta.hot.data` object is persisted across different instances of the same updated module. It can be used to pass on information from a previous version of the module to the next one.

Note that re-assignment of `data` itself is not supported. Instead, you should mutate properties of the `data` object so information added from other handlers are preserved.

```js twoslash
import 'vite/client'
// ---cut---
// ok
import.meta.hot.data.someValue = 'hello'

// not supported
import.meta.hot.data = { someValue: 'hello' }
```

## `hot.decline()`

This is currently a noop and is there for backward compatibility. This could change in the future if there is a new usage for it. To indicate that the module is not hot-updatable, use `hot.invalidate()`.

## `hot.invalidate(message?: string)`

A self-accepting module may realize during runtime that it can't handle a HMR update, and so the update needs to be forcefully propagated to importers. By calling `import.meta.hot.invalidate()`, the HMR server will invalidate the importers of the caller, as if the caller wasn't self-accepting. This will log a message both in the browser console and in the terminal. You can pass a message to give some context on why the invalidation happened.

Note that you should always call `import.meta.hot.accept` even if you plan to call `invalidate` immediately afterwards, or else the HMR client won't listen for future changes to the self-accepting module. To communicate your intent clearly, we recommend calling `invalidate` within the `accept` callback like so:

```js twoslash
import 'vite/client'
// ---cut---
import.meta.hot.accept((module) => {
  // You may use the new module instance to decide whether to invalidate.
  if (cannotHandleUpdate(module)) {
    import.meta.hot.invalidate()
  }
})
```

## `hot.on(event, cb)`

Listen to an HMR event.

The following HMR events are dispatched by Vite automatically:

- `'vite:beforeUpdate'` when an update is about to be applied (e.g. a module will be replaced)
- `'vite:afterUpdate'` when an update has just been applied (e.g. a module has been replaced)
- `'vite:beforeFullReload'` when a full reload is about to occur
- `'vite:beforePrune'` when modules that are no longer needed are about to be pruned
- `'vite:invalidate'` when a module is invalidated with `import.meta.hot.invalidate()`
- `'vite:error'` when an error occurs (e.g. syntax error)
- `'vite:ws:disconnect'` when the WebSocket connection is lost
- `'vite:ws:connect'` when the WebSocket connection is (re-)established

Custom HMR events can also be sent from plugins. See [handleHotUpdate](./api-plugin#handlehotupdate) for more details.

## `hot.off(event, cb)`

Remove callback from the event listeners.

## `hot.send(event, data)`

Send custom events back to Vite's dev server.

If called before connected, the data will be buffered and sent once the connection is established.

See [Client-server Communication](/guide/api-plugin.html#client-server-communication) for more details, including a section on [Typing Custom Events](/guide/api-plugin.html#typescript-for-custom-events).

## Further Reading

If you'd like to learn more about how to use the HMR API and how it works under-the-hood. Check out these resources:

- [Hot Module Replacement is Easy](https://bjornlu.com/blog/hot-module-replacement-is-easy)

# Preview Options

## preview.host

- **Type:** `string | boolean`
- **Default:** [`server.host`](./server-options#server-host)

Specify which IP addresses the server should listen on.
Set this to `0.0.0.0` or `true` to listen on all addresses, including LAN and public addresses.

This can be set via the CLI using `--host 0.0.0.0` or `--host`.

::: tip NOTE

There are cases when other servers might respond instead of Vite.
See [`server.host`](./server-options#server-host) for more details.

:::

## preview.port

- **Type:** `number`
- **Default:** `4173`

Specify server port. Note if the port is already being used, Vite will automatically try the next available port so this may not be the actual port the server ends up listening on.

Example:

```js
export default defineConfig({
  server: {
    port: 3030,
  },
  preview: {
    port: 8080,
  },
})
```

## preview.strictPort

- **Type:** `boolean`
- **Default:** [`server.strictPort`](./server-options#server-strictport)

Set to `true` to exit if port is already in use, instead of automatically trying the next available port.

## preview.https

- **Type:** `https.ServerOptions`
- **Default:** [`server.https`](./server-options#server-https)

Enable TLS + HTTP/2. Note this downgrades to TLS only when the [`server.proxy` option](./server-options#server-proxy) is also used.

The value can also be an [options object](https://nodejs.org/api/https.html#https_https_createserver_options_requestlistener) passed to `https.createServer()`.

## preview.open

- **Type:** `boolean | string`
- **Default:** [`server.open`](./server-options#server-open)

Automatically open the app in the browser on server start. When the value is a string, it will be used as the URL's pathname. If you want to open the server in a specific browser you like, you can set the env `process.env.BROWSER` (e.g. `firefox`). You can also set `process.env.BROWSER_ARGS` to pass additional arguments (e.g. `--incognito`).

`BROWSER` and `BROWSER_ARGS` are also special environment variables you can set in the `.env` file to configure it. See [the `open` package](https://github.com/sindresorhus/open#app) for more details.

## preview.proxy

- **Type:** `Record<string, string | ProxyOptions>`
- **Default:** [`server.proxy`](./server-options#server-proxy)

Configure custom proxy rules for the preview server. Expects an object of `{ key: options }` pairs. If the key starts with `^`, it will be interpreted as a `RegExp`. The `configure` option can be used to access the proxy instance.

Uses [`http-proxy`](https://github.com/http-party/node-http-proxy). Full options [here](https://github.com/http-party/node-http-proxy#options).

## preview.cors

- **Type:** `boolean | CorsOptions`
- **Default:** [`server.cors`](./server-options#server-cors)

Configure CORS for the preview server. This is enabled by default and allows any origin. Pass an [options object]
(https://github.com/expressjs/cors#configuration-options) to fine tune the behavior or `false` to disable.

## preview.headers

- **Type:** `OutgoingHttpHeaders`

Specify server response headers.

# JavaScript API

Vite's JavaScript APIs are fully typed, and it's recommended to use TypeScript or enable JS type checking in VS Code to leverage the intellisense and validation.

## `createServer`

Type Signature:

```ts
async function createServer(inlineConfig?: InlineConfig): Promise<ViteDevServer>
```

Example Usage:

```ts twoslash
import { fileURLToPath } from 'node:url'
import { createServer } from 'vite'

const __dirname = fileURLToPath(new URL('.', import.meta.url))

const server = await createServer({
  // any valid user config options, plus `mode` and `configFile`
  configFile: false,
  root: __dirname,
  server: {
    port: 1337,
  },
})
await server.listen()

server.printUrls()
server.bindCLIShortcuts({ print: true })
```

::: tip NOTE
When using `createServer` and `build` in the same Node.js process, both functions rely on `process.env.NODE_ENV` to work properly, which also depends on the `mode` config option. To prevent conflicting behavior, set `process.env.NODE_ENV` or the `mode` of the two APIs to `development`. Otherwise, you can spawn a child process to run the APIs separately.
:::

::: tip NOTE
When using [middleware mode](/config/server-options.html#server-middlewaremode) combined with [proxy config for WebSocket](/config/server-options.html#server-proxy), the parent http server should be provided in `middlewareMode` to bind the proxy correctly.

<details>
<summary>Example</summary>

```ts twoslash
import http from 'http'
import { createServer } from 'vite'

const parentServer = http.createServer() // or express, koa, etc.

const vite = await createServer({
  server: {
    // Enable middleware mode
    middlewareMode: {
      // Provide the parent http server for proxy WebSocket
      server: parentServer,
    },
    proxy: {
      '/ws': {
        target: 'ws://localhost:3000',
        // Proxying WebSocket
        ws: true,
      },
    },
  },
})

// @noErrors: 2339
parentServer.use(vite.middlewares)
```

</details>
:::

## `InlineConfig`

The `InlineConfig` interface extends `UserConfig` with additional properties:

- `configFile`: specify config file to use. If not set, Vite will try to automatically resolve one from project root. Set to `false` to disable auto resolving.
- `envFile`: Set to `false` to disable `.env` files.

## `ResolvedConfig`

The `ResolvedConfig` interface has all the same properties of a `UserConfig`, except most properties are resolved and non-undefined. It also contains utilities like:

- `config.assetsInclude`: A function to check if an `id` is considered an asset.
- `config.logger`: Vite's internal logger object.

## `ViteDevServer`

```ts
interface ViteDevServer {
  /**
   * The resolved Vite config object.
   */
  config: ResolvedConfig
  /**
   * A connect app instance
   * - Can be used to attach custom middlewares to the dev server.
   * - Can also be used as the handler function of a custom http server
   *   or as a middleware in any connect-style Node.js frameworks.
   *
   * https://github.com/senchalabs/connect#use-middleware
   */
  middlewares: Connect.Server
  /**
   * Native Node http server instance.
   * Will be null in middleware mode.
   */
  httpServer: http.Server | null
  /**
   * Chokidar watcher instance. If `config.server.watch` is set to `null`,
   * returns a noop event emitter.
   * https://github.com/paulmillr/chokidar#api
   */
  watcher: FSWatcher
  /**
   * Web socket server with `send(payload)` method.
   */
  ws: WebSocketServer
  /**
   * Rollup plugin container that can run plugin hooks on a given file.
   */
  pluginContainer: PluginContainer
  /**
   * Module graph that tracks the import relationships, url to file mapping
   * and hmr state.
   */
  moduleGraph: ModuleGraph
  /**
   * The resolved urls Vite prints on the CLI. null in middleware mode or
   * before `server.listen` is called.
   */
  resolvedUrls: ResolvedServerUrls | null
  /**
   * Programmatically resolve, load and transform a URL and get the result
   * without going through the http request pipeline.
   */
  transformRequest(
    url: string,
    options?: TransformOptions,
  ): Promise<TransformResult | null>
  /**
   * Apply Vite built-in HTML transforms and any plugin HTML transforms.
   */
  transformIndexHtml(
    url: string,
    html: string,
    originalUrl?: string,
  ): Promise<string>
  /**
   * Load a given URL as an instantiated module for SSR.
   */
  ssrLoadModule(
    url: string,
    options?: { fixStacktrace?: boolean },
  ): Promise<Record<string, any>>
  /**
   * Fix ssr error stacktrace.
```

~~~markdown
   */
  ssrFixStacktrace(e: Error): void
  /**
   * Triggers HMR for a module in the module graph. You can use the `server.moduleGraph`
   * API to retrieve the module to be reloaded. If `hmr` is false, this is a no-op.
   */
  reloadModule(module: ModuleNode): Promise<void>
  /**
   * Start the server.
   */
  listen(port?: number, isRestart?: boolean): Promise<ViteDevServer>
  /**
   * Restart the server.
   *
   * @param forceOptimize - force the optimizer to re-bundle, same as --force cli flag
   */
  restart(forceOptimize?: boolean): Promise<void>
  /**
   * Stop the server.
   */
  close(): Promise<void>
  /**
   * Bind CLI shortcuts
   */
  bindCLIShortcuts(options?: BindCLIShortcutsOptions<ViteDevServer>): void
  /**
   * Calling `await server.waitForRequestsIdle(id)` will wait until all static imports
   * are processed. If called from a load or transform plugin hook, the id needs to be
   * passed as a parameter to avoid deadlocks. Calling this function after the first
   * static imports section of the module graph has been processed will resolve immediately.
   * @experimental
   */
  waitForRequestsIdle: (ignoredId?: string) => Promise<void>
}
```

:::info
`waitForRequestsIdle` is meant to be used as a escape hatch to improve DX for features that can't be implemented following
the on-demand nature of the Vite dev server. It can be used during startup by tools like Tailwind to delay generating the app
CSS classes until the app code has been seen, avoiding flashes of style changes. When this function is used in a load or
transform hook, and the default HTTP1 server is used, one of the six http channels will be blocked until the server processes
all static imports. Vite's dependency optimizer currently uses this function to avoid full-page reloads on missing
dependencies by delaying loading of pre-bundled dependencies until all imported dependencies have been collected from static
imported sources. Vite may switch to a different strategy in a future major release, setting
`optimizeDeps.crawlUntilStaticImports: false` by default to avoid the performance hit in large applications during cold
start.
:::

## `build`

**Type Signature:**

```ts
async function build(
  inlineConfig?: InlineConfig,
): Promise<RollupOutput | RollupOutput[]>
```

**Example Usage:**

```ts twoslash
import path from 'node:path'
import { fileURLToPath } from 'node:url'
import { build } from 'vite'

const __dirname = fileURLToPath(new URL('.', import.meta.url))

await build({
  root: path.resolve(__dirname, './project'),
  base: '/foo/',
  build: {
    rollupOptions: {
      // ...
    },
  },
})
```

## `preview`

**Type Signature:**

```ts
~~~

~~~
async function preview(inlineConfig?: InlineConfig): Promise<PreviewServer>
```

**Example Usage:**

```ts twoslash
import { preview } from 'vite'

const previewServer = await preview({
  // any valid user config options, plus `mode` and `configFile`
  preview: {
    port: 8080,
    open: true,
  },
})

previewServer.printUrls()
previewServer.bindCLIShortcuts({ print: true })
```

## `PreviewServer`

```ts
interface PreviewServer {
  /**
   * The resolved vite config object
   */
  config: ResolvedConfig
  /**
   * A connect app instance.
   * - Can be used to attach custom middlewares to the preview server.
   * - Can also be used as the handler function of a custom http server
   *   or as a middleware in any connect-style Node.js frameworks
   *
   * https://github.com/senchalabs/connect#use-middleware
   */
  middlewares: Connect.Server
  /**
   * native Node http server instance
   */
  httpServer: http.Server
  /**
   * The resolved urls Vite prints on the CLI.
   * null before server is listening.
   */
  resolvedUrls: ResolvedServerUrls | null
  /**
   * Print server urls
   */
  printUrls(): void
  /**
   * Bind CLI shortcuts
   */
  bindCLIShortcuts(options?: BindCLIShortcutsOptions<PreviewServer>): void
}
```

## `resolveConfig`

**Type Signature:**

```ts
async function resolveConfig(
  inlineConfig: InlineConfig,
  command: 'build' | 'serve',
  defaultMode = 'development',
  defaultNodeEnv = 'development',
  isPreview = false,
): Promise<ResolvedConfig>
```

The `command` value is `serve` in dev and preview, and `build` in build.

## `mergeConfig`

**Type Signature:**

```ts
function mergeConfig(
  defaults: Record<string, any>,
  overrides: Record<string, any>,
  isRoot = true,
): Record<string, any>
~~~

Deeply merge two Vite configs. `isRoot` represents the level within the Vite config which is being merged. For example, set `false` if you're merging two `build` options.

::: tip NOTE
`mergeConfig` accepts only config in object form. If you have a config in callback form, you should call it before passing into `mergeConfig`.

You can use the `defineConfig` helper to merge a config in callback form with another config:

```ts twoslash
import {
  defineConfig,
  mergeConfig,
  type UserConfigFnObject,
  type UserConfig,
} from 'vite'
declare const configAsCallback: UserConfigFnObject
declare const configAsObject: UserConfig

// ---cut---
export default defineConfig((configEnv) =>
  mergeConfig(configAsCallback(configEnv), configAsObject),
)
```

:::

## `searchForWorkspaceRoot`

Type Signature:

```ts
function searchForWorkspaceRoot(
  current: string,
  root = searchForPackageRoot(current),
): string
```

Related: [server.fs.allow](/config/server-options.md#server-fs-allow)

Search for the root of the potential workspace if it meets the following conditions, otherwise it would fallback to `root`:

- contains `workspaces` field in `package.json`
- contains one of the following file
  - `lerna.json`
  - `pnpm-workspace.yaml`

## `loadEnv`

Type Signature:

```ts
function loadEnv(
  mode: string,
  envDir: string,
  prefixes: string | string[] = 'VITE_',
): Record<string, string>
```

Related: [`.env` Files](./env-and-mode.md#env-files)

Load `.env` files within the `envDir`. By default, only env variables prefixed with `VITE_` are loaded, unless `prefixes` is changed.

## `normalizePath`

Type Signature:

```ts
function normalizePath(id: string): string
```

Related: [Path Normalization](./api-plugin.md#path-normalization)

Normalizes a path to interoperate between Vite plugins.

## `transformWithEsbuild`

Type Signature:

```ts
```

```ts
async function transformWithEsbuild(
  code: string,
  filename: string,
  options?: EsbuildTransformOptions,
  inMap?: object,
): Promise<ESBuildTransformResult>
```

Transform JavaScript or TypeScript with esbuild. Useful for plugins that prefer matching Vite's internal esbuild transform.

## `loadConfigFromFile`

Type Signature:

```ts
async function loadConfigFromFile(
  configEnv: ConfigEnv,
  configFile?: string,
  configRoot: string = process.cwd(),
  logLevel?: LogLevel,
  customLogger?: Logger,
): Promise<{
  path: string
  config: UserConfig
  dependencies: string[]
} | null>
```

Load a Vite config file manually with esbuild.

## `preprocessCSS`

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/13815)

Type Signature:

```ts
async function preprocessCSS(
  code: string,
  filename: string,
  config: ResolvedConfig,
): Promise<PreprocessCSSResult>

interface PreprocessCSSResult {
  code: string
  map?: SourceMapInput
  modules?: Record<string, string>
  deps?: Set<string>
}
```

Pre-processes `.css`, `.scss`, `.sass`, `.less`, `.styl` and `.stylus` files to plain CSS so it can be used in browsers or parsed by other tools. Similar to the [built-in CSS pre-processing support](/guide/features#css-pre-processors), the corresponding pre-processor must be installed if used.

The pre-processor used is inferred from the `filename` extension. If the `filename` ends with `.module.{ext}`, it is inferred as a [CSS module](https://github.com/css-modules/css-modules) and the returned result will include a `modules` object mapping the original class names to the transformed ones.

Note that pre-processing will not resolve URLs in `url()` or `image-set()`.

# Server Options

## server.host

- **Type:** `string | boolean`
- **Default:** `'localhost'`

Specify which IP addresses the server should listen on.
Set this to `0.0.0.0` or `true` to listen on all addresses, including LAN and public addresses.

This can be set via the CLI using `--host 0.0.0.0` or `--host`.

::: tip NOTE

There are cases when other servers might respond instead of Vite.

The first case is when `localhost` is used. Node.js under v17 reorders the result of DNS-resolved addresses by default. When accessing `localhost`, browsers use DNS to resolve the address and that address might differ from the address which Vite is listening to. Vite prints the resolved address when it differs.

You can set [`dns.setDefaultResultOrder('verbatim')`](https://nodejs.org/api/dns.html#dns_dns_setdefaultresultorder_order) to disable the reordering behavior. Vite will then print the address as `localhost`.

```js twoslash
// vite.config.js
import { defineConfig } from 'vite'
import dns from 'node:dns'

dns.setDefaultResultOrder('verbatim')

export default defineConfig({
  // omit
})
```

The second case is when wildcard hosts (e.g. `0.0.0.0`) are used. This is because servers listening on non-wildcard hosts take priority over those listening on wildcard hosts.

:::

::: tip Accessing the server on WSL2 from your LAN

When running Vite on WSL2, it is not sufficient to set `host: true` to access the server from your LAN.
See [the WSL document](https://learn.microsoft.com/en-us/windows/wsl/networking#accessing-a-wsl-2-distribution-from-your-local-area-network-lan) for more details.

:::

## server.port

- **Type:** `number`
- **Default:** `5173`

Specify server port. Note if the port is already being used, Vite will automatically try the next available port so this may not be the actual port the server ends up listening on.

## server.strictPort

- **Type:** `boolean`

Set to `true` to exit if port is already in use, instead of automatically trying the next available port.

## server.https

- **Type:** `https.ServerOptions`

Enable TLS + HTTP/2. Note this downgrades to TLS only when the [`server.proxy` option](#server-proxy) is also used.

The value can also be an [options object](https://nodejs.org/api/https.html#https_https_createserver_options_requestlistener) passed to `https.createServer()`.

A valid certificate is needed. For a basic setup, you can add [@vitejs/plugin-basic-ssl](https://github.com/vitejs/vite-plugin-basic-ssl) to the project plugins, which will automatically create and cache a self-signed certificate. But we recommend creating your own certificates.

## server.open

- **Type:** `boolean | string`

Automatically open the app in the browser on server start. When the value is a string, it will be used as the URL's pathname.
If you want to open the server in a specific browser you like, you can set the env `process.env.BROWSER` (e.g. `firefox`).
You can also set `process.env.BROWSER_ARGS` to pass additional arguments (e.g. `--incognito`).

`BROWSER` and `BROWSER_ARGS` are also special environment variables you can set in the `.env` file to configure it. See [the `open` package](https://github.com/sindresorhus/open#app) for more details.

Example:

```js
export default defineConfig({
  server: {
    open: '/docs/index.html',
  },
})
```

## server.proxy

- **Type:** `Record<string, string | ProxyOptions>`

Configure custom proxy rules for the dev server. Expects an object of `{ key: options }` pairs. Any requests that request path starts with that key will be proxied to that specified target. If the key starts with `^`, it will be interpreted as a `RegExp`. The `configure` option can be used to access the proxy instance.

Note that if you are using non-relative [`base`](/config/shared-options.md#base), you must prefix each key with that `base`.

Extends [`http-proxy`](https://github.com/http-party/node-http-proxy#options). Additional options are [here](https://github.com/vitejs/vite/blob/main/packages/vite/src/node/server/middlewares/proxy.ts#L13).

In some cases, you might also want to configure the underlying dev server (e.g. to add custom middlewares to the internal [connect](https://github.com/senchalabs/connect) app). In order to do that, you need to write your own [plugin](/guide/using-plugins.html) and use [configureServer](/guide/api-plugin.html#configureserver) function.

Example:

```js
export default defineConfig({
  server: {
    proxy: {
      // string shorthand: http://localhost:5173/foo -> http://localhost:4567/foo
      '/foo': 'http://localhost:4567',
      // with options: http://localhost:5173/api/bar-> http://jsonplaceholder.typicode.com/bar
      '/api': {
        target: 'http://jsonplaceholder.typicode.com',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, ''),
      },
      // with RegExp: http://localhost:5173/fallback/ -> http://jsonplaceholder.typicode.com/
      '^/fallback/.*': {
        target: 'http://jsonplaceholder.typicode.com',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/fallback/, ''),
      },
      // Using the proxy instance
      '/api': {
        target: 'http://jsonplaceholder.typicode.com',
        changeOrigin: true,
        configure: (proxy, options) => {
          // proxy will be an instance of 'http-proxy'
        },
      },
      // Proxying websockets or socket.io: ws://localhost:5173/socket.io -> ws://localhost:5174/socket.io
      // Exercise caution using `rewriteWsOrigin` as it can leave the proxying open to CSRF attacks.
      '/socket.io': {
        target: 'ws://localhost:5174',
        ws: true,
        rewriteWsOrigin: true,
      },
    },
  },
})
```

## server.cors

- **Type:** `boolean | CorsOptions`

Configure CORS for the dev server. This is enabled by default and allows any origin. Pass an [options object](https://github.com/expressjs/cors#configuration-options) to fine tune the behavior or `false` to disable.

## server.headers

- **Type:** `OutgoingHttpHeaders`

Specify server response headers.

## server.hmr

- **Type:** `boolean | { protocol?: string, host?: string, port?: number, path?: string, timeout?: number, overlay?: boolean, clientPort?: number, server?: Server }`

Disable or configure HMR connection (in cases where the HMR websocket must use a different address from the http server).

Set `server.hmr.overlay` to `false` to disable the server error overlay.

`protocol` sets the WebSocket protocol used for the HMR connection: `ws` (WebSocket) or `wss` (WebSocket Secure).

`clientPort` is an advanced option that overrides the port only on the client side, allowing you to serve the websocket on a different port than the client code looks for it on.

When `server.hmr.server` is defined, Vite will process the HMR connection requests through the provided server. If not in middleware mode, Vite will attempt to process HMR connection requests through the existing server. This can be helpful when using self-signed certificates or when you want to expose Vite over a network on a single port.

Check out [`vite-setup-catalogue`](https://github.com/sapphi-red/vite-setup-catalogue) for some examples.

::: tip NOTE

With the default configuration, reverse proxies in front of Vite are expected to support proxying WebSocket. If the Vite HMR client fails to connect WebSocket, the client will fall back to connecting the WebSocket directly to the Vite HMR server bypassing the reverse proxies:

```
Direct websocket connection fallback. Check out https://vitejs.dev/config/server-options.html#server-hmr to remove the previous connection error.
```

The error that appears in the Browser when the fallback happens can be ignored. To avoid the error by directly bypassing reverse proxies, you could either:

- configure the reverse proxy to proxy WebSocket too
- set [`server.strictPort = true`](#server-strictport) and set `server.hmr.clientPort` to the same value with `server.port`
- set `server.hmr.port` to a different value from [`server.port`](#server-port)

:::

## server.warmup

- **Type:** `{ clientFiles?: string[], ssrFiles?: string[] }`
- **Related:** [Warm Up Frequently Used Files](/guide/performance.html#warm-up-frequently-used-files)

Warm up files to transform and cache the results in advance. This improves the initial page load during server starts and prevents transform waterfalls.

`clientFiles` are files that are used in the client only, while `ssrFiles` are files that are used in SSR only. They accept an array of file paths or [`fast-glob`](https://github.com/mrmlnc/fast-glob) patterns relative to the `root`.

Make sure to only add files that are frequently used to not overload the Vite dev server on startup.

```js
export default defineConfig({
  server: {
    warmup: {
      clientFiles: ['./src/components/*.vue', './src/utils/big-utils.js'],
      ssrFiles: ['./src/server/modules/*.js'],
    },
  },
})
```

## server.watch

- **Type:** `object | null`

File system watcher options to pass on to [chokidar](https://github.com/paulmillr/chokidar#api).

The Vite server watcher watches the `root` and skips the `.git/`, `node_modules/`, and Vite's `cacheDir` and `build.outDir` directories by default. When updating a watched file, Vite will apply HMR and update the page only if needed.

If set to `null`, no files will be watched. `server.watcher` will provide a compatible event emitter, but calling `add` or `unwatch` will have no effect.

::: warning Watching files in `node_modules`

It's currently not possible to watch files and packages in `node_modules`. For further progress and workarounds, you can follow [issue #8619](https://github.com/vitejs/vite/issues/8619).

:::

::: warning Using Vite on Windows Subsystem for Linux (WSL) 2

When running Vite on WSL2, file system watching does not work when a file is edited by Windows applications (non-WSL2 process). This is due to [a WSL2 limitation](https://github.com/microsoft/WSL/issues/4739). This also applies to running on Docker with a WSL2 backend.

To fix it, you could either:

- **Recommended**: Use WSL2 applications to edit your files.
  - It is also recommended to move the project folder outside of a Windows filesystem. Accessing Windows filesystem from WSL2 is slow. Removing that overhead will improve performance.
- Set `{ usePolling: true }`.
  - Note that [`usePolling` leads to high CPU utilization](https://github.com/paulmillr/chokidar#performance).

:::

## server.middlewareMode

- **Type:** `boolean`
- **Default:** `false`

Create Vite server in middleware mode.

- **Related:** [appType](./shared-options#apptype), [SSR - Setting Up the Dev Server](/guide/ssr#setting-up-the-dev-server)

- **Example:**

```js twoslash
import express from 'express'
import { createServer as createViteServer } from 'vite'

async function createServer() {
  const app = express()

  // Create Vite server in middleware mode
  const vite = await createViteServer({
    server: { middlewareMode: true },
    appType: 'custom', // don't include Vite's default HTML handling middlewares
  })
  // Use vite's connect instance as middleware
  app.use(vite.middlewares)

  app.use('*', async (req, res) => {
    // Since `appType` is `'custom'`, should serve response here.
    // Note: if `appType` is `'spa'` or `'mpa'`, Vite includes middlewares to handle
    // HTML requests and 404s so user middlewares should be added
    // before Vite's middlewares to take effect instead
  })
}

createServer()
```

## server.fs.strict

- **Type:** `boolean`
- **Default:** `true` (enabled by default since Vite 2.7)

Restrict serving files outside of workspace root.

## server.fs.allow

- **Type:** `string[]`

Restrict files that could be served via `/@fs/`. When `server.fs.strict` is set to `true`, accessing files outside this directory list that aren't imported from an allowed file will result in a 403.

Both directories and files can be provided.

Vite will search for the root of the potential workspace and use it as default. A valid workspace met the following conditions, otherwise will fall back to the [project root](/guide/#index-html-and-project-root).

- contains `workspaces` field in `package.json`
- contains one of the following file
  - `lerna.json`
  - `pnpm-workspace.yaml`

Accepts a path to specify the custom workspace root. Could be a absolute path or a path relative to [project root](/guide/#index-html-and-project-root). For example:

```js
export default defineConfig({
  server: {
    fs: {
      // Allow serving files from one level up to the project root
      allow: ['..'],
    },
  },
})
```

When `server.fs.allow` is specified, the auto workspace root detection will be disabled. To extend the original behavior, a utility `searchForWorkspaceRoot` is exposed:

```js
import { defineConfig, searchForWorkspaceRoot } from 'vite'

export default defineConfig({
  server: {
    fs: {
      allow: [
        // search up for workspace root
        searchForWorkspaceRoot(process.cwd()),
        // your custom rules
        '/path/to/custom/allow_directory',
        '/path/to/custom/allow_file.demo',
      ],
    },
  },
})
```

## server.fs.deny

- **Type:** `string[]`
- **Default:** `['.env', '.env.*', '*.{crt,pem}']`

Blocklist for sensitive files being restricted to be served by Vite dev server. This will have higher priority than [`server.fs.allow`](#server-fs-allow). [picomatch patterns](https://github.com/micromatch/picomatch#globbing-features) are supported.

## server.fs.cachedChecks

- **Type:** `boolean`
- **Default:** `false`
- **Experimental**

Caches filenames of accessed directories to avoid repeated filesystem operations. Particularly in Windows, this could result in a performance boost. It is disabled by default due to edge cases when writing a file in a cached folder and immediately importing it.

## server.origin

- **Type:** `string`

Defines the origin of the generated asset URLs during development.

```js
export default defineConfig({
  server: {
    origin: 'http://127.0.0.1:8080',
  },
})
```

## server.sourcemapIgnoreList

- **Type:** `false | (sourcePath: string, sourcemapPath: string) => boolean`
- **Default:** `(sourcePath) => sourcePath.includes('node_modules')`

Whether or not to ignore source files in the server sourcemap, used to populate the [`x_google_ignoreList` source map extension](https://developer.chrome.com/articles/x-google-ignore-list/).

`server.sourcemapIgnoreList` is the equivalent of [`build.rollupOptions.output.sourcemapIgnoreList`](https://rollupjs.org/configuration-options/#output-sourcemapignorelist) for the dev server. A difference between the two config options is that the rollup function is called with a relative path for `sourcePath` while `server.sourcemapIgnoreList` is called with an absolute path. During dev, most modules have the map and the source in the same folder, so the relative path for `sourcePath` is the file name itself. In these cases, absolute paths makes it convenient to be used instead.

By default, it excludes all paths containing `node_modules`. You can pass `false` to disable this behavior, or, for full control, a function that takes the source path and sourcemap path and returns whether to ignore the source path.

```js
export default defineConfig({
  server: {
    // This is the default value, and will add all files with node_modules
```

~~~~
    // in their paths to the ignore list.
    sourcemapIgnoreList(sourcePath, sourcemapPath) {
      return sourcePath.includes('node_modules')
    },
  },
})
```

::: tip Note
[`server.sourcemapIgnoreList`](#server-sourcemapignorelist) and [`build.rollupOptions.output.sourcemapIgnoreList`]
(https://rollupjs.org/configuration-options/#output-sourcemapignorelist) need to be set independently.
`server.sourcemapIgnoreList` is a server only config and doesn't get its default value from the defined rollup options.
:::
~~~~

# Plugin API

Vite plugins extends Rollup's well-designed plugin interface with a few extra Vite-specific options. As a result, you can write a Vite plugin once and have it work for both dev and build.

It is recommended to go through [Rollup's plugin documentation](https://rollupjs.org/plugin-development/) first before reading the sections below.

## Authoring a Plugin

Vite strives to offer established patterns out of the box, so before creating a new plugin make sure that you check the [Features guide](https://vitejs.dev/guide/features) to see if your need is covered. Also review available community plugins, both in the form of a [compatible Rollup plugin](https://github.com/rollup/awesome) and [Vite Specific plugins](https://github.com/vitejs/awesome-vite#plugins)

When creating a plugin, you can inline it in your `vite.config.js`. There is no need to create a new package for it. Once you see that a plugin was useful in your projects, consider sharing it to help others [in the ecosystem](https://chat.vitejs.dev).

::: tip
When learning, debugging, or authoring plugins, we suggest including [vite-plugin-inspect](https://github.com/antfu/vite-plugin-inspect) in your project. It allows you to inspect the intermediate state of Vite plugins. After installing, you can visit `localhost:5173/__inspect/` to inspect the modules and transformation stack of your project. Check out install instructions in the [vite-plugin-inspect docs](https://github.com/antfu/vite-plugin-inspect).
![vite-plugin-inspect](/images/vite-plugin-inspect.png)
:::

## Conventions

If the plugin doesn't use Vite specific hooks and can be implemented as a [Compatible Rollup Plugin](#rollup-plugin-compatibility), then it is recommended to use the [Rollup Plugin naming conventions](https://rollupjs.org/plugin-development/#conventions).

- Rollup Plugins should have a clear name with `rollup-plugin-` prefix.
- Include `rollup-plugin` and `vite-plugin` keywords in package.json.

This exposes the plugin to be also used in pure Rollup or WMR based projects

For Vite only plugins

- Vite Plugins should have a clear name with `vite-plugin-` prefix.
- Include `vite-plugin` keyword in package.json.
- Include a section in the plugin docs detailing why it is a Vite only plugin (for example, it uses Vite specific plugin hooks).

If your plugin is only going to work for a particular framework, its name should be included as part of the prefix

- `vite-plugin-vue-` prefix for Vue Plugins
- `vite-plugin-react-` prefix for React Plugins
- `vite-plugin-svelte-` prefix for Svelte Plugins

See also [Virtual Modules Convention](#virtual-modules-convention).

## Plugins config

Users will add plugins to the project `devDependencies` and configure them using the `plugins` array option.

```js
// vite.config.js
import vitePlugin from 'vite-plugin-feature'
import rollupPlugin from 'rollup-plugin-feature'

export default defineConfig({
  plugins: [vitePlugin(), rollupPlugin()],
})
```

Falsy plugins will be ignored, which can be used to easily activate or deactivate plugins.

`plugins` also accepts presets including several plugins as a single element. This is useful for complex features (like framework integration) that are implemented using several plugins. The array will be flattened internally.

```js
// framework-plugin
import frameworkRefresh from 'vite-plugin-framework-refresh'
import frameworkDevtools from 'vite-plugin-framework-devtools'

export default function framework(config) {
  return [frameworkRefresh(config), frameworkDevTools(config)]
}
```

```js
```

~~~~markdown
// vite.config.js
import { defineConfig } from 'vite'
import framework from 'vite-plugin-framework'

export default defineConfig({
  plugins: [framework()],
})
```

## Simple Examples

:::tip
It is common convention to author a Vite/Rollup plugin as a factory function that returns the actual plugin object. The
function can accept options which allows users to customize the behavior of the plugin.
:::

### Transforming Custom File Types

```js
const fileRegex = /\.(my-file-ext)$/

export default function myPlugin() {
  return {
    name: 'transform-file',

    transform(src, id) {
      if (fileRegex.test(id)) {
        return {
          code: compileFileToJS(src),
          map: null, // provide source map if available
        }
      }
    },
  }
}
```

### Importing a Virtual File

See the example in the [next section](#virtual-modules-convention).

## Virtual Modules Convention

Virtual modules are a useful scheme that allows you to pass build time information to the source files using normal ESM
import syntax.

```js
export default function myPlugin() {
  const virtualModuleId = 'virtual:my-module'
  const resolvedVirtualModuleId = '\0' + virtualModuleId

  return {
    name: 'my-plugin', // required, will show up in warnings and errors
    resolveId(id) {
      if (id === virtualModuleId) {
        return resolvedVirtualModuleId
      }
    },
    load(id) {
      if (id === resolvedVirtualModuleId) {
        return `export const msg = "from virtual module"`
      }
    },
  }
}
```

Which allows importing the module in JavaScript:

```js
import { msg } from 'virtual:my-module'

console.log(msg)
```

Virtual modules in Vite (and Rollup) are prefixed with `virtual:` for the user-facing path by convention. If possible the
plugin name should be used as a namespace to avoid collisions with other plugins in the ecosystem. For example, a `vite-
plugin-posts` could ask users to import a `virtual:posts` or `virtual:posts/helpers` virtual modules to get build time
information. Internally, plugins that use virtual modules should prefix the module ID with `\0` while resolving the id, a
convention from the rollup ecosystem. This prevents other plugins from trying to process the id (like node resolution), and
core features like sourcemaps can use this info to differentiate between virtual modules and regular files. `\0` is not a
permitted char in import URLs so we have to replace them during import analysis. A `\0{id}` virtual id ends up encoded as
`/@id/__x00__{id}` during dev in the browser. The id will be decoded back before entering the plugins pipeline, so this is
~~~~

not seen by plugins hooks code.

Note that modules directly derived from a real file, as in the case of a script module in a Single File Component (like a .vue or .svelte SFC) don't need to follow this convention. SFCs generally generate a set of submodules when processed but the code in these can be mapped back to the filesystem. Using `\0` for these submodules would prevent sourcemaps from working correctly.

## Universal Hooks

During dev, the Vite dev server creates a plugin container that invokes [Rollup Build Hooks](https://rollupjs.org/plugin-development/#build-hooks) the same way Rollup does it.

The following hooks are called once on server start:

- [`options`](https://rollupjs.org/plugin-development/#options)
- [`buildStart`](https://rollupjs.org/plugin-development/#buildstart)

The following hooks are called on each incoming module request:

- [`resolveId`](https://rollupjs.org/plugin-development/#resolveid)
- [`load`](https://rollupjs.org/plugin-development/#load)
- [`transform`](https://rollupjs.org/plugin-development/#transform)

These hooks also have an extended `options` parameter with additional Vite-specific properties. You can read more in the [SSR documentation](/guide/ssr#ssr-specific-plugin-logic).

Some `resolveId` calls' `importer` value may be an absolute path for a generic `index.html` at root as it's not always possible to derive the actual importer due to Vite's unbundled dev server pattern. For imports handled within Vite's resolve pipeline, the importer can be tracked during the import analysis phase, providing the correct `importer` value.

The following hooks are called when the server is closed:

- [`buildEnd`](https://rollupjs.org/plugin-development/#buildend)
- [`closeBundle`](https://rollupjs.org/plugin-development/#closebundle)

Note that the [`moduleParsed`](https://rollupjs.org/plugin-development/#moduleparsed) hook is **not** called during dev, because Vite avoids full AST parses for better performance.

[Output Generation Hooks](https://rollupjs.org/plugin-development/#output-generation-hooks) (except `closeBundle`) are **not** called during dev. You can think of Vite's dev server as only calling `rollup.rollup()` without calling `bundle.generate()`.

## Vite Specific Hooks

Vite plugins can also provide hooks that serve Vite-specific purposes. These hooks are ignored by Rollup.

### `config`

- **Type:** `(config: UserConfig, env: { mode: string, command: string }) => UserConfig | null | void`
- **Kind:** `async`, `sequential`

  Modify Vite config before it's resolved. The hook receives the raw user config (CLI options merged with config file) and the current config env which exposes the `mode` and `command` being used. It can return a partial config object that will be deeply merged into existing config, or directly mutate the config (if the default merging cannot achieve the desired result).

  Example:

  ```js
  // return partial config (recommended)
  const partialConfigPlugin = () => ({
    name: 'return-partial',
    config: () => ({
      resolve: {
        alias: {
          foo: 'bar',
        },
      },
    }),
  })

  // mutate the config directly (use only when merging doesn't work)
  const mutateConfigPlugin = () => ({
    name: 'mutate-config',
    config(config, { command }) {
      if (command === 'build') {
        config.root = 'foo'
      }
    },
  })
  ```

  ::: warning Note
  User plugins are resolved before running this hook so injecting other plugins inside the `config` hook will have no effect.

  :::

### `configResolved`

- **Type:** `(config: ResolvedConfig) => void | Promise<void>`
- **Kind:** `async`, `parallel`

  Called after the Vite config is resolved. Use this hook to read and store the final resolved config. It is also useful when the plugin needs to do something different based on the command being run.

  Example:

  ```js
  const examplePlugin = () => {
    let config

    return {
      name: 'read-config',

      configResolved(resolvedConfig) {
        // store the resolved config
        config = resolvedConfig
      },

      // use stored config in other hooks
      transform(code, id) {
        if (config.command === 'serve') {
          // dev: plugin invoked by dev server
        } else {
          // build: plugin invoked by Rollup
        }
      },
    }
  }
  ```

  Note that the `command` value is `serve` in dev (in the cli `vite`, `vite dev`, and `vite serve` are aliases).

### `configureServer`

- **Type:** `(server: ViteDevServer) => (() => void) | void | Promise<(() => void) | void>`
- **Kind:** `async`, `sequential`
- **See also:** [ViteDevServer](./api-javascript#vitedevserver)

  Hook for configuring the dev server. The most common use case is adding custom middlewares to the internal [connect](https://github.com/senchalabs/connect) app:

  ```js
  const myPlugin = () => ({
    name: 'configure-server',
    configureServer(server) {
      server.middlewares.use((req, res, next) => {
        // custom handle request...
      })
    },
  })
  ```

  Injecting Post Middleware

  The `configureServer` hook is called before internal middlewares are installed, so the custom middlewares will run before internal middlewares by default. If you want to inject a middleware **after** internal middlewares, you can return a function from `configureServer`, which will be called after internal middlewares are installed:

  ```js
  const myPlugin = () => ({
    name: 'configure-server',
    configureServer(server) {
      // return a post hook that is called after internal middlewares are
      // installed
      return () => {
        server.middlewares.use((req, res, next) => {
          // custom handle request...
        })
      }
    },
  })
  ```

  Storing Server Access

  In some cases, other plugin hooks may need access to the dev server instance (e.g. accessing the web socket server, the file system watcher, or the module graph). This hook can also be used to store the server instance for access in other hooks:

```js
const myPlugin = () => {
  let server
  return {
    name: 'configure-server',
    configureServer(_server) {
      server = _server
    },
    transform(code, id) {
      if (server) {
        // use server...
      }
    },
  }
}
```

Note `configureServer` is not called when running the production build so your other hooks need to guard against its absence.

### `configurePreviewServer`

- **Type:** `(server: PreviewServer) => (() => void) | void | Promise<(() => void) | void>`
- **Kind:** `async`, `sequential`
- **See also:** [PreviewServer](./api-javascript#previewserver)

Same as [`configureServer`](/guide/api-plugin.html#configureserver) but for the preview server. Similarly to `configureServer`, the `configurePreviewServer` hook is called before other middlewares are installed. If you want to inject a middleware **after** other middlewares, you can return a function from `configurePreviewServer`, which will be called after internal middlewares are installed:

```js
const myPlugin = () => ({
  name: 'configure-preview-server',
  configurePreviewServer(server) {
    // return a post hook that is called after other middlewares are
    // installed
    return () => {
      server.middlewares.use((req, res, next) => {
        // custom handle request...
      })
    }
  },
})
```

### `transformIndexHtml`

- **Type:** `IndexHtmlTransformHook | { order?: 'pre' | 'post', handler: IndexHtmlTransformHook }`
- **Kind:** `async`, `sequential`

Dedicated hook for transforming HTML entry point files such as `index.html`. The hook receives the current HTML string and a transform context. The context exposes the [`ViteDevServer`](./api-javascript#vitedevserver) instance during dev, and exposes the Rollup output bundle during build.

The hook can be async and can return one of the following:

- Transformed HTML string
- An array of tag descriptor objects (`{ tag, attrs, children }`) to inject to the existing HTML. Each tag can also specify where it should be injected to (default is prepending to `<head>`)
- An object containing both as `{ html, tags }`

By default `order` is `undefined`, with this hook applied after the HTML has been transformed. In order to inject a script that should go through the Vite plugins pipeline, `order: 'pre'` will apply the hook before processing the HTML. `order: 'post'` applies the hook after all hooks with `order` undefined are applied.

Basic Example:

```js
const htmlPlugin = () => {
  return {
    name: 'html-transform',
    transformIndexHtml(html) {
      return html.replace(
        /<title>(.*?)<\/title>/,
        `<title>Title replaced!</title>`,
      )
    },
  }
}
```

  Full Hook Signature:

  ```ts
  type IndexHtmlTransformHook = (
    html: string,
    ctx: {
      path: string
      filename: string
      server?: ViteDevServer
      bundle?: import('rollup').OutputBundle
      chunk?: import('rollup').OutputChunk
    },
  ) =>
    | IndexHtmlTransformResult
    | void
    | Promise<IndexHtmlTransformResult | void>

  type IndexHtmlTransformResult =
    | string
    | HtmlTagDescriptor[]
    | {
        html: string
        tags: HtmlTagDescriptor[]
      }

  interface HtmlTagDescriptor {
    tag: string
    attrs?: Record<string, string | boolean>
    children?: string | HtmlTagDescriptor[]
    /**
     * default: 'head-prepend'
     */
    injectTo?: 'head' | 'body' | 'head-prepend' | 'body-prepend'
  }
  ```

  ::: warning Note
  This hook won't be called if you are using a framework that has custom handling of entry files (for example [SvelteKit](https://github.com/sveltejs/kit/discussions/8269#discussioncomment-4509145)).
  :::

### `handleHotUpdate`

- **Type:** `(ctx: HmrContext) => Array<ModuleNode> | void | Promise<Array<ModuleNode> | void>`
- **See also:** [HMR API](./api-hmr)

  Perform custom HMR update handling. The hook receives a context object with the following signature:

  ```ts
  interface HmrContext {
    file: string
    timestamp: number
    modules: Array<ModuleNode>
    read: () => string | Promise<string>
    server: ViteDevServer
  }
  ```

  - `modules` is an array of modules that are affected by the changed file. It's an array because a single file may map to multiple served modules (e.g. Vue SFCs).

  - `read` is an async read function that returns the content of the file. This is provided because on some systems, the file change callback may fire too fast before the editor finishes updating the file and direct `fs.readFile` will return empty content. The read function passed in normalizes this behavior.

  The hook can choose to:

  - Filter and narrow down the affected module list so that the HMR is more accurate.

  - Return an empty array and perform a full reload:

    ```js
    handleHotUpdate({ server, modules, timestamp }) {
      // Invalidate modules manually
      const invalidatedModules = new Set()
      for (const mod of modules) {
        server.moduleGraph.invalidateModule(
          mod,
          invalidatedModules,
          timestamp,
          true
        )
      }
    ```

      server.ws.send({ type: 'full-reload' })
      return []
    }

  - Return an empty array and perform complete custom HMR handling by sending custom events to the client:

    ```js
    handleHotUpdate({ server }) {
      server.ws.send({
        type: 'custom',
        event: 'special-update',
        data: {}
      })
      return []
    }
    ```

    Client code should register corresponding handler using the [HMR API](./api-hmr) (this could be injected by the same plugin's `transform` hook):

    ```js
    if (import.meta.hot) {
      import.meta.hot.on('special-update', (data) => {
        // perform custom update
      })
    }
    ```

## Plugin Ordering

A Vite plugin can additionally specify an `enforce` property (similar to webpack loaders) to adjust its application order. The value of `enforce` can be either `"pre"` or `"post"`. The resolved plugins will be in the following order:

- Alias
- User plugins with `enforce: 'pre'`
- Vite core plugins
- User plugins without enforce value
- Vite build plugins
- User plugins with `enforce: 'post'`
- Vite post build plugins (minify, manifest, reporting)

Note that this is separate from hooks ordering, those are still separately subject to their `order` attribute [as usual for Rollup hooks](https://rollupjs.org/plugin-development/#build-hooks).

## Conditional Application

By default plugins are invoked for both serve and build. In cases where a plugin needs to be conditionally applied only during serve or build, use the `apply` property to only invoke them during `'build'` or `'serve'`:

```js
function myPlugin() {
  return {
    name: 'build-only',
    apply: 'build', // or 'serve'
  }
}
```

A function can also be used for more precise control:

```js
apply(config, { command }) {
  // apply only on build but not for SSR
  return command === 'build' && !config.build.ssr
}
```

## Rollup Plugin Compatibility

A fair number of Rollup plugins will work directly as a Vite plugin (e.g. `@rollup/plugin-alias` or `@rollup/plugin-json`), but not all of them, since some plugin hooks do not make sense in an unbundled dev server context.

In general, as long as a Rollup plugin fits the following criteria then it should just work as a Vite plugin:

- It doesn't use the [`moduleParsed`](https://rollupjs.org/plugin-development/#moduleparsed) hook.
- It doesn't have strong coupling between bundle-phase hooks and output-phase hooks.

If a Rollup plugin only makes sense for the build phase, then it can be specified under `build.rollupOptions.plugins` instead. It will work the same as a Vite plugin with `enforce: 'post'` and `apply: 'build'`.

You can also augment an existing Rollup plugin with Vite-only properties:

```js
// vite.config.js
import example from 'rollup-plugin-example'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [
    {
      ...example(),
      enforce: 'post',
      apply: 'build',
    },
  ],
})
```

## Path Normalization

Vite normalizes paths while resolving ids to use POSIX separators (/) while preserving the volume in Windows. On the other hand, Rollup keeps resolved paths untouched by default, so resolved ids have win32 separators (\\ ) in Windows. However, Rollup plugins use a [`normalizePath` utility function](https://github.com/rollup/plugins/tree/master/packages/pluginutils#normalizepath) from `@rollup/pluginutils` internally, which converts separators to POSIX before performing comparisons. This means that when these plugins are used in Vite, the `include` and `exclude` config pattern and other similar paths against resolved ids comparisons work correctly.

So, for Vite plugins, when comparing paths against resolved ids it is important to first normalize the paths to use POSIX separators. An equivalent `normalizePath` utility function is exported from the `vite` module.

```js
import { normalizePath } from 'vite'

normalizePath('foo\\bar') // 'foo/bar'
normalizePath('foo/bar') // 'foo/bar'
```

## Filtering, include/exclude pattern

Vite exposes [`@rollup/pluginutils`'s `createFilter`](https://github.com/rollup/plugins/tree/master/packages/pluginutils#createfilter) function to encourage Vite specific plugins and integrations to use the standard include/exclude filtering pattern, which is also used in Vite core itself.

## Client-server Communication

Since Vite 2.9, we provide some utilities for plugins to help handle the communication with clients.

### Server to Client

On the plugin side, we could use `server.ws.send` to broadcast events to the client:

```js
// vite.config.js
export default defineConfig({
  plugins: [
    {
      // ...
      configureServer(server) {
        server.ws.on('connection', () => {
          server.ws.send('my:greetings', { msg: 'hello' })
        })
      },
    },
  ],
})
```

::: tip NOTE
We recommend **always prefixing** your event names to avoid collisions with other plugins.
:::

On the client side, use [`hot.on`](/guide/api-hmr.html#hot-on-event-cb) to listen to the events:

```ts twoslash
import 'vite/client'
// ---cut---
// client side
if (import.meta.hot) {
  import.meta.hot.on('my:greetings', (data) => {
    console.log(data.msg) // hello
  })
}
```

### Client to Server

To send events from the client to the server, we can use [`hot.send`](/guide/api-hmr.html#hot-send-event-payload):

```ts
// client side
if (import.meta.hot) {
  import.meta.hot.send('my:from-client', { msg: 'Hey!' })
}
```

Then use `server.ws.on` and listen to the events on the server side:

```js
// vite.config.js
export default defineConfig({
  plugins: [
    {
      // ...
      configureServer(server) {
        server.ws.on('my:from-client', (data, client) => {
          console.log('Message from client:', data.msg) // Hey!
          // reply only to the client (if needed)
          client.send('my:ack', { msg: 'Hi! I got your message!' })
        })
      },
    },
  ],
})
```

### TypeScript for Custom Events

Internally, vite infers the type of a payload from the `CustomEventMap` interface, it is possible to type custom events by extending the interface:

:::tip Note
Make sure to include the `.d.ts` extension when specifying TypeScript declaration files. Otherwise, Typescript may not know which file the module is trying to extend.
:::

```ts
// events.d.ts
import 'vite/types/customEvent.d.ts'

declare module 'vite/types/customEvent.d.ts' {
  interface CustomEventMap {
    'custom:foo': { msg: string }
    // 'event-key': payload
  }
}
```

This interface extension is utilized by `InferCustomEventPayload<T>` to infer the payload type for event `T`. For more information on how this interface is utilized, refer to the [HMR API Documentation](./api-hmr#hmr-api).

```ts twoslash
import 'vite/client'
import type { InferCustomEventPayload } from 'vite/types/customEvent.d.ts'
declare module 'vite/types/customEvent.d.ts' {
  interface CustomEventMap {
    'custom:foo': { msg: string }
  }
}
// ---cut---
type CustomFooPayload = InferCustomEventPayload<'custom:foo'>
import.meta.hot?.on('custom:foo', (payload) => {
  // The type of payload will be { msg: string }
})
import.meta.hot?.on('unknown:event', (payload) => {
  // The type of payload will be any
})
```

```
---
title: Vite 4.0 is out!
author:
  name: The Vite Team
date: 2022-12-09
sidebar: false
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 4
  - - meta
    - property: og:image
      content: https://vitejs.dev/og-image-announcing-vite4.png
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite4
  - - meta
    - property: og:description
      content: Vite 4 Release Announcement
  - - meta
    - name: twitter:card
      content: summary_large_image
---

# Vite 4.0 is out!

_December 9, 2022_ - Check out the [Vite 5.0 announcement](./announcing-vite5.md)

Vite 3 [was released](./announcing-vite3.md) five months ago. npm downloads per week have gone from 1 million to 2.5 million
since then. The ecosystem has matured too, and continues to grow. In this year's [Jamstack Conf survey]
(https://twitter.com/vite_js/status/1589665610119585793), usage among the community jumped from 14% to 32% while keeping a
high 9.7 satisfaction score. We saw the stable releases of [Astro 1.0](https://astro.build/), [Nuxt 3]
(https://v3.nuxtjs.org/), and other Vite-powered frameworks that are innovating and collaborating: [SvelteKit]
(https://kit.svelte.dev/), [Solid Start](https://www.solidjs.com/blog/introducing-solidstart), [Qwik City]
(https://qwik.builder.io/qwikcity/overview/). Storybook announced first-class support for Vite as one of its main features
for [Storybook 7.0](https://storybook.js.org/blog/first-class-vite-support-in-storybook/). Deno now [supports Vite]
(https://www.youtube.com/watch?v=Zjojo9wdvmY). [Vitest](https://vitest.dev) adoption is exploding, it will soon represent
half of Vite's npm downloads. Nx is also investing in the ecosystem, and [officially supports Vite]
(https://nx.dev/packages/vite).

[![Vite 4 Ecosystem](/ecosystem-vite4.png)](https://viteconf.org/2022/replay)

As a showcase of the growth Vite and related projects have experienced, the Vite ecosystem gathered on October 11th at
[ViteConf 2022](https://viteconf.org/2022/replay). We saw representatives from the main web framework and tools tell stories
of innovation and collaboration. And in a symbolic move, the Rollup team choose that exact day to release [Rollup 3]
(https://rollupjs.org).

Today, the Vite [team](https://vitejs.dev/team) with the help of our ecosystem partners, is happy to announce the release of
Vite 4, powered during build time by Rollup 3. We've worked with the ecosystem to ensure a smooth upgrade path for this new
major. Vite is now using [Rollup 3](https://github.com/vitejs/vite/issues/9870), which allowed us to simplify Vite's internal
asset handling and has many improvements. See the [Rollup 3 release notes here]
(https://github.com/rollup/rollup/releases/tag/v3.0.0).

![Vite 4 Announcement Cover Image](/og-image-announcing-vite4.png)

Quick links:

- [Docs](/)
- [Migration Guide](https://v4.vitejs.dev/guide/migration.html)
- [Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#400-2022-12-09)

Docs in other languages:

- [ç®€ä½“ä¸æ–‡](https://cn.vitejs.dev/)
- [æ—¥æœ¬èªž](https://ja.vitejs.dev/)
- [EspaÃ±ol](https://es.vitejs.dev/)

If you recently started using Vite, we suggest reading the [Why Vite Guide](https://vitejs.dev/guide/why.html) and checking
out [the Getting Started](https://vitejs.dev/guide/) and [Features guide](https://vitejs.dev/guide/features). If you want to
get involved, contributions are welcome at [GitHub](https://github.com/vitejs/vite). Almost [700 collaborators]
(https://github.com/vitejs/vite/graphs/contributors) have contributed to Vite. Follow the updates on [Twitter]
(https://twitter.com/vite_js) and [Mastodon](https://webtoo.ls/@vite), or join collaborate with others on our [Discord
community](http://chat.vitejs.dev/).

## Start playing with Vite 4

Use `pnpm create vite` to scaffold a Vite project with your preferred framework, or open a started template online to play
with Vite 4 using [vite.new](https://vite.new).

You can also run `pnpm create vite-extra` to get access to templates from other frameworks and runtimes (Solid, Deno, SSR,
```

and library starters). `create vite-extra` templates are also available when you run `create vite` under the `Others` option.

Note that Vite starter templates are intended to be used as a playground to test Vite with different frameworks. When building your next project, we recommend reaching out to the starters recommended by each framework. Some frameworks now redirect in `create vite` to their starters too (`create-vue` and `Nuxt 3` for Vue, and `SvelteKit` for Svelte).

## New React plugin using SWC during development

[SWC](https://swc.rs/) is now a mature replacement for [Babel](https://babeljs.io/), especially in the context of React projects. SWC's React Fast Refresh implementation is a lot faster than Babel, and for some projects, it is now a better alternative. From Vite 4, two plugins are available for React projects with different tradeoffs. We believe that both approaches are worth supporting at this point, and we'll continue to explore improvements to both plugins in the future.

### @vitejs/plugin-react

[@vitejs/plugin-react](https://github.com/vitejs/vite-plugin-react) is a plugin that uses esbuild and Babel, achieving fast HMR with a small package footprint and the flexibility of being able to use the Babel transform pipeline.

### @vitejs/plugin-react-swc (new)

[@vitejs/plugin-react-swc](https://github.com/vitejs/vite-plugin-react-swc) is a new plugin that uses esbuild during build, but replaces Babel with SWC during development. For big projects that don't require non-standard React extensions, cold start and Hot Module Replacement (HMR) can be significantly faster.

## Browser Compatibility

The modern browser build now targets `safari14` by default for wider ES2020 compatibility. This means that modern builds can now use [`BigInt`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) and that the [nullish coalescing operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing) isn't transpiled anymore. If you need to support older browsers, you can add [`@vitejs/plugin-legacy`](https://github.com/vitejs/vite/tree/main/packages/plugin-legacy) as usual.

## Importing CSS as a String

In Vite 3, importing the default export of a `.css` file could introduce a double loading of CSS.

```ts
import cssString from './global.css'
```

This double loading could occur since a `.css` file will be emitted and it's likely that the CSS string will also be used by the application code â€” for example, injected by the framework runtime. From Vite 4, the `.css` default export [has been deprecated](https://github.com/vitejs/vite/issues/11094). The `?inline` query suffix modifier needs to be used in this case, as that doesn't emit the imported `.css` styles.

```ts
import stuff from './global.css?inline'
```

Learn more in the [Migration Guide](https://v4.vitejs.dev/guide/migration.html).

## Environment Variables

Vite now uses `dotenv` 16 and `dotenv-expand` 9 (previously `dotenv` 14 and `dotenv-expand` 5). If you have a value including `#` or `` ` ``, you will need to wrap them with quotes.

```diff
-VITE_APP=ab#cd`ef
+VITE_APP="ab#cd`ef"
```

For more details, see the [`dotenv`](https://github.com/motdotla/dotenv/blob/master/CHANGELOG.md) and [`dotenv-expand` changelog](https://github.com/motdotla/dotenv-expand/blob/master/CHANGELOG.md).

## Other Features

- CLI Shortcuts (press `h` during dev to see them all) ([#11228](https://github.com/vitejs/vite/pull/11228))
- Support for patch-package when pre bundling dependencies ([#10286](https://github.com/vitejs/vite/issues/10286))
- Cleaner build logs output ([#10895](https://github.com/vitejs/vite/issues/10895)) and switch to `kB` to align with browser dev tools ([#10982](https://github.com/vitejs/vite/issues/10982))
- Improved error messages during SSR ([#11156](https://github.com/vitejs/vite/issues/11156))

## Reduced Package Size

Vite cares about its footprint, to speed up installation, especially in the use case of playgrounds for documentation and reproductions. And once more, this major brings improvements in Vite's package size. Vite 4 install size is 23% smaller compared to vite 3.2.5 (14.1 MB vs 18.3 MB).

## Upgrades to Vite Core

[Vite Core](https://github.com/vitejs/vite) and [vite-ecosystem-ci](https://github.com/vitejs/vite-ecosystem-ci) continue to evolve to provide a better experience to maintainers and collaborators and to ensure that Vite development scales to cope with the growth in the ecosystem.

### Framework plugins out of core

[`@vitejs/plugin-vue`](https://github.com/vitejs/vite-plugin-vue) and [`@vitejs/plugin-react`]
(https://github.com/vitejs/vite-plugin-react) have been part of Vite core monorepo since the first versions of Vite. This
helped us to get a close feedback loop when making changes as we were getting both Core and the plugins tested and released
together. With [vite-ecosystem-ci](https://github.com/vitejs/vite-ecosystem-ci) we can get this feedback with these plugins
developed on independent repositories, so from Vite 4, [they have been moved out of the Vite core monorepo]
(https://github.com/vitejs/vite/pull/11158). This is meaningful for Vite's framework-agnostic story and will allow us to
build independent teams to maintain each of the plugins. If you have bugs to report or features to request, please create
issues on the new repositories moving forward: [`vitejs/vite-plugin-vue`](https://github.com/vitejs/vite-plugin-vue) and
[`vitejs/vite-plugin-react`](https://github.com/vitejs/vite-plugin-react).

### vite-ecosystem-ci improvements

[vite-ecosystem-ci](https://github.com/vitejs/vite-ecosystem-ci) extends Vite's CI by providing on-demand status reports on
the state of the CIs of [most major downstream projects](https://github.com/vitejs/vite-ecosystem-ci/tree/main/tests). We run
vite-ecosystem-ci three times a week against Vite's main branch and receive timely reports before introducing a regression.
Vite 4 will soon be compatible with most projects using Vite, which already prepared branches with the needed changes and
will be releasing them in the next few days. We are also able to run vite-ecosystem-ci on-demand on PRs using `/ecosystem-ci
run` in a comment, allowing us to know [the effect of changes](https://github.com/vitejs/vite/pull/11269#issuecomment-
1343365064) before they hit main.

## Acknowledgments

Vite 4 wouldn't be possible without uncountable hours of work by Vite contributors, many of them maintainers of downstream
projects and plugins, and the efforts of the [Vite Team](/team). All of us have worked together to improve Vite's DX once
more, for every framework and app using it. We're grateful to be able to improve a common base for such a vibrant ecosystem.

We're also thankful to individuals and companies sponsoring the Vite team, and companies investing directly in Vite's future:
[@antfu7](https://twitter.com/antfu7)'s work on Vite and the ecosystem is part of his job at [Nuxt Labs]
(https://nuxtlabs.com/), [Astro](https://astro.build) is funding [@bluwyoo](https://twitter.com/bluwyoo)'s' Vite core work,
and [StackBlitz](https://stackblitz.com/) hires [@patak_dev](https://twitter.com/patak_dev) to work full time on Vite.

## Next steps

Our immediate focus would be on triaging newly opened issues to avoid disruption by possible regressions. If you would like
to get involved and help us improve Vite, we suggest starting with issues triaging. Join [our Discord]
(https://chat.vitejs.dev) and reach out on the `#contributing` channel. Polish our `#docs` story, and `#help` others. We need
to continue to build a helpful and welcoming community for the next wave of users, as Vite's adoption continues to grow.

There are a lot of open fronts to keep improving the DX of everyone that has chosen Vite to power their frameworks and
develop their apps. Onwards!

```
---
title: Vite 5.1 is out!
author:
  name: The Vite Team
date: 2024-02-08
sidebar: false
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 5.1
  - - meta
    - property: og:image
      content: https://vitejs.dev/og-image-announcing-vite5-1.png
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite5-1
  - - meta
    - property: og:description
      content: Vite 5.1 Release Announcement
  - - meta
    - name: twitter:card
      content: summary_large_image
---
```

# Vite 5.1 is out!

February 8, 2024

![Vite 5.1 Announcement Cover Image](/og-image-announcing-vite5-1.png)

Vite 5 [was released](./announcing-vite5.md) last November, and it represented another big leap for Vite and the ecosystem. A few weeks ago we celebrated 10 million weekly npm downloads and 900 contributors to the Vite repo. Today, we're excited to announce the release of Vite 5.1.

Quick links: [Docs](/), [Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#510-2024-02-08)

Docs in other languages: [ç®€ä½“ä¸æ–‡](https://cn.vitejs.dev/), [æ—¥æœ¬èªž](https://ja.vitejs.dev/), [EspaÃ±ol](https://es.vitejs.dev/), [PortuguÃªs](https://pt.vitejs.dev/), [í•œêµì–´](https://ko.vitejs.dev/), [Deutsch](https://de.vitejs.dev/)

Try Vite 5.1 online in StackBlitz: [vanilla](https://vite.new/vanilla-ts), [vue](https://vite.new/vue-ts), [react](https://vite.new/react-ts), [preact](https://vite.new/preact-ts), [lit](https://vite.new/lit-ts), [svelte](https://vite.new/svelte-ts), [solid](https://vite.new/solid-ts), [qwik](https://vite.new/qwik-ts).

If you're new to Vite, we suggest reading first the [Getting Started](/guide/) and [Features](/guide/features) guides.

To stay up to date, follow us on [X](https://x.com/vite_js) or [Mastodon](https://webtoo.ls/@vite).

## Vite Runtime API

Vite 5.1 adds experimental support for a new Vite Runtime API. It allows running any code by processing it with Vite plugins first. It is different from `server.ssrLoadModule` because the runtime implementation is decoupled from the server. This lets library and framework authors implement their own layer of communication between the server and the runtime. This new API is intended to replace Vite's current SSR primitives once it is stable.

The new API brings many benefits:

- Support for HMR during SSR.
- It is decoupled from the server, so there is no limit on how many clients can use a single server - every client has its own module cache (you can even communicate with it how you want - using message channel/fetch call/direct function call/websocket).
- It doesn't depend on any node/bun/deno built-in APIs, so it can run in any environment.
- It's easy to integrate with tools that have their own mechanism to run code (you can provide a runner to use `eval` instead of `new AsyncFunction` for example).

The initial idea [was proposed by Pooya Parsa](https://github.com/nuxt/vite/pull/201) and implemented by [Anthony Fu](https://github.com/antfu) as the [vite-node](https://github.com/vitest-dev/vitest/tree/main/packages/vite-node#readme) package to [power Nuxt 3 Dev SSR](https://antfu.me/posts/dev-ssr-on-nuxt) and later also used as the base for [Vitest](https://vitest.dev). So the general idea of vite-node has been battle-tested for quite some time now. This is a new iteration of the API by [Vladimir Sheremet](https://github.com/sheremet-va), who had already re-implemented vite-node in Vitest and took the learnings to make the API even more powerful and flexible when adding it to Vite Core. The PR was one year in the makings, you can see the evolution and discussions with ecosystem maintainers [here](https://github.com/vitejs/vite/issues/12165).

Read more in the [Vite Runtime API guide](/guide/api-vite-runtime) and [give us feedback](https://github.com/vitejs/vite/discussions/15774).

## Features

### Improved support for `.css?url`

Import CSS files as URLs now works reliably and correctly. This was the last remaining hurdle in Remix's move to Vite. See
([#15259](https://github.com/vitejs/vite/issues/15259)).

### `build.assetsInlineLimit` now supports a callback

Users can now [provide a callback](/config/build-options.html#build-assetsinlinelimit) that returns a boolean to opt-in or
opt-out of inlining for specific assets. If `undefined` is returned, the default logic applies. See ([#15366]
(https://github.com/vitejs/vite/issues/15366)).

### Improved HMR for circular import

In Vite 5.0, accepted modules within circular imports always triggered a full page reload even if they can be handled fine in
the client. This is now relaxed to allow HMR to apply without a full page reload, but if any error happens during HMR, the
page will be reloaded. See ([#15118](https://github.com/vitejs/vite/issues/15118)).

### Support `ssr.external: true` to externalize all SSR packages

Historically, Vite externalizes all packages except for linked packages. This new option can be used to force externalize all
packages including linked packages too. This is handy in tests within monorepos where we want to emulate the usual case of
all packages externalized, or when using `ssrLoadModule` to load an arbitrary file and we want to always external packages as
we don't care about HMR. See ([#10939](https://github.com/vitejs/vite/issues/10939)).

### Expose `close` method in the preview server

The preview server now exposes a `close` method, which will properly teardown the server including all opened socket
connections. See ([#15630](https://github.com/vitejs/vite/issues/15630)).

## Performance improvements

Vite keeps getting faster with each release, and Vite 5.1 is packed with performance improvements. We measured the loading
time for 10K modules (25 level deep tree) using [vite-dev-server-perf](https://github.com/yyx990803/vite-dev-server-perf) for
all minor versions from Vite 4.0. This is a good benchmark to measure the effect of Vite's bundle-less approach. Each module
is a small TypeScript file with a counter and imports to other files in the tree, so this mostly measuring the time it takes
to do the requests a separate modules. In Vite 4.0, loading 10K modules took 8 seconds on a M1 MAX. We had a breakthrough in
[Vite 4.3 were we focused on performance](./announcing-vite4-3.md), and we were able to load them in 6.35 seconds. In Vite
5.1, we managed to do another performance leap. Vite is now serving the 10K modules in 5.35 seconds.

![Vite 10K Modules Loading time progression](/vite5-1-10K-modules-loading-time.png)

The results of this benchmark run on Headless Puppeteer and are a good way to compare versions. They don't represent the time
as experienced by users though. When running the same 10K modules in an Incognito window is Chrome, we have:

10K Modules	Vite 5.0	Vite 5.1
Loading time	2892ms	2765ms
Loading time (cached)	2778ms	2477ms
Full reload	2003ms	1878ms
Full reload (cached)	1682ms	1604ms

### Run CSS preprocessors in threads

Vite now has opt-in support for running CSS preprocessors in threads. You can enable it using [`css.preprocessorMaxWorkers:
true`](/config/shared-options.html#css-preprocessormaxworkers). For a Vuetify 2 project, dev startup time was reduced by 40%
with this feature enabled. There is [performance comparison for others setups in the PR]
(https://github.com/vitejs/vite/pull/13584#issuecomment-1678827918). See ([#13584]
(https://github.com/vitejs/vite/issues/13584)). [Give Feedback](https://github.com/vitejs/vite/discussions/15835).

### New options to improve server cold starts

You can set `optimizeDeps.holdUntilCrawlEnd: false` to switch to a new strategy for deps optimization that may help in big
projects. We're considering switching to this strategy by default in the future. [Give Feedback]
(https://github.com/vitejs/vite/discussions/15834). ([#15244](https://github.com/vitejs/vite/issues/15244))

### Faster resolving with cached checks

The `fs.cachedChecks` optimization is now enabled by default. In Windows, `tryFsResolve` was ~14x faster with it, and
resolving ids overall got a ~5x speed up in the triangle benchmark. ([#15704](https://github.com/vitejs/vite/issues/15704))

### Internal performance improvements

The dev server had several incremental performance gains. A new middleware to short-circuit on 304 ([#15586]
(https://github.com/vitejs/vite/issues/15586)). We avoided `parseRequest` in hot paths ([#15617]
(https://github.com/vitejs/vite/issues/15617)). Rollup is now properly lazy loaded ([#15621]
(https://github.com/vitejs/vite/issues/15621))

## Deprecations

We continue to reduce Vite's API surface where possible to make the project maintainable long term.

### Deprecated `as` option in `import.meta.glob`

The standard moved to [Import Attributes](https://github.com/tc39/proposal-import-attributes), but we don't plan to replace `as` with a new option at this point. Instead, it is recommended that the user switches to `query`. See ([#14420](https://github.com/vitejs/vite/issues/14420)).

### Removed experimental build-time pre-bundling

Build-time pre-bundling, an experimental feature added in Vite 3, is removed. With Rollup 4 switching its parser to native, and Rolldown being worked on, both the performance and the dev-vs-build inconsistency story for this feature are no longer valid. We want to continue improving dev/build consistency, and have concluded that using Rolldown for "prebundling during dev" and "production builds" is the better bet moving forward. Rolldown may also implement caching in a way that is a lot more efficient during build than deps prebundling. See ([#15184](https://github.com/vitejs/vite/issues/15184)).

## Get Involved

We are grateful to the [900 contributors to Vite Core](https://github.com/vitejs/vite/graphs/contributors), and the maintainers of plugins, integrations, tools, and translations that keeps pushing the ecosystem forward. If you're enjoying Vite, we invite you to participate and help us. Check out our [Contributing Guide](https://github.com/vitejs/vite/blob/main/CONTRIBUTING.md), and jump into [triaging issues](https://github.com/vitejs/vite/issues), [reviewing PRs](https://github.com/vitejs/vite/pulls), answering questions at [GitHub Discussions](https://github.com/vitejs/vite/discussions) and helping others in the community in [Vite Land](https://chat.vitejs.dev).

## Acknowledgments

Vite 5.1 is possible thanks to our community of contributors, maintainers in the ecosystem, and the [Vite Team](/team). A shout out to the individuals and companies sponsoring Vite development. [StackBlitz](https://stackblitz.com/), [Nuxt Labs](https://nuxtlabs.com/), and [Astro](https://astro.build) for hiring Vite team members. And also to the sponsors on [Vite's GitHub Sponsors](https://github.com/sponsors/vitejs), [Vite's Open Collective](https://opencollective.com/vite), and [Evan You's GitHub Sponsors](https://github.com/sponsors/yyx990803).

# Shared Options

## root

- **Type:** `string`
- **Default:** `process.cwd()`

Project root directory (where `index.html` is located). Can be an absolute path, or a path relative to the current working directory.

See [Project Root](/guide/#index-html-and-project-root) for more details.

## base

- **Type:** `string`
- **Default:** `/`
- **Related:** [`server.origin`](/config/server-options.md#server-origin)

Base public path when served in development or production. Valid values include:

- Absolute URL pathname, e.g. `/foo/`
- Full URL, e.g. `https://bar.com/foo/` (The origin part won't be used in development so the value is the same as `/foo/`)
- Empty string or `./` (for embedded deployment)

See [Public Base Path](/guide/build#public-base-path) for more details.

## mode

- **Type:** `string`
- **Default:** `'development'` for serve, `'production'` for build

Specifying this in config will override the default mode for **both serve and build**. This value can also be overridden via the command line `--mode` option.

See [Env Variables and Modes](/guide/env-and-mode) for more details.

## define

- **Type:** `Record<string, any>`

Define global constant replacements. Entries will be defined as globals during dev and statically replaced during build.

Vite uses [esbuild defines](https://esbuild.github.io/api/#define) to perform replacements, so value expressions must be a string that contains a JSON-serializable value (null, boolean, number, string, array, or object) or a single identifier. For non-string values, Vite will automatically convert it to a string with `JSON.stringify`.

Example:

```js
export default defineConfig({
  define: {
    __APP_VERSION__: JSON.stringify('v1.0.0'),
    __API_URL__: 'window.__backend_api_url',
  },
})
```

::: tip NOTE
For TypeScript users, make sure to add the type declarations in the `env.d.ts` or `vite-env.d.ts` file to get type checks and Intellisense.

Example:

```ts
// vite-env.d.ts
declare const __APP_VERSION__: string
```

:::

## plugins

- **Type:** `(Plugin | Plugin[] | Promise<Plugin | Plugin[]>)[]`

Array of plugins to use. Falsy plugins are ignored and arrays of plugins are flattened. If a promise is returned, it would be resolved before running. See [Plugin API](/guide/api-plugin) for more details on Vite plugins.

## publicDir

- **Type:** `string | false`
- **Default:** `"public"`

Directory to serve as plain static assets. Files in this directory are served at `/` during dev and copied to the root of `outDir` during build, and are always served or copied as-is without transform. The value can be either an absolute file system path or a path relative to project root.

Defining `publicDir` as `false` disables this feature.

See [The `public` Directory](/guide/assets#the-public-directory) for more details.

## cacheDir

- **Type:** `string`
- **Default:** `"node_modules/.vite"`

Directory to save cache files. Files in this directory are pre-bundled deps or some other cache files generated by vite, which can improve the performance. You can use `--force` flag or manually delete the directory to regenerate the cache files. The value can be either an absolute file system path or a path relative to project root. Default to `.vite` when no package.json is detected.

## resolve.alias

- **Type:**
  `Record<string, string> | Array<{ find: string | RegExp, replacement: string, customResolver?: ResolverFunction | ResolverObject }>`

Will be passed to `@rollup/plugin-alias` as its [entries option](https://github.com/rollup/plugins/tree/master/packages/alias#entries). Can either be an object, or an array of `{ find, replacement, customResolver }` pairs.

When aliasing to file system paths, always use absolute paths. Relative alias values will be used as-is and will not be resolved into file system paths.

More advanced custom resolution can be achieved through [plugins](/guide/api-plugin).

::: warning Using with SSR
If you have configured aliases for [SSR externalized dependencies](/guide/ssr.md#ssr-externals), you may want to alias the actual `node_modules` packages. Both [Yarn](https://classic.yarnpkg.com/en/docs/cli/add/#toc-yarn-add-alias) and [pnpm](https://pnpm.io/aliases/) support aliasing via the `npm:` prefix.
:::

## resolve.dedupe

- **Type:** `string[]`

If you have duplicated copies of the same dependency in your app (likely due to hoisting or linked packages in monorepos), use this option to force Vite to always resolve listed dependencies to the same copy (from project root).

:::warning SSR + ESM
For SSR builds, deduplication does not work for ESM build outputs configured from `build.rollupOptions.output`. A workaround is to use CJS build outputs until ESM has better plugin support for module loading.
:::

## resolve.conditions

- **Type:** `string[]`

Additional allowed conditions when resolving [Conditional Exports](https://nodejs.org/api/packages.html#packages_conditional_exports) from a package.

A package with conditional exports may have the following `exports` field in its `package.json`:

```json
{
  "exports": {
    ".": {
      "import": "./index.mjs",
      "require": "./index.js"
    }
  }
}
```

Here, `import` and `require` are "conditions". Conditions can be nested and should be specified from most specific to least specific.

Vite has a list of "allowed conditions" and will match the first condition that is in the allowed list. The default allowed conditions are: `import`, `module`, `browser`, `default`, and `production/development` based on current mode. The `resolve.conditions` config option allows specifying additional allowed conditions.

:::warning Resolving subpath exports
Export keys ending with "/" is deprecated by Node and may not work well. Please contact the package author to use [`*` subpath patterns](https://nodejs.org/api/packages.html#package-entry-points) instead.
:::

## resolve.mainFields

- **Type:** `string[]`
- **Default:** `['browser', 'module', 'jsnext:main', 'jsnext']`

List of fields in `package.json` to try when resolving a package's entry point. Note this takes lower precedence than conditional exports resolved from the `exports` field: if an entry point is successfully resolved from `exports`, the main field will be ignored.

## resolve.extensions

- **Type:** `string[]`
- **Default:** `['.mjs', '.js', '.mts', '.ts', '.jsx', '.tsx', '.json']`

List of file extensions to try for imports that omit extensions. Note it is **NOT** recommended to omit extensions for custom import types (e.g. `.vue`) since it can interfere with IDE and type support.

## resolve.preserveSymlinks

- **Type:** `boolean`
- **Default:** `false`

Enabling this setting causes vite to determine file identity by the original file path (i.e. the path without following symlinks) instead of the real file path (i.e. the path after following symlinks).

- **Related:** [esbuild#preserve-symlinks](https://esbuild.github.io/api/#preserve-symlinks), [webpack#resolve.symlinks
  ](https://webpack.js.org/configuration/resolve/#resolvesymlinks)

## html.cspNonce

- **Type:** `string`
- **Related:** [Content Security Policy (CSP)](/guide/features#content-security-policy-csp)

A nonce value placeholder that will be used when generating script / style tags. Setting this value will also generate a meta tag with nonce value.

## css.modules

- **Type:**
  ```ts
  interface CSSModulesOptions {
    getJSON?: (
      cssFileName: string,
      json: Record<string, string>,
      outputFileName: string,
    ) => void
    scopeBehaviour?: 'global' | 'local'
    globalModulePaths?: RegExp[]
    exportGlobals?: boolean
    generateScopedName?:
      | string
      | ((name: string, filename: string, css: string) => string)
    hashPrefix?: string
    /**
     * default: undefined
     */
    localsConvention?:
      | 'camelCase'
      | 'camelCaseOnly'
      | 'dashes'
      | 'dashesOnly'
      | ((
          originalClassName: string,
          generatedClassName: string,
          inputFile: string,
        ) => string)
  }
  ```

Configure CSS modules behavior. The options are passed on to [postcss-modules](https://github.com/css-modules/postcss-modules).

This option doesn't have any effect when using [Lightning CSS](../guide/features.md#lightning-css). If enabled, [`css.lightningcss.cssModules`](https://lightningcss.dev/css-modules.html) should be used instead.

## css.postcss

- **Type:** `string | (postcss.ProcessOptions & { plugins?: postcss.AcceptedPlugin[] })`

Inline PostCSS config or a custom directory to search PostCSS config from (default is project root).

For inline PostCSS config, it expects the same format as `postcss.config.js`. But for `plugins` property, only [array format](https://github.com/postcss/postcss-load-config/blob/main/README.md#array) can be used.

The search is done using [postcss-load-config](https://github.com/postcss/postcss-load-config) and only the supported config file names are loaded.

Note if an inline config is provided, Vite will not search for other PostCSS config sources.

## css.preprocessorOptions

- **Type:** `Record<string, object>`

Specify options to pass to CSS pre-processors. The file extensions are used as keys for the options. The supported options for each preprocessors can be found in their respective documentation:

- `sass`/`scss` - top level option `api: "legacy" | "modern" | "modern-compiler"` (default `"legacy"`) allows switching which sass API to use. For the best performance, it's recommended to use `api: "modern-compiler"` with `sass-embedded` package. [Options (legacy)](https://sass-lang.com/documentation/js-api/interfaces/LegacyStringOptions), [Options (modern)](https://sass-lang.com/documentation/js-api/interfaces/stringoptions/).
- `less` - [Options](https://lesscss.org/usage/#less-options).
- `styl`/`stylus` - Only [`define`](https://stylus-lang.com/docs/js.html#define-name-node) is supported, which can be passed as an object.

Example:

```js
export default defineConfig({
  css: {
    preprocessorOptions: {
      less: {
        math: 'parens-division',
      },
      styl: {
        define: {
          $specialColor: new stylus.nodes.RGBA(51, 197, 255, 1),
        },
      },
      scss: {
        api: 'modern-compiler', // or "modern", "legacy"
        importers: [
          // ...
        ],
      },
    },
  },
})
```

### css.preprocessorOptions[extension].additionalData

- **Type:** `string | ((source: string, filename: string) => (string | { content: string; map?: SourceMap }))`

This option can be used to inject extra code for each style content. Note that if you include actual styles and not just variables, those styles will be duplicated in the final bundle.

Example:

```js
export default defineConfig({
  css: {
    preprocessorOptions: {
      scss: {
        additionalData: `$injectedColor: orange;`,
      },
    },
  },
})
```

## css.preprocessorMaxWorkers

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/15835)
- **Type:** `number | true`
- **Default:** `0` (does not create any workers and run in the main thread)

If this option is set, CSS preprocessors will run in workers when possible. `true` means the number of CPUs minus 1.

## css.devSourcemap

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/13845)
- **Type:** `boolean`
- **Default:** `false`

Whether to enable sourcemaps during dev.

## css.transformer

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/13835)
- **Type:** `'postcss' | 'lightningcss'`
- **Default:** `'postcss'`

Selects the engine used for CSS processing. Check out [Lightning CSS](../guide/features.md#lightning-css) for more information.

::: info Duplicate `@import`s
Note that postcss (postcss-import) has a different behavior with duplicated `@import` from browsers. See [postcss/postcss-import#462](https://github.com/postcss/postcss-import/issues/462).
:::

## css.lightningcss

- **Experimental:** [Give Feedback](https://github.com/vitejs/vite/discussions/13835)
- **Type:**

```js
import type {
  CSSModulesConfig,
  Drafts,
  Features,
  NonStandard,
  PseudoClasses,
  Targets,
} from 'lightningcss'
```

```js
{
  targets?: Targets
  include?: Features
  exclude?: Features
  drafts?: Drafts
  nonStandard?: NonStandard
  pseudoClasses?: PseudoClasses
  unusedSymbols?: string[]
  cssModules?: CSSModulesConfig,
  // ...
}
```

Configures Lightning CSS. Full transform options can be found in [the Lightning CSS repo](https://github.com/parcel-bundler/lightningcss/blob/master/node/index.d.ts).

## json.namedExports

- **Type:** `boolean`
- **Default:** `true`

Whether to support named imports from `.json` files.

## json.stringify

- **Type:** `boolean`
- **Default:** `false`

If set to `true`, imported JSON will be transformed into `export default JSON.parse("...")` which is significantly more performant than Object literals, especially when the JSON file is large.

Enabling this disables named imports.

## esbuild

- **Type:** `ESBuildOptions | false`

`ESBuildOptions` extends [esbuild's own transform options](https://esbuild.github.io/api/#transform). The most common use case is customizing JSX:

```js
export default defineConfig({
  esbuild: {
    jsxFactory: 'h',
    jsxFragment: 'Fragment',
  },
})
```

By default, esbuild is applied to `ts`, `jsx` and `tsx` files. You can customize this with `esbuild.include` and `esbuild.exclude`, which can be a regex, a [picomatch](https://github.com/micromatch/picomatch#globbing-features) pattern, or an array of either.

In addition, you can also use `esbuild.jsxInject` to automatically inject JSX helper imports for every file transformed by esbuild:

```js
export default defineConfig({
  esbuild: {
    jsxInject: `import React from 'react'`,
  },
})
```

When [`build.minify`](./build-options.md#build-minify) is `true`, all minify optimizations are applied by default. To disable [certain aspects](https://esbuild.github.io/api/#minify) of it, set any of `esbuild.minifyIdentifiers`, `esbuild.minifySyntax`, or `esbuild.minifyWhitespace` options to `false`. Note the `esbuild.minify` option can't be used to override `build.minify`.

Set to `false` to disable esbuild transforms.

## assetsInclude

- **Type:** `string | RegExp | (string | RegExp)[]`
- **Related:** [Static Asset Handling](/guide/assets)

Specify additional [picomatch patterns](https://github.com/micromatch/picomatch#globbing-features) to be treated as static assets so that:

- They will be excluded from the plugin transform pipeline when referenced from HTML or directly requested over `fetch` or XHR.

- Importing them from JS will return their resolved URL string (this can be overwritten if you have a `enforce: 'pre'` plugin to handle the asset type differently).

The built-in asset type list can be found [here](https://github.com/vitejs/vite/blob/main/packages/vite/src/node/constants.ts).

Example:

```js
export default defineConfig({
  assetsInclude: ['**/*.gltf'],
})
```

## logLevel

- **Type:** `'info' | 'warn' | 'error' | 'silent'`

Adjust console output verbosity. Default is `'info'`.

## customLogger

- **Type:**
  ```ts
  interface Logger {
    info(msg: string, options?: LogOptions): void
    warn(msg: string, options?: LogOptions): void
    warnOnce(msg: string, options?: LogOptions): void
    error(msg: string, options?: LogErrorOptions): void
    clearScreen(type: LogType): void
    hasErrorLogged(error: Error | RollupError): boolean
    hasWarned: boolean
  }
  ```

Use a custom logger to log messages. You can use Vite's `createLogger` API to get the default logger and customize it to, for example, change the message or filter out certain warnings.

```ts twoslash
import { createLogger, defineConfig } from 'vite'

const logger = createLogger()
const loggerWarn = logger.warn

logger.warn = (msg, options) => {
  // Ignore empty CSS files warning
  if (msg.includes('vite:css') && msg.includes(' is empty')) return
  loggerWarn(msg, options)
}

export default defineConfig({
  customLogger: logger,
})
```

## clearScreen

- **Type:** `boolean`
- **Default:** `true`

Set to `false` to prevent Vite from clearing the terminal screen when logging certain messages. Via command line, use `--clearScreen false`.

## envDir

- **Type:** `string`
- **Default:** `root`

The directory from which `.env` files are loaded. Can be an absolute path, or a path relative to the project root.

See [here](/guide/env-and-mode#env-files) for more about environment files.

## envPrefix

- **Type:** `string | string[]`
- **Default:** `VITE_`

Env variables starting with `envPrefix` will be exposed to your client source code via import.meta.env.

:::warning SECURITY NOTES
`envPrefix` should not be set as `''`, which will expose all your env variables and cause unexpected leaking of sensitive information. Vite will throw an error when detecting `''`.

If you would like to expose an unprefixed variable, you can use [define](#define) to expose it:

```js
define: {
  'import.meta.env.ENV_VARIABLE': JSON.stringify(process.env.ENV_VARIABLE)
}
```

:::

## appType

- **Type:** `'spa' | 'mpa' | 'custom'`
- **Default:** `'spa'`

Whether your application is a Single Page Application (SPA), a [Multi Page Application (MPA)](../guide/build#multi-page-app), or Custom Application (SSR and frameworks with custom HTML handling):

- `'spa'`: include HTML middlewares and use SPA fallback. Configure [sirv](https://github.com/lukeed/sirv) with `single: true` in preview
- `'mpa'`: include HTML middlewares
- `'custom'`: don't include HTML middlewares

Learn more in Vite's [SSR guide](/guide/ssr#vite-cli). Related: [`server.middlewareMode`](./server-options#server-middlewaremode).

```
---
title: Vite 5.0 is out!
author:
  name: The Vite Team
date: 2023-11-16
sidebar: false
head:
  - - meta
    - property: og:type
      content: website
  - - meta
    - property: og:title
      content: Announcing Vite 5
  - - meta
    - property: og:image
      content: https://vitejs.dev/og-image-announcing-vite5.png
  - - meta
    - property: og:url
      content: https://vitejs.dev/blog/announcing-vite5
  - - meta
    - property: og:description
      content: Vite 5 Release Announcement
  - - meta
    - name: twitter:card
      content: summary_large_image
---
```

# Vite 5.0 is out!

November 16, 2023

![Vite 5 Announcement Cover Image](/og-image-announcing-vite5.png)

Vite 4 [was released](./announcing-vite4.md) almost a year ago, and it served as a solid base for the ecosystem. npm downloads per week jumped from 2.5 million to 7.5 million, as projects keep building on a shared infrastructure. Frameworks continued to innovate, and on top of [Astro](https://astro.build/), [Nuxt](https://nuxt.com/), [SvelteKit](https://kit.svelte.dev/), [Solid Start](https://www.solidjs.com/blog/introducing-solidstart), [Qwik City](https://qwik.builder.io/qwikcity/overview/), between others, we saw new frameworks joining and making the ecosystem stronger. [RedwoodJS](https://redwoodjs.com/) and [Remix](https://remix.run/) switching to Vite paves the way for further adoption in the React ecosystem. [Vitest](https://vitest.dev) kept growing at an even faster pace than Vite. Its team has been hard at work and will soon [release Vitest 1.0](https://github.com/vitest-dev/vitest/issues/3596). The story of Vite when used with other tools such as [Storybook](https://storybook.js.org), [Nx](https://nx.dev), and [Playwright](https://playwright.dev) kept improving, and the same goes for environments, with Vite dev working both in [Deno](https://deno.com) and [Bun](https://bun.sh).

We had the second edition of [ViteConf](https://viteconf.org/23/replay) a month ago, hosted by [StackBlitz](https://stackblitz.com). Like last year, most of the projects in the ecosystem got together to share ideas and connect to keep expanding the commons. We're also seeing new pieces complement the meta-framework tool belt like [Volar](https://volarjs.dev/) and [Nitro](https://nitro.unjs.io/). The Rollup team released [Rollup 4](https://rollupjs.org) that same day, a tradition Lukas started last year.

Six months ago, Vite 4.3 [was released](./announcing-vite4.md). This release significantly improved the dev server performance. However, there is still ample room for improvement. At ViteConf, [Evan You unveiled Vite's long-term plan to work on Rolldown](https://www.youtube.com/watch?v=hrdwQHoAp0M), a Rust-port of Rollup with compatible APIs. Once it is ready, we intend to use it in Vite Core to take on the tasks of both Rollup and esbuild. This will mean a boost in build performance (and later on in dev performance too as we move perf-sensitive parts of Vite itself to Rust), and a big reduction of inconsistencies between dev and build. Rolldown is currently in early stages and the team is preparing to open source the codebase before the end of the year. Stay tuned!

Today, we mark another big milestone in Vite's path. The Vite [team](/team), [contributors](https://github.com/vitejs/vite/graphs/contributors), and ecosystem partners, are excited to announce the release of Vite 5. Vite is now using [Rollup 4](https://github.com/vitejs/vite/pull/14508), which already represents a big boost in build performance. And there are also new options to improve your dev server performance profile.

Vite 5 focuses on cleaning up the API (removing deprecated features) and streamlines several features closing long-standing issues, for example switching `define` to use proper AST replacements instead of regexes. We also continue to take steps to future-proof Vite (Node.js 18+ is now required, and [the CJS Node API has been deprecated](/guide/migration#deprecate-cjs-node-api)).

Quick links:

- [Docs](/)
- [Migration Guide](/guide/migration)
- [Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#500-2023-11-16)

Docs in other languages:

- [ç®€ä½“ä¸æ–‡](https://cn.vitejs.dev/)
- [æ—¥æœ¬èªž](https://ja.vitejs.dev/)
- [EspaÃ±ol](https://es.vitejs.dev/)
- [PortuguÃªs](https://pt.vitejs.dev/)
- [í•œêµì–´](https://ko.vitejs.dev/)
- [Deutsch](https://de.vitejs.dev/) (new translation!)

If you're new to Vite, we suggest reading first the [Getting Started](/guide/) and [Features](/guide/features) guides.

We appreciate the more than [850 contributors to Vite Core](https://github.com/vitejs/vite/graphs/contributors), and the maintainers and contributors of Vite plugins, integrations, tools, and translations that have helped us reach here. We encourage you to get involved and continue to improve Vite with us. You can learn more at our [Contributing Guide](https://github.com/vitejs/vite/blob/main/CONTRIBUTING.md). To get started, we recommend [triaging issues](https://github.com/vitejs/vite/issues), [reviewing PRs](https://github.com/vitejs/vite/pulls), sending failing tests PRs based on open issues, and helping others in [Discussions](https://github.com/vitejs/vite/discussions) and Vite Land's [help forum](https://discord.com/channels/804011606160703521/1019670660856942652). You'll learn a lot along the way and have a smooth path to further contributions to the project. If you have doubts, join us on our [Discord community](http://chat.vitejs.dev/) and say hi on the [#contributing channel](https://discord.com/channels/804011606160703521/804439875226173480).

To stay up to date, follow us on [X](https://twitter.com/vite_js) or [Mastodon](https://webtoo.ls/@vite).

## Quick start with Vite 5

Use `pnpm create vite` to scaffold a Vite project with your preferred framework, or open a started template online to play with Vite 5 using vite.new. You can also run `pnpm create vite-extra` to get access to templates from other frameworks and runtimes (Solid, Deno, SSR, and library starters). `create vite-extra` templates are also available when you run `create vite` under the `Others` option.

Note that Vite starter templates are intended to be used as a playground to test Vite with different frameworks. When building your next project, we recommend reaching out to the starters recommended by each framework. Some frameworks now redirect in `create vite` to their starters too (`create-vue` and `Nuxt 3` for Vue, and `SvelteKit` for Svelte).

## Node.js Support

Vite no longer supports Node.js 14 / 16 / 17 / 19, which reached its EOL. Node.js 18 / 20+ is now required.

## Performance

On top of Rollup 4's build performance improvements, there is a new guide to help you identify and fix common performance issues at [https://vitejs.dev/guide/performance](/guide/performance).

Vite 5 also introduces [server.warmup](/guide/performance.html#warm-up-frequently-used-files), a new feature to improve startup time. It lets you define a list of modules that should be pre-transformed as soon as the server starts. When using [`--open` or `server.open`](/config/server-options.html#server-open), Vite will also automatically warm up the entry point of your app or the provided URL to open.

## Main Changes

- [Vite is now powered by Rollup 4](/guide/migration#rollup-4)
- [The CJS Node API has been deprecated](/guide/migration#deprecate-cjs-node-api)
- [Rework `define` and `import.meta.env.*` replacement strategy](/guide/migration#rework-define-and-import-meta-env-replacement-strategy)
- [SSR externalized modules value now matches production](/guide/migration#ssr-externalized-modules-value-now-matches-production)
- [`worker.plugins` is now a function](/guide/migration#worker-plugins-is-now-a-function)
- [Allow path containing `.` to fallback to index.html](/guide/migration#allow-path-containing-to-fallback-to-index-html)
- [Align dev and preview HTML serving behavior](/guide/migration#align-dev-and-preview-html-serving-behaviour)
- [Manifest files are now generated in `.vite` directory by default](/guide/migration#manifest-files-are-now-generated-in-vite-directory-by-default)
- [CLI shortcuts require an additional `Enter` press](/guide/migration#cli-shortcuts-require-an-additional-enter-press)
- [Update `experimentalDecorators` and `useDefineForClassFields` TypeScript behavior](/guide/migration#update-experimentaldecorators-and-usedefineforclassfields-typescript-behaviour)
- [Remove `--https` flag and `https: true`](/guide/migration#remove-https-flag-and-https-true)
- [Remove `resolvePackageEntry` and `resolvePackageData` APIs](/guide/migration#remove-resolvepackageentry-and-resolvepackagedata-apis)
- [Removes previously deprecated APIs](/guide/migration#removed-deprecated-apis)
- [Read more about advanced changes affecting plugin and tool authors](/guide/migration#advanced)

## Migrating to Vite 5

We have worked with ecosystem partners to ensure a smooth migration to this new major. Once again, [vite-ecosystem-ci](https://www.youtube.com/watch?v=7L4I4lDzO48) has been crucial to help us make bolder changes while avoiding regressions. We're thrilled to see other ecosystems adopt similar schemes to improve the collaboration between their projects and downstream maintainers.

For most projects, the update to Vite 5 should be straight forward. But we advise reviewing the [detailed Migration Guide](/guide/migration) before upgrading.

A low level breakdown with the full list of changes to Vite core can be found at the [Vite 5 Changelog](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md#500-2023-11-16).

## Acknowledgments

Vite 5 is the result of long hours of work by our community of contributors, downstream maintainers, plugins authors, and the [Vite Team](/team). A big shout out to [Bjorn Lu](https://twitter.com/bluwyoo) for leading the release process for this major.

We're also thankful to individuals and companies sponsoring Vite development. [StackBlitz](https://stackblitz.com/), [Nuxt

Labs](https://nuxtlabs.com/), and [Astro](https://astro.build) continue to invest in Vite by hiring Vite team members. A shout out to sponsors on [Vite's GitHub Sponsors](https://github.com/sponsors/vitejs), [Vite's Open Collective](https://opencollective.com/vite), and [Evan You's GitHub Sponsors](https://github.com/sponsors/yyx990803). A special mention to [Remix](https://remix.run/) for becoming a Gold sponsor and contributing back after switching to Vite.

# SSR Options

## ssr.external

- **Type:** `string[] | true`
- **Related:** [SSR Externals](/guide/ssr#ssr-externals)

Externalize the given dependencies and their transitive dependencies for SSR. By default, all dependencies are externalized except for linked dependencies (for HMR). If you prefer to externalize the linked dependency, you can pass its name to this option.

If `true`, all dependencies including linked dependencies are externalized.

Note that the explicitly listed dependencies (using `string[]` type) will always take priority if they're also listed in `ssr.noExternal` (using any type).

## ssr.noExternal

- **Type:** `string | RegExp | (string | RegExp)[] | true`
- **Related:** [SSR Externals](/guide/ssr#ssr-externals)

Prevent listed dependencies from being externalized for SSR, which they will get bundled in build. By default, only linked dependencies are not externalized (for HMR). If you prefer to externalize the linked dependency, you can pass its name to the `ssr.external` option.

If `true`, no dependencies are externalized. However, dependencies explicitly listed in `ssr.external` (using `string[]` type) can take priority and still be externalized. If `ssr.target: 'node'` is set, Node.js built-ins will also be externalized by default.

Note that if both `ssr.noExternal: true` and `ssr.external: true` are configured, `ssr.noExternal` takes priority and no dependencies are externalized.

## ssr.target

- **Type:** `'node' | 'webworker'`
- **Default:** `node`

Build target for the SSR server.

## ssr.resolve.conditions

- **Type:** `string[]`
- **Related:** [Resolve Conditions](./shared-options.md#resolve-conditions)

Defaults to the root [`resolve.conditions`](./shared-options.md#resolve-conditions).

These conditions are used in the plugin pipeline, and only affect non-externalized dependencies during the SSR build. Use `ssr.resolve.externalConditions` to affect externalized imports.

## ssr.resolve.externalConditions

- **Type:** `string[]`
- **Default:** `[]`

Conditions that are used during ssr import (including `ssrLoadModule`) of externalized dependencies.

# Vite Runtime API

:::warning Low-level API
This API was introduced in Vite 5.1 as an experimental feature. It was added to [gather feedback](https://github.com/vitejs/vite/discussions/15774). There will likely be breaking changes, so make sure to pin the Vite version to `~5.1.0` when using it. This is a low-level API meant for library and framework authors. If your goal is to create an application, make sure to check out the higher-level SSR plugins and tools at [Awesome Vite SSR section](https://github.com/vitejs/awesome-vite#ssr) first.

Currently, the API is being revised as the [Environment API](https://github.com/vitejs/vite/discussions/16358) which is released at `^6.0.0-alpha.0`.
:::

The "Vite Runtime" is a tool that allows running any code by processing it with Vite plugins first. It is different from `server.ssrLoadModule` because the runtime implementation is decoupled from the server. This allows library and framework authors to implement their own layer of communication between the server and the runtime.

One of the goals of this feature is to provide a customizable API to process and run the code. Vite provides enough tools to use Vite Runtime out of the box, but users can build upon it if their needs do not align with Vite's built-in implementation.

All APIs can be imported from `vite/runtime` unless stated otherwise.

## `ViteRuntime`

Type Signature:

```ts
export class ViteRuntime {
  constructor(
    public options: ViteRuntimeOptions,
    public runner: ViteModuleRunner,
    private debug?: ViteRuntimeDebugger,
  ) {}
  /**
   * URL to execute. Accepts file path, server path, or id relative to the root.
   */
  public async executeUrl<T = any>(url: string): Promise<T>
  /**
   * Entry point URL to execute. Accepts file path, server path or id relative to the root.
   * In the case of a full reload triggered by HMR, this is the module that will be reloaded.
   * If this method is called multiple times, all entry points will be reloaded one at a time.
   */
  public async executeEntrypoint<T = any>(url: string): Promise<T>
  /**
   * Clear all caches including HMR listeners.
   */
  public clearCache(): void
  /**
   * Clears all caches, removes all HMR listeners, and resets source map support.
   * This method doesn't stop the HMR connection.
   */
  public async destroy(): Promise<void>
  /**
   * Returns `true` if the runtime has been destroyed by calling `destroy()` method.
   */
  public isDestroyed(): boolean
}
```

::: tip Advanced Usage
If you are just migrating from `server.ssrLoadModule` and want to support HMR, consider using [`createViteRuntime`](#createviteruntime) instead.
:::

The `ViteRuntime` class requires `root` and `fetchModule` options when initiated. Vite exposes `ssrFetchModule` on the [`server`](/guide/api-javascript) instance for easier integration with Vite SSR. Vite also exports `fetchModule` from its main entry point - it doesn't make any assumptions about how the code is running unlike `ssrFetchModule` that expects the code to run using `new Function`. This can be seen in source maps that these functions return.

Runner in `ViteRuntime` is responsible for executing the code. Vite exports `ESModulesRunner` out of the box, it uses `new AsyncFunction` to run the code. You can provide your own implementation if your JavaScript runtime doesn't support unsafe evaluation.

The two main methods that runtime exposes are `executeUrl` and `executeEntrypoint`. The only difference between them is that all modules executed by `executeEntrypoint` will be reexecuted if HMR triggers `full-reload` event. Be aware that Vite Runtime doesn't update `exports` object when this happens (it overrides it), you would need to run `executeUrl` or get the module from `moduleCache` again if you rely on having the latest `exports` object.

Example Usage:

```js
import { ViteRuntime, ESModulesRunner } from 'vite/runtime'
import { root, fetchModule } from './rpc-implementation.js'
```

~~~~
const runtime = new ViteRuntime(
  {
    root,
    fetchModule,
    // you can also provide hmr.connection to support HMR
  },
  new ESModulesRunner(),
)

await runtime.executeEntrypoint('/src/entry-point.js')
```

## `ViteRuntimeOptions`

```ts
export interface ViteRuntimeOptions {
  /**
   * Root of the project
   */
  root: string
  /**
   * A method to get the information about the module.
   * For SSR, Vite exposes `server.ssrFetchModule` function that you can use here.
   * For other runtime use cases, Vite also exposes `fetchModule` from its main entry point.
   */
  fetchModule: FetchFunction
  /**
   * 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.
   */
  hmr?:
    | false
    | {
        /**
         * Configure how HMR communicates between the client and the server.
         */
        connection: HMRRuntimeConnection
        /**
         * Configure HMR logger.
         */
        logger?: false | HMRLogger
      }
  /**
   * Custom module cache. If not provided, it creates a separate module cache for each ViteRuntime instance.
   */
  moduleCache?: ModuleCacheMap
}
```

## `ViteModuleRunner`

**Type Signature:**

```ts
export interface ViteModuleRunner {
  /**
   * Run code that was transformed by Vite.
   * @param context Function context
   * @param code Transformed code
   * @param id ID that was used to fetch the module
   */
  runViteModule(
    context: ViteRuntimeModuleContext,
    code: string,
    id: string,
  ): Promise<any>
  /**
   * Run externalized module.
   * @param file File URL to the external module
   */
  runExternalModule(file: string): Promise<any>
}
~~~~

Vite exports `ESModulesRunner` that implements this interface by default. It uses `new AsyncFunction` to run code, so if the code has inlined source map it should contain an [offset of 2 lines](https://tc39.es/ecma262/#sec-createdynamicfunction) to accommodate for new lines added. This is done automatically by `server.ssrFetchModule`. If your runner implementation doesn't have this constraint, you should use `fetchModule` (exported from `vite`) directly.

## HMRRuntimeConnection

Type Signature:

```ts
export interface HMRRuntimeConnection {
  /**
   * Checked before sending messages to the client.
   */
  isReady(): boolean
  /**
   * Send message to the client.
   */
  send(message: string): void
  /**
   * Configure how HMR is handled when this connection triggers an update.
   * This method expects that connection will start listening for HMR updates and call this callback when it's received.
   */
  onUpdate(callback: (payload: HMRPayload) => void): void
}
```

This interface defines how HMR communication is established. Vite exports `ServerHMRConnector` from the main entry point to support HMR during Vite SSR. The `isReady` and `send` methods are usually called when the custom event is triggered (like, `import.meta.hot.send("my-event")`).

`onUpdate` is called only once when the new runtime is initiated. It passed down a method that should be called when connection triggers the HMR event. The implementation depends on the type of connection (as an example, it can be `WebSocket`/`EventEmitter`/`MessageChannel`), but it usually looks something like this:

```js
function onUpdate(callback) {
  this.connection.on('hmr', (event) => callback(event.data))
}
```

The callback is queued and it will wait for the current update to be resolved before processing the next update. Unlike the browser implementation, HMR updates in Vite Runtime wait until all listeners (like, `vite:beforeUpdate`/`vite:beforeFullReload`) are finished before updating the modules.

## `createViteRuntime`

Type Signature:

```ts
async function createViteRuntime(
  server: ViteDevServer,
  options?: MainThreadRuntimeOptions,
): Promise<ViteRuntime>
```

Example Usage:

```js
import { createServer } from 'vite'

const __dirname = fileURLToPath(new URL('.', import.meta.url))

;(async () => {
  const server = await createServer({
    root: __dirname,
  })
  await server.listen()

  const runtime = await createViteRuntime(server)
  await runtime.executeEntrypoint('/src/entry-point.js')
})()
```

This method serves as an easy replacement for `server.ssrLoadModule`. Unlike `ssrLoadModule`, `createViteRuntime` provides HMR support out of the box. You can pass down [`options`](#mainthreadruntimeoptions) to customize how SSR runtime behaves to suit your needs.

## `MainThreadRuntimeOptions`

```ts
```

```ts
export interface MainThreadRuntimeOptions
  extends Omit<ViteRuntimeOptions, 'root' | 'fetchModule' | 'hmr'> {
  /**
   * Disable HMR or configure HMR logger.
   */
  hmr?:
    | false
    | {
        logger?: false | HMRLogger
      }
  /**
   * Provide a custom module runner. This controls how the code is executed.
   */
  runner?: ViteModuleRunner
}
```

# Static Asset Handling

- Related: [Public Base Path](./build#public-base-path)
- Related: [`assetsInclude` config option](/config/shared-options.md#assetsinclude)

## Importing Asset as URL

Importing a static asset will return the resolved public URL when it is served:

```js twoslash
import 'vite/client'
// ---cut---
import imgUrl from './img.png'
document.getElementById('hero-img').src = imgUrl
```

For example, `imgUrl` will be `/img.png` during development, and become `/assets/img.2d8efhg.png` in the production build.

The behavior is similar to webpack's `file-loader`. The difference is that the import can be either using absolute public paths (based on project root during dev) or relative paths.

- `url()` references in CSS are handled the same way.

- If using the Vue plugin, asset references in Vue SFC templates are automatically converted into imports.

- Common image, media, and font filetypes are detected as assets automatically. You can extend the internal list using the [`assetsInclude` option](/config/shared-options.md#assetsinclude).

- Referenced assets are included as part of the build assets graph, will get hashed file names, and can be processed by plugins for optimization.

- Assets smaller in bytes than the [`assetsInlineLimit` option](/config/build-options.md#build-assetsinlinelimit) will be inlined as base64 data URLs.

- Git LFS placeholders are automatically excluded from inlining because they do not contain the content of the file they represent. To get inlining, make sure to download the file contents via Git LFS before building.

- TypeScript, by default, does not recognize static asset imports as valid modules. To fix this, include [`vite/client`](./features#client-types).

::: tip Inlining SVGs through `url()`
When passing a URL of SVG to a manually constructed `url()` by JS, the variable should be wrapped within double quotes.

```js twoslash
import 'vite/client'
// ---cut---
import imgUrl from './img.svg'
document.getElementById('hero-img').style.background = `url("${imgUrl}")`
```

:::

### Explicit URL Imports

Assets that are not included in the internal list or in `assetsInclude`, can be explicitly imported as a URL using the `?url` suffix. This is useful, for example, to import [Houdini Paint Worklets](https://houdini.how/usage).

```js twoslash
import 'vite/client'
// ---cut---
import workletURL from 'extra-scalloped-border/worklet.js?url'
CSS.paintWorklet.addModule(workletURL)
```

### Importing Asset as String

Assets can be imported as strings using the `?raw` suffix.

```js twoslash
import 'vite/client'
// ---cut---
import shaderString from './shader.glsl?raw'
```

### Importing Script as a Worker

Scripts can be imported as web workers with the `?worker` or `?sharedworker` suffix.

```js twoslash
import 'vite/client'
// ---cut---
// Separate chunk in the production build
import Worker from './shader.js?worker'
```

const worker = new Worker()

```js twoslash
import 'vite/client'
// ---cut---
// sharedworker
import SharedWorker from './shader.js?sharedworker'
const sharedWorker = new SharedWorker()
```

```js twoslash
import 'vite/client'
// ---cut---
// Inlined as base64 strings
import InlineWorker from './shader.js?worker&inline'
```

Check out the [Web Worker section](./features.md#web-workers) for more details.

## The `public` Directory

If you have assets that are:

- Never referenced in source code (e.g. `robots.txt`)
- Must retain the exact same file name (without hashing)
- ...or you simply don't want to have to import an asset first just to get its URL

Then you can place the asset in a special `public` directory under your project root. Assets in this directory will be served at root path `/` during dev, and copied to the root of the dist directory as-is.

The directory defaults to `<root>/public`, but can be configured via the [`publicDir` option](/config/shared-options.md#publicdir).

Note that:

- You should always reference `public` assets using root absolute path - for example, `public/icon.png` should be referenced in source code as `/icon.png`.
- Assets in `public` cannot be imported from JavaScript.

## new URL(url, import.meta.url)

[import.meta.url](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import.meta) is a native ESM feature that exposes the current module's URL. Combining it with the native [URL constructor](https://developer.mozilla.org/en-US/docs/Web/API/URL), we can obtain the full, resolved URL of a static asset using relative path from a JavaScript module:

```js
const imgUrl = new URL('./img.png', import.meta.url).href

document.getElementById('hero-img').src = imgUrl
```

This works natively in modern browsers - in fact, Vite doesn't need to process this code at all during development!

This pattern also supports dynamic URLs via template literals:

```js
function getImageUrl(name) {
  return new URL(`./dir/${name}.png`, import.meta.url).href
}
```

During the production build, Vite will perform necessary transforms so that the URLs still point to the correct location even after bundling and asset hashing. However, the URL string must be static so it can be analyzed, otherwise the code will be left as is, which can cause runtime errors if `build.target` does not support `import.meta.url`

```js
// Vite will not transform this
const imgUrl = new URL(imagePath, import.meta.url).href
```

::: warning Does not work with SSR
This pattern does not work if you are using Vite for Server-Side Rendering, because `import.meta.url` have different semantics in browsers vs. Node.js. The server bundle also cannot determine the client host URL ahead of time.
:::

# Worker Options

Options related to Web Workers.

## worker.format

- **Type:** `'es' | 'iife'`
- **Default:** `'iife'`

Output format for worker bundle.

## worker.plugins

- **Type:** [`() => (Plugin | Plugin[])[]`](./shared-options#plugins)

Vite plugins that apply to the worker bundles. Note that [config.plugins](./shared-options#plugins) only applies to workers in dev, it should be configured here instead for build.
The function should return new plugin instances as they are used in parallel rollup worker builds. As such, modifying `config.worker` options in the `config` hook will be ignored.

## worker.rollupOptions

- **Type:** [`RollupOptions`](https://rollupjs.org/configuration-options/)

Rollup options to build worker bundle.

Backend Integration

:::tip Note
If you want to serve the HTML using a traditional backend (e.g. Rails, Laravel) but use Vite for serving assets, check for existing integrations listed in [Awesome Vite](https://github.com/vitejs/awesome-vite#integrations-with-backends).

If you need a custom integration, you can follow the steps in this guide to configure it manually
:::

1. In your Vite config, configure the entry and enable build manifest:

   ```js twoslash
   import { defineConfig } from 'vite'
   // ---cut---
   // vite.config.js
   export default defineConfig({
     build: {
       // generate .vite/manifest.json in outDir
       manifest: true,
       rollupOptions: {
         // overwrite default .html entry
         input: '/path/to/main.js',
       },
     },
   })
   ```

   If you haven't disabled the [module preload polyfill](/config/build-options.md#build-polyfillmodulepreload), you also need to import the polyfill in your entry

   ```js
   // add the beginning of your app entry
   import 'vite/modulepreload-polyfill'
   ```

2. For development, inject the following in your server's HTML template (substitute `http://localhost:5173` with the local URL Vite is running at):

   ```html
   <!-- if development -->
   <script type="module" src="http://localhost:5173/@vite/client"></script>
   <script type="module" src="http://localhost:5173/main.js"></script>
   ```

   In order to properly serve assets, you have two options:

   - Make sure the server is configured to proxy static assets requests to the Vite server
   - Set [`server.origin`](/config/server-options.md#server-origin) so that generated asset URLs will be resolved using the back-end server URL instead of a relative path

   This is needed for assets such as images to load properly.

   Note if you are using React with `@vitejs/plugin-react`, you'll also need to add this before the above scripts, since the plugin is not able to modify the HTML you are serving (substitute `http://localhost:5173` with the local URL Vite is running at):

   ```html
   <script type="module">
     import RefreshRuntime from 'http://localhost:5173/@react-refresh'
     RefreshRuntime.injectIntoGlobalHook(window)
     window.$RefreshReg$ = () => {}
     window.$RefreshSig$ = () => (type) => type
     window.__vite_plugin_react_preamble_installed__ = true
   </script>
   ```

3. For production: after running `vite build`, a `.vite/manifest.json` file will be generated alongside other asset files. An example manifest file looks like this:

   ```json
   {
     "_shared-!~{003}~.js": {
       "file": "assets/shared-ChJ_j-JJ.css",
       "src": "_shared-!~{003}~.js"
     },
     "_shared-B7PI925R.js": {
       "file": "assets/shared-B7PI925R.js",
       "name": "shared",
       "css": ["assets/shared-ChJ_j-JJ.css"]
     },
     "baz.js": {
       "file": "assets/baz-B2H3sXNv.js",
       "name": "baz",
   ```

~~~text
        "src": "baz.js",
        "isDynamicEntry": true
      },
      "views/bar.js": {
        "file": "assets/bar-gkvgaI9m.js",
        "name": "bar",
        "src": "views/bar.js",
        "isEntry": true,
        "imports": ["_shared-B7PI925R.js"],
        "dynamicImports": ["baz.js"]
      },
      "views/foo.js": {
        "file": "assets/foo-BRBmoGS9.js",
        "name": "foo",
        "src": "views/foo.js",
        "isEntry": true,
        "imports": ["_shared-B7PI925R.js"],
        "css": ["assets/foo-5UjPuW-k.css"]
      }
    }
    ```

    - The manifest has a `Record<name, chunk>` structure
    - For entry or dynamic entry chunks, the key is the relative src path from project root.
    - For non entry chunks, the key is the base name of the generated file prefixed with `_`.
    - Chunks will contain information on its static and dynamic imports (both are keys that map to the corresponding chunk in
the manifest), and also its corresponding CSS and asset files (if any).

4. You can use this file to render links or preload directives with hashed filenames.

   Here is an example HTML template to render the proper links. The syntax here is for
   explanation only, substitute with your server templating language. The `importedChunks`
   function is for illustration and isn't provided by Vite.

   ```html
   <!-- if production -->

   <!-- for cssFile of manifest[name].css -->
   <link rel="stylesheet" href="/{{ cssFile }}" />

   <!-- for chunk of importedChunks(manifest, name) -->
   <!-- for cssFile of chunk.css -->
   <link rel="stylesheet" href="/{{ cssFile }}" />

   <script type="module" src="/{{ manifest[name].file }}"></script>

   <!-- for chunk of importedChunks(manifest, name) -->
   <link rel="modulepreload" href="/{{ chunk.file }}" />
   ```

   Specifically, a backend generating HTML should include the following tags given a manifest
   file and an entry point:

   - A `<link rel="stylesheet">` tag for each file in the entry point chunk's `css` list
   - Recursively follow all chunks in the entry point's `imports` list and include a
     `<link rel="stylesheet">` tag for each CSS file of each imported chunk.
   - A tag for the `file` key of the entry point chunk (`<script type="module">` for JavaScript,
     or `<link rel="stylesheet">` for CSS)
   - Optionally, `<link rel="modulepreload">` tag for the `file` of each imported JavaScript
     chunk, again recursively following the imports starting from the entry point chunk.

   Following the above example manifest, for the entry point `views/foo.js` the following tags should be included in
production:

   ```html
   <link rel="stylesheet" href="assets/foo-5UjPuW-k.css" />
   <link rel="stylesheet" href="assets/shared-ChJ_j-JJ.css" />
   <script type="module" src="assets/foo-BRBmoGS9.js"></script>
   <!-- optional -->
   <link rel="modulepreload" href="assets/shared-B7PI925R.js" />
   ```

   While the following should be included for the entry point `views/bar.js`:

   ```html
   <link rel="stylesheet" href="assets/shared-ChJ_j-JJ.css" />
   <script type="module" src="assets/bar-gkvgaI9m.js"></script>
   <!-- optional -->
   <link rel="modulepreload" href="assets/shared-B7PI925R.js" />
   ```
~~~

```
---
layout: home

title: Vite
titleTemplate: Next Generation Frontend Tooling

hero:
  name: Vite
  text: Next Generation Frontend Tooling
  tagline: Get ready for a development environment that can finally catch up with you.
  image:
    src: /logo-with-shadow.png
    alt: Vite
  actions:
    - theme: brand
      text: Get Started
      link: /guide/
    - theme: alt
      text: Why Vite?
      link: /guide/why
    - theme: alt
      text: View on GitHub
      link: https://github.com/vitejs/vite
    - theme: brand
      text: âš¡ ViteConf 24!
      link: https://viteconf.org/?utm=vite-homepage

features:
  - icon: ðŸ'¡
    title: Instant Server Start
    details: On demand file serving over native ESM, no bundling required!
  - icon: âš¡ï¸
    title: Lightning Fast HMR
    details: Hot Module Replacement (HMR) that stays fast regardless of app size.
  - icon: ðŸ› ï¸
    title: Rich Features
    details: Out-of-the-box support for TypeScript, JSX, CSS and more.
  - icon: ðŸ"¦
    title: Optimized Build
    details: Pre-configured Rollup build with multi-page and library mode support.
  - icon: ðŸ"©
    title: Universal Plugins
    details: Rollup-superset plugin interface shared between dev and build.
  - icon: ðŸ"'
    title: Fully Typed APIs
    details: Flexible programmatic APIs with full TypeScript typing.
---

<script setup>
import { onMounted } from 'vue'

onMounted(() => {
  const urlParams = new URLSearchParams(window.location.search)
  if (urlParams.get('uwu') != null) {
    const img = document.querySelector('.VPHero .VPImage.image-src')
    img.src = '/logo-uwu.png'
    img.alt = 'Vite Kawaii Logo by @icarusgkx'
  }
})
</script>
```

# Plugins

:::tip NOTE
Vite aims to provide out-of-the-box support for common web development patterns. Before searching for a Vite or Compatible Rollup plugin, check out the [Features Guide](../guide/features.md). A lot of the cases where a plugin would be needed in a Rollup project are already covered in Vite.
:::

Check out [Using Plugins](../guide/using-plugins) for information on how to use plugins.

## Official Plugins

### [@vitejs/plugin-vue](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue)

- Provides Vue 3 Single File Components support.

### [@vitejs/plugin-vue-jsx](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue-jsx)

- Provides Vue 3 JSX support (via [dedicated Babel transform](https://github.com/vuejs/jsx-next)).

### [@vitejs/plugin-vue2](https://github.com/vitejs/vite-plugin-vue2)

- Provides Vue 2.7 Single File Components support.

### [@vitejs/plugin-vue2-jsx](https://github.com/vitejs/vite-plugin-vue2-jsx)

- Provides Vue 2.7 JSX support (via [dedicated Babel transform](https://github.com/vuejs/jsx-vue2/)).

### [@vitejs/plugin-react](https://github.com/vitejs/vite-plugin-react/tree/main/packages/plugin-react)

- Uses esbuild and Babel, achieving fast HMR with a small package footprint and the flexibility of being able to use the Babel transform pipeline. Without additional Babel plugins, only esbuild is used during builds.

### [@vitejs/plugin-react-swc](https://github.com/vitejs/vite-plugin-react-swc)

- Replaces Babel with SWC during development. During builds, SWC+esbuild are used when using plugins, and esbuild only otherwise. For big projects that don't require non-standard React extensions, cold start and Hot Module Replacement (HMR) can be significantly faster.

### [@vitejs/plugin-legacy](https://github.com/vitejs/vite/tree/main/packages/plugin-legacy)

- Provides legacy browsers support for the production build.

## Community Plugins

Check out [awesome-vite](https://github.com/vitejs/awesome-vite#plugins) - you can also submit a PR to list your plugins there.

## Rollup Plugins

[Vite plugins](../guide/api-plugin) are an extension of Rollup's plugin interface. Check out the [Rollup Plugin Compatibility section](../guide/api-plugin#rollup-plugin-compatibility) for more information.

# Building for Production

When it is time to deploy your app for production, simply run the `vite build` command. By default, it uses `<root>/index.html` as the build entry point, and produces an application bundle that is suitable to be served over a static hosting service. Check out the [Deploying a Static Site](./static-deploy) for guides about popular services.

## Browser Compatibility

The production bundle assumes support for modern JavaScript. By default, Vite targets browsers which support the [native ES Modules](https://caniuse.com/es6-module), [native ESM dynamic import](https://caniuse.com/es6-module-dynamic-import), and [`import.meta`](https://caniuse.com/mdn-javascript_operators_import_meta):

- Chrome >=87
- Firefox >=78
- Safari >=14
- Edge >=88

You can specify custom targets via the [`build.target` config option](/config/build-options.md#build-target), where the lowest target is `es2015`.

Note that by default, Vite only handles syntax transforms and **does not cover polyfills**. You can check out https://cdnjs.cloudflare.com/polyfill/ which automatically generates polyfill bundles based on the user's browser UserAgent string.

Legacy browsers can be supported via [@vitejs/plugin-legacy](https://github.com/vitejs/vite/tree/main/packages/plugin-legacy), which will automatically generate legacy chunks and corresponding ES language feature polyfills. The legacy chunks are conditionally loaded only in browsers that do not have native ESM support.

## Public Base Path

- Related: [Asset Handling](./assets)

If you are deploying your project under a nested public path, simply specify the [`base` config option](/config/shared-options.md#base) and all asset paths will be rewritten accordingly. This option can also be specified as a command line flag, e.g. `vite build --base=/my/public/path/`.

JS-imported asset URLs, CSS `url()` references, and asset references in your `.html` files are all automatically adjusted to respect this option during build.

The exception is when you need to dynamically concatenate URLs on the fly. In this case, you can use the globally injected `import.meta.env.BASE_URL` variable which will be the public base path. Note this variable is statically replaced during build so it must appear exactly as-is (i.e. `import.meta.env['BASE_URL']` won't work).

For advanced base path control, check out [Advanced Base Options](#advanced-base-options).

## Customizing the Build

The build can be customized via various [build config options](/config/build-options.md). Specifically, you can directly adjust the underlying [Rollup options](https://rollupjs.org/configuration-options/) via `build.rollupOptions`:

```js
export default defineConfig({
  build: {
    rollupOptions: {
      // https://rollupjs.org/configuration-options/
    },
  },
})
```

For example, you can specify multiple Rollup outputs with plugins that are only applied during build.

## Chunking Strategy

You can configure how chunks are split using `build.rollupOptions.output.manualChunks` (see [Rollup docs](https://rollupjs.org/configuration-options/#output-manualchunks)). If you use a framework, refer to their documentation for configuring how chunks are splitted.

## Load Error Handling

Vite emits `vite:preloadError` event when it fails to load dynamic imports. `event.payload` contains the original import error. If you call `event.preventDefault()`, the error will not be thrown.

```js twoslash
window.addEventListener('vite:preloadError', (event) => {
  window.location.reload() // for example, refresh the page
})
```

When a new deployment occurs, the hosting service may delete the assets from previous deployments. As a result, a user who visited your site before the new deployment might encounter an import error. This error happens because the assets running on that user's device are outdated and it tries to import the corresponding old chunk, which is deleted. This event is useful for addressing this situation.

## Rebuild on files changes

You can enable rollup watcher with `vite build --watch`. Or, you can directly adjust the underlying [`WatcherOptions`](https://rollupjs.org/configuration-options/#watch) via `build.watch`:

```js
// vite.config.js
export default defineConfig({
  build: {
    watch: {
      // https://rollupjs.org/configuration-options/#watch
    },
  },
})
```

With the `--watch` flag enabled, changes to the `vite.config.js`, as well as any files to be bundled, will trigger a rebuild.

## Multi-Page App

Suppose you have the following source code structure:

```
â”œâ”€â”€ package.json
â”œâ”€â”€ vite.config.js
â”œâ”€â”€ index.html
â”œâ”€â”€ main.js
â””â”€â”€ nested
    â”œâ”€â”€ index.html
    â””â”€â”€ nested.js
```

During dev, simply navigate or link to `/nested/` - it works as expected, just like for a normal static file server.

During build, all you need to do is to specify multiple `.html` files as entry points:

```js twoslash
// vite.config.js
import { resolve } from 'path'
import { defineConfig } from 'vite'

export default defineConfig({
  build: {
    rollupOptions: {
      input: {
        main: resolve(__dirname, 'index.html'),
        nested: resolve(__dirname, 'nested/index.html'),
      },
    },
  },
})
```

If you specify a different root, remember that `__dirname` will still be the folder of your vite.config.js file when resolving the input paths. Therefore, you will need to add your `root` entry to the arguments for `resolve`.

Note that for HTML files, Vite ignores the name given to the entry in the `rollupOptions.input` object and instead respects the resolved id of the file when generating the HTML asset in the dist folder. This ensures a consistent structure with the way the dev server works.

## Library Mode

When you are developing a browser-oriented library, you are likely spending most of the time on a test/demo page that imports your actual library. With Vite, you can use your `index.html` for that purpose to get the smooth development experience.

When it is time to bundle your library for distribution, use the [`build.lib` config option](/config/build-options.md#build-lib). Make sure to also externalize any dependencies that you do not want to bundle into your library, e.g. `vue` or `react`:

```js twoslash
// vite.config.js
import { resolve } from 'path'
import { defineConfig } from 'vite'

export default defineConfig({
  build: {
    lib: {
      // Could also be a dictionary or array of multiple entry points
      entry: resolve(__dirname, 'lib/main.js'),
      name: 'MyLib',
      // the proper extensions will be added
      fileName: 'my-lib',
    },
```

~~~
    rollupOptions: {
      // make sure to externalize deps that shouldn't be bundled
      // into your library
      external: ['vue'],
      output: {
        // Provide global variables to use in the UMD build
        // for externalized deps
        globals: {
          vue: 'Vue',
        },
      },
    },
  },
})
```

The entry file would contain exports that can be imported by users of your package:

```js
// lib/main.js
import Foo from './Foo.vue'
import Bar from './Bar.vue'
export { Foo, Bar }
```

Running `vite build` with this config uses a Rollup preset that is oriented towards shipping libraries and produces two
bundle formats: `es` and `umd` (configurable via `build.lib`):

```
$ vite build
building for production...
dist/my-lib.js      0.08 kB / gzip: 0.07 kB
dist/my-lib.umd.cjs 0.30 kB / gzip: 0.16 kB
```

Recommended `package.json` for your lib:

```json
{
  "name": "my-lib",
  "type": "module",
  "files": ["dist"],
  "main": "./dist/my-lib.umd.cjs",
  "module": "./dist/my-lib.js",
  "exports": {
    ".": {
      "import": "./dist/my-lib.js",
      "require": "./dist/my-lib.umd.cjs"
    }
  }
}
```

Or, if exposing multiple entry points:

```json
{
  "name": "my-lib",
  "type": "module",
  "files": ["dist"],
  "main": "./dist/my-lib.cjs",
  "module": "./dist/my-lib.js",
  "exports": {
    ".": {
      "import": "./dist/my-lib.js",
      "require": "./dist/my-lib.cjs"
    },
    "./secondary": {
      "import": "./dist/secondary.js",
      "require": "./dist/secondary.cjs"
    }
  }
}
```

::: tip File Extensions
If the `package.json` does not contain `"type": "module"`, Vite will generate different file extensions for Node.js
compatibility. `.js` will become `.mjs` and `.cjs` will become `.js`.
:::

::: tip Environment Variables
In library mode, all [`import.meta.env.*`](./env-and-mode.md) usage are statically replaced when building for production.
However, `process.env.*` usage are not, so that consumers of your library can dynamically change it. If this is undesirable,
~~~

you can use `define: { 'process.env.NODE_ENV': '"production"' }` for example to statically replace them, or use [`esm-env`](https://github.com/benmccann/esm-env) for better compatibility with bundlers and runtimes.
:::

::: warning Advanced Usage
Library mode includes a simple and opinionated configuration for browser-oriented and JS framework libraries. If you are building non-browser libraries, or require advanced build flows, you can use [Rollup](https://rollupjs.org) or [esbuild](https://esbuild.github.io) directly.
:::

## Advanced Base Options

::: warning
This feature is experimental. [Give Feedback](https://github.com/vitejs/vite/discussions/13834).
:::

For advanced use cases, the deployed assets and public files may be in different paths, for example to use different cache strategies.
A user may choose to deploy in three different paths:

- The generated entry HTML files (which may be processed during SSR)
- The generated hashed assets (JS, CSS, and other file types like images)
- The copied [public files](assets.md#the-public-directory)

A single static [base](#public-base-path) isn't enough in these scenarios. Vite provides experimental support for advanced base options during build, using `experimental.renderBuiltUrl`.

```ts twoslash
import type { UserConfig } from 'vite'
// prettier-ignore
const config: UserConfig = {
// ---cut-before---
experimental: {
  renderBuiltUrl(filename, { hostType }) {
    if (hostType === 'js') {
      return { runtime: `window.__toCdnUrl(${JSON.stringify(filename)})` }
    } else {
      return { relative: true }
    }
  },
},
// ---cut-after---
}
```

If the hashed assets and public files aren't deployed together, options for each group can be defined independently using asset `type` included in the second `context` param given to the function.

```ts twoslash
import type { UserConfig } from 'vite'
import path from 'node:path'
// prettier-ignore
const config: UserConfig = {
// ---cut-before---
experimental: {
  renderBuiltUrl(filename, { hostId, hostType, type }) {
    if (type === 'public') {
      return 'https://www.domain.com/' + filename
    } else if (path.extname(hostId) === '.js') {
      return { runtime: `window.__assetsPath(${JSON.stringify(filename)})` }
    } else {
      return 'https://cdn.domain.com/assets/' + filename
    }
  },
},
// ---cut-after---
}
```

Note that the `filename` passed is a decoded URL, and if the function returns a URL string, it should also be decoded. Vite will handle the encoding automatically when rendering the URLs. If an object with `runtime` is returned, encoding should be handled yourself where needed as the runtime code will be rendered as is.

# Command Line Interface

## Dev server

### `vite`

Start Vite dev server in the current directory. `vite dev` and `vite serve` are aliases for `vite`.

#### Usage

```bash
vite [root]
```

#### Options

Options	
`--host [host]`	Specify hostname (`string`)
`--port <port>`	Specify port (`number`)
`--open [path]`	Open browser on startup (`boolean | string`)
`--cors`	Enable CORS (`boolean`)
`--strictPort`	Exit if specified port is already in use (`boolean`)
`--force`	Force the optimizer to ignore the cache and re-bundle (`boolean`)
`-c, --config <file>`	Use specified config file (`string`)
`--base <path>`	Public base path (default: `/`) (`string`)
`-l, --logLevel <level>`	info | warn | error | silent (`string`)
`--clearScreen`	Allow/disable clear screen when logging (`boolean`)
`--profile`	Start built-in Node.js inspector (check [Performance bottlenecks](/guide/troubleshooting#performance-bottlenecks))
`-d, --debug [feat]`	Show debug logs (`string | boolean`)
`-f, --filter <filter>`	Filter debug logs (`string`)
`-m, --mode <mode>`	Set env mode (`string`)
`-h, --help`	Display available CLI options
`-v, --version`	Display version number

## Build

### `vite build`

Build for production.

#### Usage

```bash
vite build [root]
```

#### Options

Options	
`--target <target>`	Transpile target (default: `"modules"`) (`string`)
`--outDir <dir>`	Output directory (default: `dist`) (`string`)
`--assetsDir <dir>`	Directory under outDir to place assets in (default: `"assets"`) (`string`)
`--assetsInlineLimit <number>`	Static asset base64 inline threshold in bytes (default: `4096`) (`number`)
`--ssr [entry]`	Build specified entry for server-side rendering (`string`)
`--sourcemap [output]`	Output source maps for build (default: `false`) (`boolean | "inline" | "hidden"`)

`--minify [minifier]`	Enable/disable minification, or specify minifier to use (default: `"esbuild"`) (`boolean | "terser" | "esbuild"`)
`--manifest [name]`	Emit build manifest json (`boolean | string`)
`--ssrManifest [name]`	Emit ssr manifest json (`boolean | string`)
`--emptyOutDir`	Force empty outDir when it's outside of root (`boolean`)
`-w, --watch`	Rebuilds when modules have changed on disk (`boolean`)
`-c, --config <file>`	Use specified config file (`string`)
`--base <path>`	Public base path (default: `/`) (`string`)
`-l, --logLevel <level>`	Info | warn | error | silent (`string`)
`--clearScreen`	Allow/disable clear screen when logging (`boolean`)
`--profile`	Start built-in Node.js inspector (check [Performance bottlenecks](/guide/troubleshooting#performance-bottlenecks))
`-d, --debug [feat]`	Show debug logs (`string | boolean`)
`-f, --filter <filter>`	Filter debug logs (`string`)
`-m, --mode <mode>`	Set env mode (`string`)
`-h, --help`	Display available CLI options

## Others

### `vite optimize`

Pre-bundle dependencies.

#### Usage

```bash
vite optimize [root]
```

#### Options

Options	
`--force`	Force the optimizer to ignore the cache and re-bundle (`boolean`)
`-c, --config <file>`	Use specified config file (`string`)
`--base <path>`	Public base path (default: `/`) (`string`)
`-l, --logLevel <level>`	Info | warn | error | silent (`string`)
`--clearScreen`	Allow/disable clear screen when logging (`boolean`)
`-d, --debug [feat]`	Show debug logs (`string | boolean`)
`-f, --filter <filter>`	Filter debug logs (`string`)
`-m, --mode <mode>`	Set env mode (`string`)
`-h, --help`	Display available CLI options

### `vite preview`

Locally preview the production build. Do not use this as a production server as it's not designed for it.

#### Usage

```bash
vite preview [root]
```

#### Options

Options	
`--host [host]`	Specify hostname (`string`)
`--port <port>`	Specify port (`number`)
`--strictPort`	Exit if specified port is already in use (`boolean`)
`--open [path]`	Open browser on startup (`boolean | string`)
`--outDir <dir>`	Output directory (default: `dist`)(`string`)
`-c, --config <file>`	Use specified config file (`string`)
`--base <path>`	Public base path (default: `/`) (`string`)
`-l, --logLevel <level>`	Info | warn | error | silent (`string`)
`--clearScreen`	Allow/disable clear screen when logging (`boolean`)
`-d, --debug [feat]`	Show debug logs (`string | boolean`)
`-f, --filter <filter>`	Filter debug logs (`string`)
`-m, --mode <mode>`	Set env mode (`string`)
`-h, --help`	Display available CLI options

# Comparisons

## WMR

[WMR](https://github.com/preactjs/wmr) by the Preact team looked to provide a similar feature set. Vite's universal Rollup plugin API for dev and build was inspired by it.

WMR is no longer maintained. The Preact team now recommends Vite with [@preactjs/preset-vite](https://github.com/preactjs/preset-vite).

## @web/dev-server

[@web/dev-server](https://modern-web.dev/docs/dev-server/overview/) (previously `es-dev-server`) is a great project and Vite 1.0's Koa-based server setup was inspired by it.

`@web/dev-server` is a bit lower-level in terms of scope. It does not provide official framework integrations, and requires manually setting up a Rollup configuration for the production build.

Overall, Vite is a more opinionated / higher-level tool that aims to provide a more out-of-the-box workflow. That said, the `@web` umbrella project contains many other excellent tools that may benefit Vite users as well.

## Snowpack

[Snowpack](https://www.snowpack.dev/) was also a no-bundle native ESM dev server, very similar in scope to Vite. The project is no longer being maintained. The Snowpack team is now working on [Astro](https://astro.build/), a static site builder powered by Vite. The Astro team is now an active player in the ecosystem, and they are helping to improve Vite.

Aside from different implementation details, the two projects shared a lot in terms of technical advantages over traditional tooling. Vite's dependency pre-bundling is also inspired by Snowpack v1 (now [`esinstall`](https://github.com/snowpackjs/snowpack/tree/main/esinstall)). Some of the main differences between the two projects are listed in [the v2 Comparisons Guide](https://v2.vitejs.dev/guide/comparisons).

# Releases

Vite releases follow [Semantic Versioning](https://semver.org/). You can see the latest stable version of Vite in the [Vite npm package page](https://www.npmjs.com/package/vite).

A full changelog of past releases is [available on GitHub](https://github.com/vitejs/vite/blob/main/packages/vite/CHANGELOG.md).

## Release Cycle

Vite does not have a fixed release cycle.

- **Patch** releases are released as needed (usually every week).
- **Minor** releases always contain new features and are released as needed. Minor releases always have a beta pre-release phase (usually every two months).
- **Major** releases generally align with [Node.js EOL schedule](https://endoflife.date/nodejs), and will be announced ahead of time. These releases will go through long-term discussions with the ecosystem, and have alpha and beta pre-release phases (usually every year).

The Vite versions ranges that are supported by the Vite team is automatically determined by:

- **Current Minor** gets regular fixes.
- **Previous Major** (only for its latest minor) and **Previous Minor** receives important fixes and security patches.
- **Second-to-last Major** (only for its latest minor) and **Second-to-last Minor** receives security patches.
- All versions before these are no longer supported.

As an example, if the Vite latest is at 5.3.10:

- Regular patches are released for `vite@5.3`.
- Important fixes and security patches are backported to `vite@4` and `vite@5.2`.
- Security patches are also backported to `vite@3`, and `vite@5.1`.
- `vite@2` and `vite@5.0` are no longer supported. Users should upgrade to receive updates.

We recommend updating Vite regularly. Check out the [Migration Guides](https://vitejs.dev/guide/migration.html) when you update to each Major. The Vite team works closely with the main projects in the ecosystem to ensure the quality of new versions. We test new Vite versions before releasing them through the [vite-ecosystem-ci project](https://github.com/vitejs/vite-ecosystem-ci). Most projects using Vite should be able to quickly offer support or migrate to new versions as soon as they are released.

## Semantic Versioning Edge Cases

### TypeScript Definitions

We may ship incompatible changes to TypeScript definitions between minor versions. This is because:

- Sometimes TypeScript itself ships incompatible changes between minor versions, and we may have to adjust types to support newer versions of TypeScript.
- Occasionally we may need to adopt features that are only available in a newer version of TypeScript, raising the minimum required version of TypeScript.
- If you are using TypeScript, you can use a semver range that locks the current minor and manually upgrade when a new minor version of Vite is released.

### esbuild

[esbuild](https://esbuild.github.io/) is pre-1.0.0 and sometimes it has a breaking change we may need to include to have access to newer features and performance improvements. We may bump the esbuild's version in a Vite Minor.

### Node.js non-LTS versions

Non-LTS Node.js versions (odd-numbered) are not tested as part of Vite's CI, but they should still work before their [EOL](https://endoflife.date/nodejs).

## Pre Releases

Minor releases typically go through a non-fixed number of beta releases. Major releases will go through an alpha phase and a beta phase.

Pre-releases allow early adopters and maintainers from the Ecosystem to do integration and stability testing, and provide feedback. Do not use pre-releases in production. All pre-releases are considered unstable and may ship breaking changes in between. Always pin to exact versions when using pre-releases.

## Deprecations

We periodically deprecate features that have been superseded by better alternatives in Minor releases. Deprecated features will continue to work with a type or logged warning. They will be removed in the next major release after entering deprecated status. The [Migration Guide](https://vitejs.dev/guide/migration.html) for each major will list these removals and document an upgrade path for them.

## Experimental Features

Some features are marked as experimental when released in a stable version of Vite. Experimental features allow us to gather real-world experience to influence their final design. The goal is to let users provide feedback by testing them in production. Experimental features themselves are considered unstable, and should only be used in a controlled manner. These features may change between Minors, so users must pin their Vite version when they rely on them. We will create [a GitHub discussion](https://github.com/vitejs/vite/discussions/categories/feedback?discussions_q=is%3Aopen+label%3Aexperimental+category%3AFeedback) for each experimental feature.

```
---
layout: page
title: Meet the Team
description: The development of Vite is guided by an international team.
---

<script setup>
import {
  VPTeamPage,
  VPTeamPageTitle,
  VPTeamPageSection,
  VPTeamMembers
} from 'vitepress/theme'
import { core, emeriti } from './_data/team'
</script>

<VPTeamPage>
  <VPTeamPageTitle>
    <template #title>Meet the Team</template>
    <template #lead>
      The development of Vite is guided by an international team, some of whom
      have chosen to be featured below.
    </template>
  </VPTeamPageTitle>
  <VPTeamMembers :members="core" />
  <VPTeamPageSection>
    <template #title>Team Emeriti</template>
    <template #lead>
      Here we honor some no-longer-active team members who have made valuable
      contributions in the past.
    </template>
    <template #members>
      <VPTeamMembers size="small" :members="emeriti" />
    </template>
  </VPTeamPageSection>
</VPTeamPage>
```

# Dependency Pre-Bundling

When you run `vite` for the first time, Vite prebundles your project dependencies before loading your site locally. It is done automatically and transparently by default.

## The Why

This is Vite performing what we call "dependency pre-bundling". This process serves two purposes:

1. **CommonJS and UMD compatibility:** During development, Vite's dev serves all code as native ESM. Therefore, Vite must convert dependencies that are shipped as CommonJS or UMD into ESM first.

   When converting CommonJS dependencies, Vite performs smart import analysis so that named imports to CommonJS modules will work as expected even if the exports are dynamically assigned (e.g. React):

   ```js
   // works as expected
   import React, { useState } from 'react'
   ```

2. **Performance:** Vite converts ESM dependencies with many internal modules into a single module to improve subsequent page load performance.

   Some packages ship their ES modules builds as many separate files importing one another. For example, [`lodash-es` has over 600 internal modules](https://unpkg.com/browse/lodash-es/)! When we do `import { debounce } from 'lodash-es'`, the browser fires off 600+ HTTP requests at the same time! Even though the server has no problem handling them, the large amount of requests create a network congestion on the browser side, causing the page to load noticeably slower.

   By pre-bundling `lodash-es` into a single module, we now only need one HTTP request instead!

::: tip NOTE
Dependency pre-bundling only applies in development mode, and uses `esbuild` to convert dependencies to ESM. In production builds, `@rollup/plugin-commonjs` is used instead.
:::

## Automatic Dependency Discovery

If an existing cache is not found, Vite will crawl your source code and automatically discover dependency imports (i.e. "bare imports" that expect to be resolved from `node_modules`) and use these found imports as entry points for the pre-bundle. The pre-bundling is performed with `esbuild` so it's typically very fast.

After the server has already started, if a new dependency import is encountered that isn't already in the cache, Vite will re-run the dep bundling process and reload the page if needed.

## Monorepos and Linked Dependencies

In a monorepo setup, a dependency may be a linked package from the same repo. Vite automatically detects dependencies that are not resolved from `node_modules` and treats the linked dep as source code. It will not attempt to bundle the linked dep, and will analyze the linked dep's dependency list instead.

However, this requires the linked dep to be exported as ESM. If not, you can add the dependency to [`optimizeDeps.include`](/config/dep-optimization-options.md#optimizedeps-include) and [`build.commonjsOptions.include`](/config/build-options.md#build-commonjsoptions) in your config.

```js twoslash
import { defineConfig } from 'vite'
// ---cut---
export default defineConfig({
  optimizeDeps: {
    include: ['linked-dep'],
  },
  build: {
    commonjsOptions: {
      include: [/linked-dep/, /node_modules/],
    },
  },
})
```

When making changes to the linked dep, restart the dev server with the `--force` command line option for the changes to take effect.

## Customizing the Behavior

The default dependency discovery heuristics may not always be desirable. In cases where you want to explicitly include/exclude dependencies from the list, use the [`optimizeDeps` config options](/config/dep-optimization-options.md).

A typical use case for `optimizeDeps.include` or `optimizeDeps.exclude` is when you have an import that is not directly discoverable in the source code. For example, maybe the import is created as a result of a plugin transform. This means Vite won't be able to discover the import on the initial scan - it can only discover it after the file is requested by the browser and transformed. This will cause the server to immediately re-bundle after server start.

Both `include` and `exclude` can be used to deal with this. If the dependency is large (with many internal modules) or is

CommonJS, then you should include it; If the dependency is small and is already valid ESM, you can exclude it and let the browser load it directly.

You can further customize esbuild too with the [`optimizeDeps.esbuildOptions` option](/config/dep-optimization-options.md#optimizedeps-esbuildoptions). For example, adding an esbuild plugin to handle special files in dependencies or changing the [build `target`](https://esbuild.github.io/api/#target).

## Caching

### File System Cache

Vite caches the pre-bundled dependencies in `node_modules/.vite`. It determines whether it needs to re-run the pre-bundling step based on a few sources:

- Package manager lockfile content, e.g. `package-lock.json`, `yarn.lock`, `pnpm-lock.yaml` or `bun.lockb`.
- Patches folder modification time.
- Relevant fields in your `vite.config.js`, if present.
- `NODE_ENV` value.

The pre-bundling step will only need to be re-run when one of the above has changed.

If for some reason you want to force Vite to re-bundle deps, you can either start the dev server with the `--force` command line option, or manually delete the `node_modules/.vite` cache directory.

### Browser Cache

Resolved dependency requests are strongly cached with HTTP headers `max-age=31536000,immutable` to improve page reload performance during dev. Once cached, these requests will never hit the dev server again. They are auto invalidated by the appended version query if a different version is installed (as reflected in your package manager lockfile). If you want to debug your dependencies by making local edits, you can:

1. Temporarily disable cache via the Network tab of your browser devtools;
2. Restart Vite dev server with the `--force` flag to re-bundle the deps;
3. Reload the page.

# Env Variables and Modes

## Env Variables

Vite exposes env variables on the special **`import.meta.env`** object, which are statically replaced at build time. Some built-in variables are available in all cases:

- **`import.meta.env.MODE`**: {string} the [mode](#modes) the app is running in.

- **`import.meta.env.BASE_URL`**: {string} the base url the app is being served from. This is determined by the [`base` config option](/config/shared-options.md#base).

- **`import.meta.env.PROD`**: {boolean} whether the app is running in production (running the dev server with `NODE_ENV='production'` or running an app built with `NODE_ENV='production'`).

- **`import.meta.env.DEV`**: {boolean} whether the app is running in development (always the opposite of `import.meta.env.PROD`)

- **`import.meta.env.SSR`**: {boolean} whether the app is running in the [server](./ssr.md#conditional-logic).

## `.env` Files

Vite uses [dotenv](https://github.com/motdotla/dotenv) to load additional environment variables from the following files in your [environment directory](/config/shared-options.md#envdir):

```
.env                # loaded in all cases
.env.local          # loaded in all cases, ignored by git
.env.[mode]         # only loaded in specified mode
.env.[mode].local   # only loaded in specified mode, ignored by git
```

:::tip Env Loading Priorities

An env file for a specific mode (e.g. `.env.production`) will take higher priority than a generic one (e.g. `.env`).

In addition, environment variables that already exist when Vite is executed have the highest priority and will not be overwritten by `.env` files. For example, when running `VITE_SOME_KEY=123 vite build`.

`.env` files are loaded at the start of Vite. Restart the server after making changes.
:::

Loaded env variables are also exposed to your client source code via `import.meta.env` as strings.

To prevent accidentally leaking env variables to the client, only variables prefixed with `VITE_` are exposed to your Vite-processed code. e.g. for the following env variables:

```
VITE_SOME_KEY=123
DB_PASSWORD=foobar
```

Only `VITE_SOME_KEY` will be exposed as `import.meta.env.VITE_SOME_KEY` to your client source code, but `DB_PASSWORD` will not.

```js
console.log(import.meta.env.VITE_SOME_KEY) // "123"
console.log(import.meta.env.DB_PASSWORD) // undefined
```

:::tip Env parsing

As shown above, `VITE_SOME_KEY` is a number but returns a string when parsed. The same would also happen for boolean env variables. Make sure to convert to the desired type when using it in your code.
:::

Also, Vite uses [dotenv-expand](https://github.com/motdotla/dotenv-expand) to expand variables out of the box. To learn more about the syntax, check out [their docs](https://github.com/motdotla/dotenv-expand#what-rules-does-the-expansion-engine-follow).

Note that if you want to use `$` inside your environment value, you have to escape it with `\`.

```
KEY=123
NEW_KEY1=test$foo   # test
NEW_KEY2=test\$foo  # test$foo
NEW_KEY3=test$KEY   # test123
```

If you want to customize the env variables prefix, see the [envPrefix](/config/shared-options.html#envprefix) option.

:::warning SECURITY NOTES

- `.env.*.local` files are local-only and can contain sensitive variables. You should add `*.local` to your `.gitignore` to avoid them being checked into git.

- Since any variables exposed to your Vite source code will end up in your client bundle, `VITE_*` variables should _not_ contain any sensitive information.
  :::

### IntelliSense for TypeScript

By default, Vite provides type definitions for `import.meta.env` in [`vite/client.d.ts`](https://github.com/vitejs/vite/blob/main/packages/vite/client.d.ts). While you can define more custom env variables in `.env.[mode]` files, you may want to get TypeScript IntelliSense for user-defined env variables that are prefixed with `VITE_`.

To achieve this, you can create an `vite-env.d.ts` in `src` directory, then augment `ImportMetaEnv` like this:

```typescript
/// <reference types="vite/client" />

interface ImportMetaEnv {
  readonly VITE_APP_TITLE: string
  // more env variables...
}

interface ImportMeta {
  readonly env: ImportMetaEnv
}
```

If your code relies on types from browser environments such as [DOM](https://github.com/microsoft/TypeScript/blob/main/src/lib/dom.generated.d.ts) and [WebWorker](https://github.com/microsoft/TypeScript/blob/main/src/lib/webworker.generated.d.ts), you can update the [lib](https://www.typescriptlang.org/tsconfig#lib) field in `tsconfig.json`.

```json
{
  "lib": ["WebWorker"]
}
```

:::warning Imports will break type augmentation

If the `ImportMetaEnv` augmentation does not work, make sure you do not have any `import` statements in `vite-env.d.ts`. See the [TypeScript documentation](https://www.typescriptlang.org/docs/handbook/2/modules.html#how-javascript-modules-are-defined) for more information.
:::

## HTML Env Replacement

Vite also supports replacing env variables in HTML files. Any properties in `import.meta.env` can be used in HTML files with a special `%ENV_NAME%` syntax:

```html
<h1>Vite is running in %MODE%</h1>
<p>Using data from %VITE_API_URL%</p>
```

If the env doesn't exist in `import.meta.env`, e.g. `%NON_EXISTENT%`, it will be ignored and not replaced, unlike `import.meta.env.NON_EXISTENT` in JS where it's replaced as `undefined`.

Given that Vite is used by many frameworks, it is intentionally unopinionated about complex replacements like conditionals. Vite can be extended using [an existing userland plugin](https://github.com/vitejs/awesome-vite#transformers) or a custom plugin that implements the [`transformIndexHtml` hook](./api-plugin#transformindexhtml).

## Modes

By default, the dev server (`dev` command) runs in `development` mode and the `build` command runs in `production` mode.

This means when running `vite build`, it will load the env variables from `.env.production` if there is one:

```
# .env.production
VITE_APP_TITLE=My App
```

In your app, you can render the title using `import.meta.env.VITE_APP_TITLE`.

In some cases, you may want to run `vite build` with a different mode to render a different title. You can overwrite the default mode used for a command by passing the `--mode` option flag. For example, if you want to build your app for a staging mode:

```bash
vite build --mode staging
```

~~~
```

And create a `.env.staging` file:

```
# .env.staging
VITE_APP_TITLE=My App (staging)
```

As `vite build` runs a production build by default, you can also change this and run a development build by using a different
mode and `.env` file configuration:

```
# .env.testing
NODE_ENV=development
```

## NODE_ENV and Modes

It's important to note that `NODE_ENV` (`process.env.NODE_ENV`) and modes are two different concepts. Here's how different
commands affect the `NODE_ENV` and mode:

| Command                                              | NODE_ENV        | Mode            |
| ---------------------------------------------------- | --------------- | --------------- |
| `vite build`                                         | `"production"`  | `"production"`  |
| `vite build --mode development`                      | `"production"`  | `"development"` |
| `NODE_ENV=development vite build`                    | `"development"` | `"production"`  |
| `NODE_ENV=development vite build --mode development` | `"development"` | `"development"` |

The different values of `NODE_ENV` and mode also reflect on its corresponding `import.meta.env` properties:

| Command                | `import.meta.env.PROD` | `import.meta.env.DEV` |
| ---------------------- | ---------------------- | --------------------- |
| `NODE_ENV=production`  | `true`                 | `false`               |
| `NODE_ENV=development` | `false`                | `true`                |
| `NODE_ENV=other`       | `false`                | `true`                |

| Command              | `import.meta.env.MODE` |
| -------------------- | ---------------------- |
| `--mode production`  | `"production"`         |
| `--mode development` | `"development"`        |
| `--mode staging`     | `"staging"`            |

:::tip `NODE_ENV` in `.env` files

`NODE_ENV=...` can be set in the command, and also in your `.env` file. If `NODE_ENV` is specified in a `.env.[mode]` file,
the mode can be used to control its value. However, both `NODE_ENV` and modes remain as two different concepts.

The main benefit with `NODE_ENV=...` in the command is that it allows Vite to detect the value early. It also allows you to
read `process.env.NODE_ENV` in your Vite config as Vite can only load the env files once the config is evaluated.
:::
~~~

# Features

At the very basic level, developing using Vite is not that different from using a static file server. However, Vite provides many enhancements over native ESM imports to support various features that are typically seen in bundler-based setups.

## NPM Dependency Resolving and Pre-Bundling

Native ES imports do not support bare module imports like the following:

```js
import { someMethod } from 'my-dep'
```

The above will throw an error in the browser. Vite will detect such bare module imports in all served source files and perform the following:

1. [Pre-bundle](./dep-pre-bundling) them to improve page loading speed and convert CommonJS / UMD modules to ESM. The pre-bundling step is performed with [esbuild](http://esbuild.github.io/) and makes Vite's cold start time significantly faster than any JavaScript-based bundler.

2. Rewrite the imports to valid URLs like `/node_modules/.vite/deps/my-dep.js?v=f3sf2ebd` so that the browser can import them properly.

Dependencies are Strongly Cached

Vite caches dependency requests via HTTP headers, so if you wish to locally edit/debug a dependency, follow the steps [here](./dep-pre-bundling#browser-cache).

## Hot Module Replacement

Vite provides an [HMR API](./api-hmr) over native ESM. Frameworks with HMR capabilities can leverage the API to provide instant, precise updates without reloading the page or blowing away application state. Vite provides first-party HMR integrations for [Vue Single File Components](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue) and [React Fast Refresh](https://github.com/vitejs/vite-plugin-react/tree/main/packages/plugin-react). There are also official integrations for Preact via [@prefresh/vite](https://github.com/JoviDeCroock/prefresh/tree/main/packages/vite).

Note you don't need to manually set these up - when you [create an app via `create-vite`](./), the selected templates would have these pre-configured for you already.

## TypeScript

Vite supports importing `.ts` files out of the box.

### Transpile Only

Note that Vite only performs transpilation on `.ts` files and does **NOT** perform type checking. It assumes type checking is taken care of by your IDE and build process.

The reason Vite does not perform type checking as part of the transform process is because the two jobs work fundamentally differently. Transpilation can work on a per-file basis and aligns perfectly with Vite's on-demand compile model. In comparison, type checking requires knowledge of the entire module graph. Shoe-horning type checking into Vite's transform pipeline will inevitably compromise Vite's speed benefits.

Vite's job is to get your source modules into a form that can run in the browser as fast as possible. To that end, we recommend separating static analysis checks from Vite's transform pipeline. This principle applies to other static analysis checks such as ESLint.

- For production builds, you can run `tsc --noEmit` in addition to Vite's build command.

- During development, if you need more than IDE hints, we recommend running `tsc --noEmit --watch` in a separate process, or use [vite-plugin-checker](https://github.com/fi3ework/vite-plugin-checker) if you prefer having type errors directly reported in the browser.

Vite uses [esbuild](https://github.com/evanw/esbuild) to transpile TypeScript into JavaScript which is about 20~30x faster than vanilla `tsc`, and HMR updates can reflect in the browser in under 50ms.

Use the [Type-Only Imports and Export](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-8.html#type-only-imports-and-export) syntax to avoid potential problems like type-only imports being incorrectly bundled, for example:

```ts
import type { T } from 'only/types'
export type { T }
```

### TypeScript Compiler Options

Some configuration fields under `compilerOptions` in `tsconfig.json` require special attention.

#### `isolatedModules`

- [TypeScript documentation](https://www.typescriptlang.org/tsconfig#isolatedModules)

Should be set to `true`.

It is because `esbuild` only performs transpilation without type information, it doesn't support certain features like const enum and implicit type-only imports.

You must set `"isolatedModules": true` in your `tsconfig.json` under `compilerOptions`, so that TS will warn you against the features that do not work with isolated transpilation.

If a dependency doesn't work well with `"isolatedModules": true`. You can use `"skipLibCheck": true` to temporarily suppress the errors until it is fixed upstream.

#### `useDefineForClassFields`

- [TypeScript documentation](https://www.typescriptlang.org/tsconfig#useDefineForClassFields)

Starting from Vite 2.5.0, the default value will be `true` if the TypeScript target is `ESNext` or `ES2022` or newer. It is consistent with the [behavior of `tsc` 4.3.2 and later](https://github.com/microsoft/TypeScript/pull/42663). It is also the standard ECMAScript runtime behavior.

Other TypeScript targets will default to `false`.

But it may be counter-intuitive for those coming from other programming languages or older versions of TypeScript.
You can read more about the transition in the [TypeScript 3.7 release notes](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#the-usedefineforclassfields-flag-and-the-declare-property-modifier).

If you are using a library that heavily relies on class fields, please be careful about the library's intended usage of it.

Most libraries expect `"useDefineForClassFields": true`, such as [MobX](https://mobx.js.org/installation.html#use-spec-compliant-transpilation-for-class-properties).

But a few libraries haven't transitioned to this new default yet, including [`lit-element`](https://github.com/lit/lit-element/issues/1030). Please explicitly set `useDefineForClassFields` to `false` in these cases.

#### `target`

- [TypeScript documentation](https://www.typescriptlang.org/tsconfig#target)

Vite does not transpile TypeScript with the configured `target` value by default, following the same behaviour as `esbuild`.

The [`esbuild.target`](/config/shared-options.html#esbuild) option can be used instead, which defaults to `esnext` for minimal transpilation. In builds, the [`build.target`](/config/build-options.html#build-target) option takes higher priority and can also be set if needed.

::: warning `useDefineForClassFields`
If `target` is not `ESNext` or `ES2022` or newer, or if there's no `tsconfig.json` file, `useDefineForClassFields` will default to `false` which can be problematic with the default `esbuild.target` value of `esnext`. It may transpile to [static initialization blocks](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Static_initialization_blocks#browser_compatibility) which may not be supported in your browser.

As such, it is recommended to set `target` to `ESNext` or `ES2022` or newer, or set `useDefineForClassFields` to `true` explicitly when configuring `tsconfig.json`.
:::

#### Other Compiler Options Affecting the Build Result

- [`extends`](https://www.typescriptlang.org/tsconfig#extends)
- [`importsNotUsedAsValues`](https://www.typescriptlang.org/tsconfig#importsNotUsedAsValues)
- [`preserveValueImports`](https://www.typescriptlang.org/tsconfig#preserveValueImports)
- [`verbatimModuleSyntax`](https://www.typescriptlang.org/tsconfig#verbatimModuleSyntax)
- [`jsx`](https://www.typescriptlang.org/tsconfig#jsx)
- [`jsxFactory`](https://www.typescriptlang.org/tsconfig#jsxFactory)
- [`jsxFragmentFactory`](https://www.typescriptlang.org/tsconfig#jsxFragmentFactory)
- [`jsxImportSource`](https://www.typescriptlang.org/tsconfig#jsxImportSource)
- [`experimentalDecorators`](https://www.typescriptlang.org/tsconfig#experimentalDecorators)
- [`alwaysStrict`](https://www.typescriptlang.org/tsconfig#alwaysStrict)

::: tip `skipLibCheck`
Vite starter templates have `"skipLibCheck": "true"` by default to avoid typechecking dependencies, as they may choose to only support specific versions and configurations of TypeScript. You can learn more at [vuejs/vue-cli#5688](https://github.com/vuejs/vue-cli/pull/5688).
:::

### Client Types

Vite's default types are for its Node.js API. To shim the environment of client side code in a Vite application, add a `d.ts` declaration file:

```typescript
/// <reference types="vite/client" />
```

Alternatively, you can add `vite/client` to `compilerOptions.types` inside `tsconfig.json`:

```json
{
  "compilerOptions": {
    "types": ["vite/client"]
  }
}
```

This will provide the following type shims:

- Asset imports (e.g. importing an `.svg` file)
- Types for the Vite-injected [env variables](./env-and-mode#env-variables) on `import.meta.env`
- Types for the [HMR API](./api-hmr) on `import.meta.hot`

::: tip
To override the default typing, add a type definition file that contains your typings. Then, add the type reference before `vite/client`.

For example, to make the default import of `*.svg` a React component:

- `vite-env-override.d.ts` (the file that contains your typings):
  ```ts
  declare module '*.svg' {
    const content: React.FC<React.SVGProps<SVGElement>>
    export default content
  }
  ```
- The file containing the reference to `vite/client`:
  ```ts
  /// <reference types="./vite-env-override.d.ts" />
  /// <reference types="vite/client" />
  ```

:::

## Vue

Vite provides first-class Vue support:

- Vue 3 SFC support via [@vitejs/plugin-vue](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue)
- Vue 3 JSX support via [@vitejs/plugin-vue-jsx](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue-jsx)
- Vue 2.7 SFC support via [@vitejs/plugin-vue2](https://github.com/vitejs/vite-plugin-vue2)
- Vue 2.7 JSX support via [@vitejs/plugin-vue2-jsx](https://github.com/vitejs/vite-plugin-vue2-jsx)

## JSX

`.jsx` and `.tsx` files are also supported out of the box. JSX transpilation is also handled via [esbuild](https://esbuild.github.io).

Vue users should use the official [@vitejs/plugin-vue-jsx](https://github.com/vitejs/vite-plugin-vue/tree/main/packages/plugin-vue-jsx) plugin, which provides Vue 3 specific features including HMR, global component resolving, directives and slots.

If using JSX without React or Vue, custom `jsxFactory` and `jsxFragment` can be configured using the [`esbuild` option](/config/shared-options.md#esbuild). For example for Preact:

```js twoslash
// vite.config.js
import { defineConfig } from 'vite'

export default defineConfig({
  esbuild: {
    jsxFactory: 'h',
    jsxFragment: 'Fragment',
  },
})
```

More details in [esbuild docs](https://esbuild.github.io/content-types/#jsx).

You can inject the JSX helpers using `jsxInject` (which is a Vite-only option) to avoid manual imports:

```js twoslash
// vite.config.js
import { defineConfig } from 'vite'

export default defineConfig({
  esbuild: {
    jsxInject: `import React from 'react'`,
  },
})
```

## CSS

Importing `.css` files will inject its content to the page via a `<style>` tag with HMR support.

### `@import` Inlining and Rebasing

Vite is pre-configured to support CSS `@import` inlining via `postcss-import`. Vite aliases are also respected for CSS `@import`. In addition, all CSS `url()` references, even if the imported files are in different directories, are always automatically rebased to ensure correctness.

`@import` aliases and URL rebasing are also supported for Sass and Less files (see [CSS Pre-processors](#css-pre-processors)).

### PostCSS

If the project contains valid PostCSS config (any format supported by [postcss-load-config](https://github.com/postcss/postcss-load-config), e.g. `postcss.config.js`), it will be automatically applied to all imported CSS.

Note that CSS minification will run after PostCSS and will use [`build.cssTarget`](/config/build-options.md#build-csstarget) option.

### CSS Modules

Any CSS file ending with `.module.css` is considered a [CSS modules file](https://github.com/css-modules/css-modules). Importing such a file will return the corresponding module object:

```css
/* example.module.css */
.red {
  color: red;
}
```

```js twoslash
import 'vite/client'
// ---cut---
import classes from './example.module.css'
document.getElementById('foo').className = classes.red
```

CSS modules behavior can be configured via the [`css.modules` option](/config/shared-options.md#css-modules).

If `css.modules.localsConvention` is set to enable camelCase locals (e.g. `localsConvention: 'camelCaseOnly'`), you can also use named imports:

```js twoslash
import 'vite/client'
// ---cut---
// .apply-color -> applyColor
import { applyColor } from './example.module.css'
document.getElementById('foo').className = applyColor
```

### CSS Pre-processors

Because Vite targets modern browsers only, it is recommended to use native CSS variables with PostCSS plugins that implement CSSWG drafts (e.g. [postcss-nesting](https://github.com/csstools/postcss-plugins/tree/main/plugins/postcss-nesting)) and author plain, future-standards-compliant CSS.

That said, Vite does provide built-in support for `.scss`, `.sass`, `.less`, `.styl` and `.stylus` files. There is no need to install Vite-specific plugins for them, but the corresponding pre-processor itself must be installed:

```bash
# .scss and .sass
npm add -D sass-embedded # or sass

# .less
npm add -D less

# .styl and .stylus
npm add -D stylus
```

If using Vue single file components, this also automatically enables `<style lang="sass">` et al.

Vite improves `@import` resolving for Sass and Less so that Vite aliases are also respected. In addition, relative `url()` references inside imported Sass/Less files that are in different directories from the root file are also automatically rebased to ensure correctness.

`@import` alias and url rebasing are not supported for Stylus due to its API constraints.

You can also use CSS modules combined with pre-processors by prepending `.module` to the file extension, for example `style.module.scss`.

### Disabling CSS injection into the page

The automatic injection of CSS contents can be turned off via the `?inline` query parameter. In this case, the processed CSS string is returned as the module's default export as usual, but the styles aren't injected to the page.

```js twoslash
import 'vite/client'
// ---cut---
import './foo.css' // will be injected into the page
import otherStyles from './bar.css?inline' // will not be injected
```

::: tip NOTE
Default and named imports from CSS files (e.g `import style from './foo.css'`) are removed since Vite 5. Use the `?inline` query instead.
:::

### Lightning CSS

Starting from Vite 4.4, there is experimental support for [Lightning CSS](https://lightningcss.dev/). You can opt into it by adding [`css.transformer: 'lightningcss'`](../config/shared-options.md#css-transformer) to your config file and install the optional [`lightningcss`](https://www.npmjs.com/package/lightningcss) dependency:

```bash
npm add -D lightningcss
```

If enabled, CSS files will be processed by Lightning CSS instead of PostCSS. To configure it, you can pass Lightning CSS options to the [`css.lightningcss`](../config/shared-options.md#css-lightningcss) config option.

To configure CSS Modules, you'll use [`css.lightningcss.cssModules`](https://lightningcss.dev/css-modules.html) instead of [`css.modules`](../config/shared-options.md#css-modules) (which configures the way PostCSS handles CSS modules).

By default, Vite uses esbuild to minify CSS. Lightning CSS can also be used as the CSS minifier with [`build.cssMinify: 'lightningcss'`](../config/build-options.md#build-cssminify).

::: tip NOTE
[CSS Pre-processors](#css-pre-processors) aren't supported when using Lightning CSS.
:::

## Static Assets

Importing a static asset will return the resolved public URL when it is served:

```js twoslash
import 'vite/client'
// ---cut---
import imgUrl from './img.png'
document.getElementById('hero-img').src = imgUrl
```

Special queries can modify how assets are loaded:

```js twoslash
import 'vite/client'
// ---cut---
// Explicitly load assets as URL
import assetAsURL from './asset.js?url'
```

```js twoslash
import 'vite/client'
// ---cut---
// Load assets as strings
import assetAsString from './shader.glsl?raw'
```

```js twoslash
import 'vite/client'
// ---cut---
// Load Web Workers
import Worker from './worker.js?worker'
```

```js twoslash
import 'vite/client'
// ---cut---
// Web Workers inlined as base64 strings at build time
import InlineWorker from './worker.js?worker&inline'
```

More details in [Static Asset Handling](./assets).

## JSON

JSON files can be directly imported - named imports are also supported:

```js twoslash
import 'vite/client'
// ---cut---
// import the entire object
import json from './example.json'
// import a root field as named exports - helps with tree-shaking!
import { field } from './example.json'
```

## Glob Import

Vite supports importing multiple modules from the file system via the special `import.meta.glob` function:

```js twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js')
```

The above will be transformed into the following:

```js
// code produced by vite
const modules = {
  './dir/foo.js': () => import('./dir/foo.js'),
  './dir/bar.js': () => import('./dir/bar.js'),
}
```

You can then iterate over the keys of the `modules` object to access the corresponding modules:

```js
for (const path in modules) {
  modules[path]().then((mod) => {
    console.log(path, mod)
  })
}
```

Matched files are by default lazy-loaded via dynamic import and will be split into separate chunks during build. If you'd rather import all the modules directly (e.g. relying on side-effects in these modules to be applied first), you can pass `{ eager: true }` as the second argument:

```js twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js', { eager: true })
```

The above will be transformed into the following:

```js
// code produced by vite
import * as __glob__0_0 from './dir/foo.js'
import * as __glob__0_1 from './dir/bar.js'
const modules = {
  './dir/foo.js': __glob__0_0,
  './dir/bar.js': __glob__0_1,
}
```

### Multiple Patterns

The first argument can be an array of globs, for example

```js twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob(['./dir/*.js', './another/*.js'])
```

### Negative Patterns

Negative glob patterns are also supported (prefixed with `!`). To ignore some files from the result, you can add exclude glob patterns to the first argument:

```js twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob(['./dir/*.js', '!**/bar.js'])
```

```js
// code produced by vite
const modules = {
  './dir/foo.js': () => import('./dir/foo.js'),
}
```

#### Named Imports

It's possible to only import parts of the modules with the `import` options.

```ts twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js', { import: 'setup' })
```

```ts
// code produced by vite
const modules = {
  './dir/foo.js': () => import('./dir/foo.js').then((m) => m.setup),
  './dir/bar.js': () => import('./dir/bar.js').then((m) => m.setup),
}
```

When combined with `eager` it's even possible to have tree-shaking enabled for those modules.

```ts twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js', {
  import: 'setup',
  eager: true,
})
```

```ts
// code produced by vite:
import { setup as __glob__0_0 } from './dir/foo.js'
import { setup as __glob__0_1 } from './dir/bar.js'
const modules = {
  './dir/foo.js': __glob__0_0,
  './dir/bar.js': __glob__0_1,
}
```

Set `import` to `default` to import the default export.

```ts twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js', {
  import: 'default',
  eager: true,
})
```

```ts
// code produced by vite:
import __glob__0_0 from './dir/foo.js'
import __glob__0_1 from './dir/bar.js'
const modules = {
  './dir/foo.js': __glob__0_0,
  './dir/bar.js': __glob__0_1,
}
```

#### Custom Queries

You can also use the `query` option to provide queries to imports, for example, to import assets [as a string](https://vitejs.dev/guide/assets.html#importing-asset-as-string) or [as a url](https://vitejs.dev/guide/assets.html#importing-asset-as-url):

```ts twoslash
import 'vite/client'
// ---cut---
const moduleStrings = import.meta.glob('./dir/*.svg', {
```

~~~markdown
  query: '?raw',
  import: 'default',
})
const moduleUrls = import.meta.glob('./dir/*.svg', {
  query: '?url',
  import: 'default',
})
```

```ts
// code produced by vite:
const moduleStrings = {
  './dir/foo.svg': () => import('./dir/foo.js?raw').then((m) => m['default']),
  './dir/bar.svg': () => import('./dir/bar.js?raw').then((m) => m['default']),
}
const moduleUrls = {
  './dir/foo.svg': () => import('./dir/foo.js?url').then((m) => m['default']),
  './dir/bar.svg': () => import('./dir/bar.js?url').then((m) => m['default']),
}
```

You can also provide custom queries for other plugins to consume:

```ts twoslash
import 'vite/client'
// ---cut---
const modules = import.meta.glob('./dir/*.js', {
  query: { foo: 'bar', bar: true },
})
```

### Glob Import Caveats

Note that:

- This is a Vite-only feature and is not a web or ES standard.
- The glob patterns are treated like import specifiers: they must be either relative (start with `./`) or absolute (start
with `/`, resolved relative to project root) or an alias path (see [`resolve.alias` option](/config/shared-
options.md#resolve-alias)).
- The glob matching is done via [`fast-glob`](https://github.com/mrmlnc/fast-glob) - check out its documentation for
[supported glob patterns](https://github.com/mrmlnc/fast-glob#pattern-syntax).
- You should also be aware that all the arguments in the `import.meta.glob` must be **passed as literals**. You can NOT use
variables or expressions in them.

## Dynamic Import

Similar to [glob import](#glob-import), Vite also supports dynamic import with variables.

```ts
const module = await import(`./dir/${file}.js`)
```

Note that variables only represent file names one level deep. If `file` is `'foo/bar'`, the import would fail. For more
advanced usage, you can use the [glob import](#glob-import) feature.

## WebAssembly

Pre-compiled `.wasm` files can be imported with `?init`.
The default export will be an initialization function that returns a Promise of the [`WebAssembly.Instance`]
(https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Instance):

```js twoslash
import 'vite/client'
// ---cut---
import init from './example.wasm?init'

init().then((instance) => {
  instance.exports.test()
})
```

The init function can also take an importObject which is passed along to [`WebAssembly.instantiate`]
(https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/instantiate) as its second argument:

```js twoslash
import 'vite/client'
import init from './example.wasm?init'
// ---cut---
init({
  imports: {
    someFunc: () => {
      /* ... */
    },
~~~

  },
}).then(() => {
  /* ... */
})

In the production build, `.wasm` files smaller than `assetInlineLimit` will be inlined as base64 strings. Otherwise, they will be treated as a [static asset](./assets) and fetched on-demand.

::: tip NOTE
[ES Module Integration Proposal for WebAssembly](https://github.com/WebAssembly/esm-integration) is not currently supported.
Use [`vite-plugin-wasm`](https://github.com/Menci/vite-plugin-wasm) or other community plugins to handle this.
:::

### Accessing the WebAssembly Module

If you need access to the `Module` object, e.g. to instantiate it multiple times, use an [explicit URL import](./assets#explicit-url-imports) to resolve the asset, and then perform the instantiation:

```js twoslash
import 'vite/client'
// ---cut---
import wasmUrl from 'foo.wasm?url'

const main = async () => {
  const responsePromise = fetch(wasmUrl)
  const { module, instance } =
    await WebAssembly.instantiateStreaming(responsePromise)
  /* ... */
}

main()
```

### Fetching the module in Node.js

In SSR, the `fetch()` happening as part of the `?init` import, may fail with `TypeError: Invalid URL`.
See the issue [Support wasm in SSR](https://github.com/vitejs/vite/issues/8882).

Here is an alternative, assuming the project base is the current directory:

```js twoslash
import 'vite/client'
// ---cut---
import wasmUrl from 'foo.wasm?url'
import { readFile } from 'node:fs/promises'

const main = async () => {
  const resolvedUrl = (await import('./test/boot.test.wasm?url')).default
  const buffer = await readFile('.' + resolvedUrl)
  const { instance } = await WebAssembly.instantiate(buffer, {
    /* ... */
  })
  /* ... */
}

main()
```

## Web Workers

### Import with Constructors

A web worker script can be imported using [`new Worker()`](https://developer.mozilla.org/en-US/docs/Web/API/Worker/Worker) and [`new SharedWorker()`](https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker/SharedWorker). Compared to the worker suffixes, this syntax leans closer to the standards and is the **recommended** way to create workers.

```ts
const worker = new Worker(new URL('./worker.js', import.meta.url))
```

The worker constructor also accepts options, which can be used to create "module" workers:

```ts
const worker = new Worker(new URL('./worker.js', import.meta.url), {
  type: 'module',
})
```

The worker detection will only work if the `new URL()` constructor is used directly inside the `new Worker()` declaration.
Additionally, all options parameters must be static values (i.e. string literals).

### Import with Query Suffixes

A web worker script can be directly imported by appending `?worker` or `?sharedworker` to the import request. The default export will be a custom worker constructor:

```js twoslash
import 'vite/client'
// ---cut---
import MyWorker from './worker?worker'

const worker = new MyWorker()
```

The worker script can also use ESM `import` statements instead of `importScripts()`. **Note**: During development this relies on [browser native support](https://caniuse.com/?search=module%20worker), but for the production build it is compiled away.

By default, the worker script will be emitted as a separate chunk in the production build. If you wish to inline the worker as base64 strings, add the `inline` query:

```js twoslash
import 'vite/client'
// ---cut---
import MyWorker from './worker?worker&inline'
```

If you wish to retrieve the worker as a URL, add the `url` query:

```js twoslash
import 'vite/client'
// ---cut---
import MyWorker from './worker?worker&url'
```

See [Worker Options](/config/worker-options.md) for details on configuring the bundling of all workers.

## Content Security Policy (CSP)

To deploy CSP, certain directives or configs must be set due to Vite's internals.

### [`'nonce-{RANDOM}'`](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/Sources#nonce-base64-value)

When [`html.cspNonce`](/config/shared-options#html-cspnonce) is set, Vite adds a nonce attribute with the specified value to any `<script>` and `<style>` tags, as well as `<link>` tags for stylesheets and module preloading. Additionally, when this option is set, Vite will inject a meta tag (`<meta property="csp-nonce" nonce="PLACEHOLDER" />`).

The nonce value of a meta tag with `property="csp-nonce"` will be used by Vite whenever necessary during both dev and after build.

:::warning
Ensure that you replace the placeholder with a unique value for each request. This is important to prevent bypassing a resource's policy, which can otherwise be easily done.
:::

### [`data:`](<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/Sources#scheme-source:~:text=schemes%20(not%20recommended).-,data%3A,-Allows%20data%3A>)

By default, during build, Vite inlines small assets as data URIs. Allowing `data:` for related directives (e.g. [`img-src`](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/img-src), [`font-src`](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/font-src)), or, disabling it by setting [`build.assetsInlineLimit: 0`](/config/build-options#build-assetsinlinelimit) is necessary.

:::warning
Do not allow `data:` for [`script-src`](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src). It will allow injection of arbitrary scripts.
:::

## Build Optimizations

> Features listed below are automatically applied as part of the build process and there is no need for explicit configuration unless you want to disable them.

### CSS Code Splitting

Vite automatically extracts the CSS used by modules in an async chunk and generates a separate file for it. The CSS file is automatically loaded via a `<link>` tag when the associated async chunk is loaded, and the async chunk is guaranteed to only be evaluated after the CSS is loaded to avoid [FOUC](https://en.wikipedia.org/wiki/Flash_of_unstyled_content#:~:text=A%20flash%20of%20unstyled%20content,before%20all%20information%20is%20retrieved.).

If you'd rather have all the CSS extracted into a single file, you can disable CSS code splitting by setting [`build.cssCodeSplit`](/config/build-options.md#build-csscodesplit) to `false`.

### Preload Directives Generation

Vite automatically generates `<link rel="modulepreload">` directives for entry chunks and their direct imports in the built HTML.

### Async Chunk Loading Optimization

In real world applications, Rollup often generates "common" chunks - code that is shared between two or more other chunks. Combined with dynamic imports, it is quite common to have the following scenario:

```
<script setup>
import graphSvg from '../images/graph.svg?raw'
</script>
<svg-image :svg="graphSvg" />
```

In the non-optimized scenarios, when async chunk `A` is imported, the browser will have to request and parse `A` before it can figure out that it also needs the common chunk `C`. This results in an extra network roundtrip:

```
Entry ---> A ---> C
```

Vite automatically rewrites code-split dynamic import calls with a preload step so that when `A` is requested, `C` is fetched **in parallel**:

```
Entry ---> (A + C)
```

It is possible for `C` to have further imports, which will result in even more roundtrips in the un-optimized scenario. Vite's optimization will trace all the direct imports to completely eliminate the roundtrips regardless of import depth.

~~~markdown
# Getting Started

<audio id="vite-audio">
  <source src="/vite.mp3" type="audio/mpeg">
</audio>

## Overview

Vite (French word for "quick", pronounced `/vit/`<button style="border:none;padding:3px;border-radius:4px;vertical-
align:bottom" id="play-vite-audio" onclick="document.getElementById('vite-audio').play();"><svg style="height:2em;width:2em">
<use href="/voice.svg#voice" /></svg></button>, like "veet") is a build tool that aims to provide a faster and leaner
development experience for modern web projects. It consists of two major parts:

- A dev server that provides [rich feature enhancements](./features) over [native ES modules]
(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules), for example extremely fast [Hot Module Replacement
(HMR)](./features#hot-module-replacement).

- A build command that bundles your code with [Rollup](https://rollupjs.org), pre-configured to output highly optimized
static assets for production.

Vite is opinionated and comes with sensible defaults out of the box. Read about what's possible in the [Features Guide]
(./features). Support for frameworks or integration with other tools is possible through [Plugins](./using-plugins). The
[Config Section](../config/) explains how to adapt Vite to your project if needed.

Vite is also highly extensible via its [Plugin API](./api-plugin) and [JavaScript API](./api-javascript) with full typing
support.

You can learn more about the rationale behind the project in the [Why Vite](./why) section.

## Browser Support

During development, Vite sets [`esnext` as the transform target](https://esbuild.github.io/api/#target), because we assume a
modern browser is used and it supports all of the latest JavaScript and CSS features. This prevents syntax lowering, letting
Vite serve modules as close as possible to the original source code.

For the production build, by default Vite targets browsers that support [native ES Modules](https://caniuse.com/es6-module),
[native ESM dynamic import](https://caniuse.com/es6-module-dynamic-import), and [`import.meta`](https://caniuse.com/mdn-
javascript_operators_import_meta). Legacy browsers can be supported via the official [@vitejs/plugin-legacy]
(https://github.com/vitejs/vite/tree/main/packages/plugin-legacy). See the [Building for Production](./build) section for
more details.

## Trying Vite Online

You can try Vite online on [StackBlitz](https://vite.new/). It runs the Vite-based build setup directly in the browser, so it
is almost identical to the local setup but doesn't require installing anything on your machine. You can navigate to
`vite.new/{template}` to select which framework to use.

The supported template presets are:

|             JavaScript              |                TypeScript                 |
| :---------------------------------: | :---------------------------------------: |
| [vanilla](https://vite.new/vanilla) | [vanilla-ts](https://vite.new/vanilla-ts) |
|     [vue](https://vite.new/vue)     |     [vue-ts](https://vite.new/vue-ts)     |
|   [react](https://vite.new/react)   |   [react-ts](https://vite.new/react-ts)   |
|  [preact](https://vite.new/preact)  |  [preact-ts](https://vite.new/preact-ts)  |
|     [lit](https://vite.new/lit)     |     [lit-ts](https://vite.new/lit-ts)     |
|  [svelte](https://vite.new/svelte)  |  [svelte-ts](https://vite.new/svelte-ts)  |
|   [solid](https://vite.new/solid)   |   [solid-ts](https://vite.new/solid-ts)   |
|    [qwik](https://vite.new/qwik)    |    [qwik-ts](https://vite.new/qwik-ts)    |

## Scaffolding Your First Vite Project

::: tip Compatibility Note
Vite requires [Node.js](https://nodejs.org/en/) version 18+ or 20+. However, some templates require a higher Node.js version
to work, please upgrade if your package manager warns about it.
:::

::: code-group

```bash [NPM]
$ npm create vite@latest
```

```bash [Yarn]
$ yarn create vite
```

```bash [PNPM]
$ pnpm create vite
```

```bash [Bun]
$ bun create vite
~~~

~~~markdown
```

:::

Then follow the prompts!

You can also directly specify the project name and the template you want to use via additional command line options. For
example, to scaffold a Vite + Vue project, run:

::: code-group

```bash [NPM]
# npm 7+, extra double-dash is needed:
$ npm create vite@latest my-vue-app -- --template vue
```

```bash [Yarn]
$ yarn create vite my-vue-app --template vue
```

```bash [PNPM]
$ pnpm create vite my-vue-app --template vue
```

```bash [Bun]
$ bun create vite my-vue-app --template vue
```

:::

See [create-vite](https://github.com/vitejs/vite/tree/main/packages/create-vite) for more details on each supported template:
`vanilla`, `vanilla-ts`, `vue`, `vue-ts`, `react`, `react-ts`, `react-swc`, `react-swc-ts`, `preact`, `preact-ts`, `lit`,
`lit-ts`, `svelte`, `svelte-ts`, `solid`, `solid-ts`, `qwik`, `qwik-ts`.

You can use `.` for the project name to scaffold in the current directory.

## Community Templates

create-vite is a tool to quickly start a project from a basic template for popular frameworks. Check out Awesome Vite for
[community maintained templates](https://github.com/vitejs/awesome-vite#templates) that include other tools or target
different frameworks.

For a template at `https://github.com/user/project`, you can try it out online using
`https://github.stackblitz.com/user/project` (adding `.stackblitz` after `github` to the URL of the project).

You can also use a tool like [degit](https://github.com/Rich-Harris/degit) to scaffold your project with one of the
templates. Assuming the project is on GitHub and uses `main` as the default branch, you can create a local copy using:

```bash
npx degit user/project#main my-project
cd my-project

npm install
npm run dev
```

## Manual Installation

In your project, you can install the `vite` CLI using:

::: code-group

```bash [NPM]
$ npm install -D vite
```

```bash [Yarn]
$ yarn add -D vite
```

```bash [PNPM]
$ pnpm add -D vite
```

```bash [Bun]
$ bun add -D vite
```

:::

And create an `index.html` file like this:

```html
~~~

<p>Hello Vite!</p>

Then run the `vite` CLI in your terminal:

```bash
vite
```

The `index.html` will be served on `http://localhost:5173`.

## `index.html` and Project Root

One thing you may have noticed is that in a Vite project, `index.html` is front-and-central instead of being tucked away inside `public`. This is intentional: during development Vite is a server, and `index.html` is the entry point to your application.

Vite treats `index.html` as source code and part of the module graph. It resolves `<script type="module" src="...">` that references your JavaScript source code. Even inline `<script type="module">` and CSS referenced via `<link href>` also enjoy Vite-specific features. In addition, URLs inside `index.html` are automatically rebased so there's no need for special `%PUBLIC_URL%` placeholders.

Similar to static http servers, Vite has the concept of a "root directory" which your files are served from. You will see it referenced as `<root>` throughout the rest of the docs. Absolute URLs in your source code will be resolved using the project root as base, so you can write code as if you are working with a normal static file server (except way more powerful!). Vite is also capable of handling dependencies that resolve to out-of-root file system locations, which makes it usable even in a monorepo-based setup.

Vite also supports [multi-page apps](./build#multi-page-app) with multiple `.html` entry points.

#### Specifying Alternative Root

Running `vite` starts the dev server using the current working directory as root. You can specify an alternative root with `vite serve some/sub/dir`.
Note that Vite will also resolve [its config file (i.e. `vite.config.js`)](/config/#configuring-vite) inside the project root, so you'll need to move it if the root is changed.

## Command Line Interface

In a project where Vite is installed, you can use the `vite` binary in your npm scripts, or run it directly with `npx vite`. Here are the default npm scripts in a scaffolded Vite project:

<!-- prettier-ignore -->
```json
{
  "scripts": {
    "dev": "vite", // start dev server, aliases: `vite dev`, `vite serve`
    "build": "vite build", // build for production
    "preview": "vite preview" // locally preview production build
  }
}
```

You can specify additional CLI options like `--port` or `--open`. For a full list of CLI options, run `npx vite --help` in your project.

Learn more about the [Command Line Interface](./cli.md)

## Using Unreleased Commits

If you can't wait for a new release to test the latest features, you will need to clone the [vite repo](https://github.com/vitejs/vite) to your local machine and then build and link it yourself ([pnpm](https://pnpm.io/) is required):

```bash
git clone https://github.com/vitejs/vite.git
cd vite
pnpm install
cd packages/vite
pnpm run build
pnpm link --global # use your preferred package manager for this step
```

Then go to your Vite based project and run `pnpm link --global vite` (or the package manager that you used to link `vite` globally). Now restart the development server to ride on the bleeding edge!

## Community

If you have questions or need help, reach out to the community at [Discord](https://chat.vitejs.dev) and [GitHub Discussions](https://github.com/vitejs/vite/discussions).

# Migration from v4

## Node.js Support

Vite no longer supports Node.js 14 / 16 / 17 / 19, which reached its EOL. Node.js 18 / 20+ is now required.

## Rollup 4

Vite is now using Rollup 4 which also brings along its breaking changes, in particular:

- Import assertions (`assertions` prop) has been renamed to import attributes (`attributes` prop).
- Acorn plugins are no longer supported.
- For Vite plugins, `this.resolve` `skipSelf` option is now `true` by default.
- For Vite plugins, `this.parse` now only supports the `allowReturnOutsideFunction` option for now.

Read the full breaking changes in [Rollup's release notes](https://github.com/rollup/rollup/releases/tag/v4.0.0) for build-related changes in [`build.rollupOptions`](/config/build-options.md#build-rollupoptions).

If you are using TypeScript, make sure to set `moduleResolution: 'bundler'` (or `node16`/`nodenext`) as Rollup 4 requires it. Or you can set `skipLibCheck: true` instead.

## Deprecate CJS Node API

The CJS Node API of Vite is deprecated. When calling `require('vite')`, a deprecation warning is now logged. You should update your files or frameworks to import the ESM build of Vite instead.

In a basic Vite project, make sure:

1. The `vite.config.js` file content is using the ESM syntax.
2. The closest `package.json` file has `"type": "module"`, or use the `.mjs`/`.mts` extension, e.g. `vite.config.mjs` or `vite.config.mts`.

For other projects, there are a few general approaches:

- **Configure ESM as default, opt-in to CJS if needed:** Add `"type": "module"` in the project `package.json`. All `*.js` files are now interpreted as ESM and need to use the ESM syntax. You can rename a file with the `.cjs` extension to keep using CJS instead.
- **Keep CJS as default, opt-in to ESM if needed:** If the project `package.json` does not have `"type": "module"`, all `*.js` files are interpreted as CJS. You can rename a file with the `.mjs` extension to use ESM instead.
- **Dynamically import Vite:** If you need to keep using CJS, you can dynamically import Vite using `import('vite')` instead. This requires your code to be written in an `async` context, but should still be manageable as Vite's API is mostly asynchronous.

See the [troubleshooting guide](/guide/troubleshooting.html#vite-cjs-node-api-deprecated) for more information.

## Rework `define` and `import.meta.env.*` replacement strategy

In Vite 4, the [`define`](/config/shared-options.md#define) and [`import.meta.env.*`](/guide/env-and-mode.md#env-variables) features use different replacement strategies in dev and build:

- In dev, both features are injected as global variables to `globalThis` and `import.meta` respectively.
- In build, both features are statically replaced with a regex.

This results in a dev and build inconsistency when trying to access the variables, and sometimes even caused failed builds. For example:

```js
// vite.config.js
export default defineConfig({
  define: {
    __APP_VERSION__: JSON.stringify('1.0.0'),
  },
})
```

```js
const data = { __APP_VERSION__ }
// dev: { __APP_VERSION__: "1.0.0" } âœ…
// build: { "1.0.0" } âŒ

const docs = 'I like import.meta.env.MODE'
// dev: "I like import.meta.env.MODE" âœ…
// build: "I like "production"" âŒ
```

Vite 5 fixes this by using `esbuild` to handle the replacements in builds, aligning with the dev behaviour.

This change should not affect most setups, as it's already documented that `define` values should follow esbuild's syntax:

> To be consistent with esbuild behavior, expressions must either be a JSON object (null, boolean, number, string, array, or object) or a single identifier.

However, if you prefer to keep statically replacing values directly, you can use [`@rollup/plugin-replace`] (https://github.com/rollup/plugins/tree/master/packages/replace).

## General Changes

### SSR externalized modules value now matches production

In Vite 4, SSR externalized modules are wrapped with `.default` and `.__esModule` handling for better interoperability, but it doesn't match the production behaviour when loaded by the runtime environment (e.g. Node.js), causing hard-to-catch inconsistencies. By default, all direct project dependencies are SSR externalized.

Vite 5 now removes the `.default` and `.__esModule` handling to match the production behaviour. In practice, this shouldn't affect properly-packaged dependencies, but if you encounter new issues loading modules, you can try these refactors:

```js
// Before:
import { foo } from 'bar'

// After:
import _bar from 'bar'
const { foo } = _bar
```

```js
// Before:
import foo from 'bar'

// After:
import * as _foo from 'bar'
const foo = _foo.default
```

Note that these changes match the Node.js behaviour, so you can also run the imports in Node.js to test it out. If you prefer to stick with the previous behaviour, you can set `legacy.proxySsrExternalModules` to `true`.

### `worker.plugins` is now a function

In Vite 4, [`worker.plugins`](/config/worker-options.md#worker-plugins) accepted an array of plugins (`(Plugin | Plugin[])[]`). From Vite 5, it needs to be configured as a function that returns an array of plugins (`() => (Plugin | Plugin[])[]`). This change is required so parallel worker builds run more consistently and predictably.

### Allow path containing `.` to fallback to index.html

In Vite 4, accessing a path in dev containing `.` did not fallback to index.html even if [`appType`](/config/shared-options.md#apptype) is set to `'spa'` (default). From Vite 5, it will fallback to index.html.

Note that the browser will no longer show a 404 error message in the console if you point the image path to a non-existent file (e.g. `<img src="./file-does-not-exist.png">`).

### Align dev and preview HTML serving behaviour

In Vite 4, the dev and preview servers serve HTML based on its directory structure and trailing slash differently. This causes inconsistencies when testing your built app. Vite 5 refactors into a single behaviour like below, given the following file structure:

```
â”œâ”€â”€ index.html
â”œâ”€â”€ file.html
â””â”€â”€ dir
    â””â”€â”€ index.html
```

Request	Before (dev)	Before (preview)	After (dev & preview)
`/dir/index.html`	`/dir/index.html`	`/dir/index.html`	`/dir/index.html`
`/dir`	`/index.html` (SPA fallback)	`/dir/index.html`	`/index.html` (SPA fallback)
`/dir/`	`/dir/index.html`	`/dir/index.html`	`/dir/index.html`
`/file.html`	`/file.html`	`/file.html`	`/file.html`
`/file`	`/index.html` (SPA fallback)	`/file.html`	`/file.html`
`/file/`	`/index.html` (SPA fallback)	`/file.html`	`/index.html` (SPA fallback)

### Manifest files are now generated in `.vite` directory by default

In Vite 4, the manifest files ([`build.manifest`](/config/build-options.md#build-manifest) and [`build.ssrManifest`](/config/build-options.md#build-ssrmanifest)) were generated in the root of [`build.outDir`](/config/build-options.md#build-outdir) by default.

From Vite 5, they will be generated in the `.vite` directory in the `build.outDir` by default. This change helps deconflict public files with the same manifest file names when they are copied to the `build.outDir`.

### Corresponding CSS files are not listed as top level entry in manifest.json file

In Vite 4, the corresponding CSS file for a JavaScript entry point was also listed as a top-level entry in the manifest file ([`build.manifest`](/config/build-options.md#build-manifest)). These entries were unintentionally added and only worked for simple cases.

In Vite 5, corresponding CSS files can only be found within the JavaScript entry file section.
When injecting the JS file, the corresponding CSS files [should be injected](/guide/backend-integration.md#:~:text=%3C!%2D%2D%20if%20production%20%2D%2D%3E%0A%3Clink%20rel%3D%22stylesheet%22%20href%3D%22/assets/%7B%7B%20manifest%5B%27main.js%27%5D.css%20%7D%7D%22%20/%3E%0A%3Cscript%20type%3D%22module%22%20src%3D%22/assets/%7B%7B%20manifest%5B%27main.js%27%5D.file%20%7D%7D%22%3E%3C/script%3E).
When the CSS should be injected separately, it must be added as a separate entry point.

### CLI shortcuts require an additional `Enter` press

CLI shortcuts, like `r` to restart the dev server, now require an additional `Enter` press to trigger the shortcut. For example, `r + Enter` to restart the dev server.

This change prevents Vite from swallowing and controlling OS-specific shortcuts, allowing better compatibility when combining the Vite dev server with other processes, and avoids the [previous caveats](https://github.com/vitejs/vite/pull/14342).

### Update `experimentalDecorators` and `useDefineForClassFields` TypeScript behaviour

Vite 5 uses esbuild 0.19 and removes the compatibility layer for esbuild 0.18, which changes how [`experimentalDecorators`](https://www.typescriptlang.org/tsconfig#experimentalDecorators) and [`useDefineForClassFields`](https://www.typescriptlang.org/tsconfig#useDefineForClassFields) are handled.

- **`experimentalDecorators` is not enabled by default**

  You need to set `compilerOptions.experimentalDecorators` to `true` in `tsconfig.json` to use decorators.

- **`useDefineForClassFields` defaults depend on the TypeScript `target` value**

  If `target` is not `ESNext` or `ES2022` or newer, or if there's no `tsconfig.json` file, `useDefineForClassFields` will default to `false` which can be problematic with the default `esbuild.target` value of `esnext`. It may transpile to [static initialization blocks](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Static_initialization_blocks#browser_compatibility) which may not be supported in your browser.

  As such, it is recommended to set `target` to `ESNext` or `ES2022` or newer, or set `useDefineForClassFields` to `true` explicitly when configuring `tsconfig.json`.

```jsonc
{
  "compilerOptions": {
    // Set true if you use decorators
    "experimentalDecorators": true,
    // Set true if you see parsing errors in your browser
    "useDefineForClassFields": true,
  },
}
```

### Remove `--https` flag and `https: true`

The `--https` flag sets `server.https: true` and `preview.https: true` internally. This config was meant to be used together with the automatic https certification generation feature which [was dropped in Vite 3](https://v3.vitejs.dev/guide/migration.html#automatic-https-certificate-generation). Hence, this config is no longer useful as it will start a Vite HTTPS server without a certificate.

If you use [`@vitejs/plugin-basic-ssl`](https://github.com/vitejs/vite-plugin-basic-ssl) or [`vite-plugin-mkcert`](https://github.com/liuweiGL/vite-plugin-mkcert), they will already set the `https` config internally, so you can remove `--https`, `server.https: true`, and `preview.https: true` in your setup.

### Remove `resolvePackageEntry` and `resolvePackageData` APIs

The `resolvePackageEntry` and `resolvePackageData` APIs are removed as they exposed Vite's internals and blocked potential Vite 4.3 optimizations in the past. These APIs can be replaced with third-party packages, for example:

- `resolvePackageEntry`: [`import.meta.resolve`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import.meta/resolve) or the [`import-meta-resolve`](https://github.com/wooorm/import-meta-resolve) package.
- `resolvePackageData`: Same as above, and crawl up the package directory to get the root `package.json`. Or use the community [`vitefu`](https://github.com/svitejs/vitefu) package.

```js
import { resolve } from 'import-meta-resolve'
import { findDepPkgJsonPath } from 'vitefu'
import fs from 'node:fs'

const pkg = 'my-lib'
const basedir = process.cwd()

// `resolvePackageEntry`:
const packageEntry = resolve(pkg, basedir)

// `resolvePackageData`:
```

const packageJsonPath = findDepPkgJsonPath(pkg, basedir)
const packageJson = JSON.parse(fs.readFileSync(packageJsonPath, 'utf-8'))
```

## Removed Deprecated APIs

- Default exports of CSS files (e.g `import style from './foo.css'`): Use the `?inline` query instead
- `import.meta.globEager`: Use `import.meta.glob('*', { eager: true })` instead
- `ssr.format: 'cjs'` and `legacy.buildSsrCjsExternalHeuristics` ([#13816](https://github.com/vitejs/vite/discussions/13816))
- `server.middlewareMode: 'ssr'` and `server.middlewareMode: 'html'`: Use [`appType`](/config/shared-options.md#apptype) + [`server.middlewareMode: true`](/config/server-options.md#server-middlewaremode) instead ([#8452](https://github.com/vitejs/vite/pull/8452))

## Advanced

There are some changes which only affect plugin/tool creators.

- [[#14119] refactor!: merge `PreviewServerForHook` into `PreviewServer` type](https://github.com/vitejs/vite/pull/14119)
  - The `configurePreviewServer` hook now accepts the `PreviewServer` type instead of `PreviewServerForHook` type.
- [[#14818] refactor(preview)!: use base middleware](https://github.com/vitejs/vite/pull/14818)
  - Middlewares added from the returned function in `configurePreviewServer` now does not have access to the `base` when comparing the `req.url` value. This aligns the behaviour with the dev server. You can check the `base` from the `configResolved` hook if needed.
- [[#14834] fix(types)!: expose httpServer with Http2SecureServer union](https://github.com/vitejs/vite/pull/14834)
  - `http.Server | http2.Http2SecureServer` is now used instead of `http.Server` where appropriate.

Also there are other breaking changes which only affect few users.

- [[#14098] fix!: avoid rewriting this (reverts #5312)](https://github.com/vitejs/vite/pull/14098)
  - Top level `this` was rewritten to `globalThis` by default when building. This behavior is now removed.
- [[#14231] feat!: add extension to internal virtual modules](https://github.com/vitejs/vite/pull/14231)
  - Internal virtual modules' id now has an extension (`.js`).
- [[#14583] refactor!: remove exporting internal APIs](https://github.com/vitejs/vite/pull/14583)
  - Removed accidentally exported internal APIs: `isDepsOptimizerEnabled` and `getDepOptimizationConfig`
  - Removed exported internal types: `DepOptimizationResult`, `DepOptimizationProcessing`, and `DepsOptimizer`
  - Renamed `ResolveWorkerOptions` type to `ResolvedWorkerOptions`
- [[#5657] fix: return 404 for resources requests outside the base path](https://github.com/vitejs/vite/pull/5657)
  - In the past, Vite responded to requests outside the base path without `Accept: text/html`, as if they were requested with the base path. Vite no longer does that and responds with 404 instead.
- [[#14723] fix(resolve)!: remove special .mjs handling](https://github.com/vitejs/vite/pull/14723)
  - In the past, when a library `"exports"` field maps to an `.mjs` file, Vite will still try to match the `"browser"` and `"module"` fields to fix compatibility with certain libraries. This behavior is now removed to align with the exports resolution algorithm.
- [[#14733] feat(resolve)!: remove `resolve.browserField`](https://github.com/vitejs/vite/pull/14733)
  - `resolve.browserField` has been deprecated since Vite 3 in favour of an updated default of `['browser', 'module', 'jsnext:main', 'jsnext']` for [`resolve.mainFields`](/config/shared-options.md#resolve-mainfields).
- [[#14855] feat!: add isPreview to ConfigEnv and resolveConfig](https://github.com/vitejs/vite/pull/14855)
  - Renamed `ssrBuild` to `isSsrBuild` in the `ConfigEnv` object.
- [[#14945] fix(css): correctly set manifest source name and emit CSS file](https://github.com/vitejs/vite/pull/14945)
  - CSS file names are now generated based on the chunk name.

## Migration from v3

Check the [Migration from v3 Guide](https://v4.vitejs.dev/guide/migration.html) in the Vite v4 docs first to see the needed changes to port your app to Vite v4, and then proceed with the changes on this page.
```

# Performance

While Vite is fast by default, performance issues can creep in as the project's requirements grow. This guide aims to help you identify and fix common performance issues, such as:

- Slow server starts
- Slow page loads
- Slow builds

## Review your Browser Setup

Some browser extensions may interfere with requests and slow down startup and reload times for large apps, especially when using browser dev tools. We recommend creating a dev-only profile without extensions, or switch to incognito mode, while using Vite's dev server in these cases. Incognito mode should also be faster than a regular profile without extensions.

The Vite dev server does hard caching of pre-bundled dependencies and implements fast 304 responses for source code. Disabling the cache while the Browser Dev Tools are open can have a big impact on startup and full-page reload times. Please check that "Disable Cache" isn't enabled while you work with the Vite server.

## Audit Configured Vite Plugins

Vite's internal and official plugins are optimized to do the least amount of work possible while providing compatibility with the broader ecosystem. For example, code transformations use regex in dev, but do a complete parse in build to ensure correctness.

However, the performance of community plugins is out of Vite's control, which may affect the developer experience. Here are a few things you can look out for when using additional Vite plugins:

1. Large dependencies that are only used in certain cases should be dynamically imported to reduce the Node.js startup time. Example refactors: [vite-plugin-react#212](https://github.com/vitejs/vite-plugin-react/pull/212) and [vite-plugin-pwa#224](https://github.com/vite-pwa/vite-plugin-pwa/pull/244).

2. The `buildStart`, `config`, and `configResolved` hooks should not run long and extensive operations. These hooks are awaited during dev server startup, which delays when you can access the site in the browser.

3. The `resolveId`, `load`, and `transform` hooks may cause some files to load slower than others. While sometimes unavoidable, it's still worth checking for possible areas to optimize. For example, checking if the `code` contains a specific keyword, or the `id` matches a specific extension, before doing the full transformation.

   The longer it takes to transform a file, the more significant the request waterfall will be when loading the site in the browser.

   You can inspect the duration it takes to transform a file using `vite --debug plugin-transform` or [vite-plugin-inspect](https://github.com/antfu/vite-plugin-inspect). Note that as asynchronous operations tend to provide inaccurate timings, you should treat the numbers as a rough estimate, but it should still reveal the more expensive operations.

::: tip Profiling
You can run `vite --profile`, visit the site, and press `p + enter` in your terminal to record a `.cpuprofile`. A tool like [speedscope](https://www.speedscope.app) can then be used to inspect the profile and identify the bottlenecks. You can also [share the profiles](https://chat.vitejs.dev) with the Vite team to help us identify performance issues.
:::

## Reduce Resolve Operations

Resolving import paths can be an expensive operation when hitting its worst case often. For example, Vite supports "guessing" import paths with the [`resolve.extensions`](/config/shared-options.md#resolve-extensions) option, which defaults to `['.mjs', '.js', '.mts', '.ts', '.jsx', '.tsx', '.json']`.

When you try to import `./Component.jsx` with `import './Component'`, Vite will run these steps to resolve it:

1. Check if `./Component` exists, no.
2. Check if `./Component.mjs` exists, no.
3. Check if `./Component.js` exists, no.
4. Check if `./Component.mts` exists, no.
5. Check if `./Component.ts` exists, no.
6. Check if `./Component.jsx` exists, yes!

As shown, a total of 6 filesystem checks is required to resolve an import path. The more implicit imports you have, the more time it adds up to resolve the paths.

Hence, it's usually better to be explicit with your import paths, e.g. `import './Component.jsx'`. You can also narrow down the list for `resolve.extensions` to reduce the general filesystem checks, but you have to make sure it works for files in `node_modules` too.

If you're a plugin author, make sure to only call [`this.resolve`](https://rollupjs.org/plugin-development/#this-resolve) when needed to reduce the number of checks above.

::: tip TypeScript
If you are using TypeScript, enable `"moduleResolution": "bundler"` and `"allowImportingTsExtensions": true` in your `tsconfig.json`'s `compilerOptions` to use `.ts` and `.tsx` extensions directly in your code.
:::

## Avoid Barrel Files

Barrel files are files that re-export the APIs of other files in the same directory. For example:

```js
// src/utils/index.js
export * from './color.js'
export * from './dom.js'
export * from './slash.js'
```

When you only import an individual API, e.g. `import { slash } from './utils'`, all the files in that barrel file need to be fetched and transformed as they may contain the `slash` API and may also contain side-effects that run on initialization. This means you're loading more files than required on the initial page load, resulting in a slower page load.

If possible, you should avoid barrel files and import the individual APIs directly, e.g. `import { slash } from './utils/slash.js'`. You can read [issue #8237](https://github.com/vitejs/vite/issues/8237) for more information.

## Warm Up Frequently Used Files

The Vite dev server only transforms files as requested by the browser, which allows it to start up quickly and only apply transformations for used files. It can also pre-transform files if it anticipates certain files will be requested shortly. However, request waterfalls may still happen if some files take longer to transform than others. For example:

Given an import graph where the left file imports the right file:

```
main.js -> BigComponent.vue -> big-utils.js -> large-data.json
```

The import relationship can only be known after the file is transformed. If `BigComponent.vue` takes some time to transform, `big-utils.js` has to wait for its turn, and so on. This causes an internal waterfall even with pre-transformation built-in.

Vite allows you to warm up files that you know are frequently used, e.g. `big-utils.js`, using the [`server.warmup`](/config/server-options.md#server-warmup) option. This way `big-utils.js` will be ready and cached to be served immediately when requested.

You can find files that are frequently used by running `vite --debug transform` and inspect the logs:

```bash
vite:transform 28.72ms /@vite/client +1ms
vite:transform 62.95ms /src/components/BigComponent.vue +1ms
vite:transform 102.54ms /src/utils/big-utils.js +1ms
```

```js
export default defineConfig({
  server: {
    warmup: {
      clientFiles: [
        './src/components/BigComponent.vue',
        './src/utils/big-utils.js',
      ],
    },
  },
})
```

Note that you should only warm up files that are frequently used to not overload the Vite dev server on startup. Check the [`server.warmup`](/config/server-options.md#server-warmup) option for more information.

Using [`--open` or `server.open`](/config/server-options.html#server-open) also provides a performance boost, as Vite will automatically warm up the entry point of your app or the provided URL to open.

## Use Lesser or Native Tooling

Keeping Vite fast with a growing codebase is about reducing the amount of work for the source files (JS/TS/CSS).

Examples of doing less work:

- Use CSS instead of Sass/Less/Stylus when possible (nesting can be handled by PostCSS)
- Don't transform SVGs into UI framework components (React, Vue, etc). Import them as strings or URLs instead.
- When using `@vitejs/plugin-react`, avoid configuring the Babel options, so it skips the transformation during build (only esbuild will be used).

Examples of using native tooling:

Using native tooling often brings larger installation size and as so is not the default when starting a new Vite project. But it may be worth the cost for larger applications.

- Try out the experimental support for [LightningCSS](https://github.com/vitejs/vite/discussions/13835)
- Use [`@vitejs/plugin-react-swc`](https://github.com/vitejs/vite-plugin-react-swc) in place of `@vitejs/plugin-react`.

# Project Philosophy

## Lean Extendable Core

Vite doesn't intend to cover every use case for every user. Vite aims to support the most common patterns to build Web apps out-of-the-box, but [Vite core](https://github.com/vitejs/vite) must remain lean with a small API surface to keep the project maintainable long-term. This goal is possible thanks to [Vite's rollup-based plugin system](./api-plugin.md). Features that can be implemented as external plugins will generally not be added to Vite core. [vite-plugin-pwa](https://vite-pwa-org.netlify.app/) is a great example of what can be achieved out of Vite core, and there are a lot of [well maintained plugins](https://github.com/vitejs/awesome-vite#plugins) to cover your needs. Vite works closely with the Rollup project to ensure that plugins can be used in both plain-rollup and Vite projects as much as possible, trying to push needed extensions to the Plugin API upstream when possible.

## Pushing the Modern Web

Vite provides opinionated features that push writing modern code. For example:

- The source code can only be written in ESM, where non-ESM dependencies need to be [pre-bundled as ESM](./dep-pre-bundling) in order to work.
- Web workers are encouraged to be written with the [`new Worker` syntax](./features#web-workers) to follow modern standards.
- Node.js modules cannot be used in the browser.

When adding new features, these patterns are followed to create a future-proof API, which may not always be compatible with other build tools.

## A Pragmatic Approach to Performance

Vite has been focused on performance since its [origins](./why.md). Its dev server architecture allows HMR that stays fast as projects scale. Vite uses native tools like [esbuild](https://esbuild.github.io/) and [SWC](https://github.com/vitejs/vite-plugin-react-swc) to implement intensive tasks but keeps the rest of the code in JS to balance speed with flexibility. When needed, framework plugins will tap into [Babel](https://babeljs.io/) to compile user code. And during build time Vite currently uses [Rollup](https://rollupjs.org/) where bundling size and having access to a wide ecosystem of plugins are more important than raw speed. Vite will continue to evolve internally, using new libraries as they appear to improve DX while keeping its API stable.

## Building Frameworks on top of Vite

Although Vite can be used by users directly, it shines as a tool to create frameworks. Vite core is framework agnostic, but there are polished plugins for each UI framework. Its [JS API](./api-javascript.md) allows App Framework authors to use Vite features to create tailored experiences for their users. Vite includes support for [SSR primitives](./ssr.md), usually present in higher-level tools but fundamental to building modern web frameworks. And Vite plugins complete the picture by offering a way to share between frameworks. Vite is also a great fit when paired with [Backend frameworks](./backend-integration.md) like [Ruby](https://vite-ruby.netlify.app/) and [Laravel](https://laravel.com/docs/10.x/vite).

## An Active Ecosystem

Vite evolution is a cooperation between framework and plugin maintainers, users, and the Vite team. We encourage active participation in Vite's Core development once a project adopts Vite. We work closely with the main projects in the ecosystem to minimize regressions on each release, aided by tools like [vite-ecosystem-ci](https://github.com/vitejs/vite-ecosystem-ci). It allows us to run the CI of major projects using Vite on selected PRs and gives us a clear status of how the Ecosystem would react to a release. We strive to fix regressions before they hit users and allow projects to update to the next versions as soon as they are released. If you are working with Vite, we invite you to join [Vite's Discord](https://chat.vitejs.dev) and get involved in the project too.

# Server-Side Rendering

:::tip Note
SSR specifically refers to front-end frameworks (for example React, Preact, Vue, and Svelte) that support running the same application in Node.js, pre-rendering it to HTML, and finally hydrating it on the client. If you are looking for integration with traditional server-side frameworks, check out the [Backend Integration guide](./backend-integration) instead.

The following guide also assumes prior experience working with SSR in your framework of choice, and will only focus on Vite-specific integration details.
:::

:::warning Low-level API
This is a low-level API meant for library and framework authors. If your goal is to create an application, make sure to check out the higher-level SSR plugins and tools at [Awesome Vite SSR section](https://github.com/vitejs/awesome-vite#ssr) first. That said, many applications are successfully built directly on top of Vite's native low-level API.

Currently, Vite is working on an improved SSR API with the [Environment API](https://github.com/vitejs/vite/discussions/16358). Check out the link for more details.
:::

:::tip Help
If you have questions, the community is usually helpful at [Vite Discord's #ssr channel](https://discord.gg/PkbxgzPhJv).
:::

## Example Projects

Vite provides built-in support for server-side rendering (SSR). [`create-vite-extra`](https://github.com/bluwy/create-vite-extra) contains example SSR setups you can use as references for this guide:

- [Vanilla](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-vanilla)
- [Vue](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-vue)
- [React](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-react)
- [Preact](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-preact)
- [Svelte](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-svelte)
- [Solid](https://github.com/bluwy/create-vite-extra/tree/master/template-ssr-solid)

You can also scaffold these projects locally by [running `create-vite`](./index.md#scaffolding-your-first-vite-project) and choose `Others > create-vite-extra` under the framework option.

## Source Structure

A typical SSR application will have the following source file structure:

```
- index.html
- server.js # main application server
- src/
  - main.js          # exports env-agnostic (universal) app code
  - entry-client.js  # mounts the app to a DOM element
  - entry-server.js  # renders the app using the framework's SSR API
```

The `index.html` will need to reference `entry-client.js` and include a placeholder where the server-rendered markup should be injected:

```html
<div id="app"><!--ssr-outlet--></div>
<script type="module" src="/src/entry-client.js"></script>
```

You can use any placeholder you prefer instead of `<!--ssr-outlet-->`, as long as it can be precisely replaced.

## Conditional Logic

If you need to perform conditional logic based on SSR vs. client, you can use

```js twoslash
import 'vite/client'
// ---cut---
if (import.meta.env.SSR) {
  // ... server only logic
}
```

This is statically replaced during build so it will allow tree-shaking of unused branches.

## Setting Up the Dev Server

When building an SSR app, you likely want to have full control over your main server and decouple Vite from the production environment. It is therefore recommended to use Vite in middleware mode. Here is an example with [express](https://expressjs.com/):

server.js

```js{15-18} twoslash
import fs from 'node:fs'
import path from 'node:path'
import { fileURLToPath } from 'node:url'
import express from 'express'
import { createServer as createViteServer } from 'vite'

const __dirname = path.dirname(fileURLToPath(import.meta.url))

async function createServer() {
  const app = express()

  // Create Vite server in middleware mode and configure the app type as
  // 'custom', disabling Vite's own HTML serving logic so parent server
  // can take control
  const vite = await createViteServer({
    server: { middlewareMode: true },
    appType: 'custom'
  })

  // Use vite's connect instance as middleware. If you use your own
  // express router (express.Router()), you should use router.use
  // When the server restarts (for example after the user modifies
  // vite.config.js), `vite.middlewares` is still going to be the same
  // reference (with a new internal stack of Vite and plugin-injected
  // middlewares). The following is valid even after restarts.
  app.use(vite.middlewares)

  app.use('*', async (req, res) => {
    // serve index.html - we will tackle this next
  })

  app.listen(5173)
}

createServer()
```

Here `vite` is an instance of [ViteDevServer](./api-javascript#vitedevserver). `vite.middlewares` is a [Connect](https://github.com/senchalabs/connect) instance which can be used as a middleware in any connect-compatible Node.js framework.

The next step is implementing the `*` handler to serve server-rendered HTML:

```js twoslash
// @noErrors
import fs from 'node:fs'
import path from 'node:path'
import { fileURLToPath } from 'node:url'

/** @type {import('express').Express} */
var app
/** @type {import('vite').ViteDevServer}  */
var vite

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

  try {
    // 1. Read index.html
    let template = fs.readFileSync(
      path.resolve(__dirname, 'index.html'),
      '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 vite.transformIndexHtml(url, template)

    // 3. Load the server entry. ssrLoadModule automatically transforms
    //    ESM source code to be usable in Node.js! There is no bundling
    //    required, and provides efficient invalidation similar to HMR.
    const { render } = await vite.ssrLoadModule('/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.
```

~~~markdown
    const html = template.replace(`<!--ssr-outlet-->`, appHtml)

    // 6. Send the rendered HTML back.
    res.status(200).set({ 'Content-Type': 'text/html' }).end(html)
  } catch (e) {
    // If an error is caught, let Vite fix the stack trace so it maps back
    // to your actual source code.
    vite.ssrFixStacktrace(e)
    next(e)
  }
})
```

The `dev` script in `package.json` should also be changed to use the server script instead:

```diff
  "scripts": {
-   "dev": "vite"
+   "dev": "node server"
  }
```

## Building for Production

To ship an SSR project for production, we need to:

1. Produce a client build as normal;
2. Produce an SSR build, which can be directly loaded via `import()` so that we don't have to go through Vite's
`ssrLoadModule`;

Our scripts in `package.json` will look like this:

```json
{
  "scripts": {
    "dev": "node server",
    "build:client": "vite build --outDir dist/client",
    "build:server": "vite build --outDir dist/server --ssr src/entry-server.js"
  }
}
```

Note the `--ssr` flag which indicates this is an SSR build. It should also specify the SSR entry.

Then, in `server.js` we need to add some production specific logic by checking `process.env.NODE_ENV`:

- Instead of reading the root `index.html`, use the `dist/client/index.html` as the template, since it contains the correct
asset links to the client build.

- Instead of `await vite.ssrLoadModule('/src/entry-server.js')`, use `import('./dist/server/entry-server.js')` (this file is
the result of the SSR build).

- Move the creation and all usage of the `vite` dev server behind dev-only conditional branches, then add static file serving
middlewares to serve files from `dist/client`.

Refer to the [example projects](#example-projects) for a working setup.

## Generating Preload Directives

`vite build` supports the `--ssrManifest` flag which will generate `.vite/ssr-manifest.json` in build output directory:

```diff
- "build:client": "vite build --outDir dist/client",
+ "build:client": "vite build --outDir dist/client --ssrManifest",
```

The above script will now generate `dist/client/.vite/ssr-manifest.json` for the client build (Yes, the SSR manifest is
generated from the client build because we want to map module IDs to client files). The manifest contains mappings of module
IDs to their associated chunks and asset files.

To leverage the manifest, frameworks need to provide a way to collect the module IDs of the components that were used during
a server render call.

`@vitejs/plugin-vue` supports this out of the box and automatically registers used component module IDs on to the associated
Vue SSR context:

```js
// src/entry-server.js
const ctx = {}
const html = await vueServerRenderer.renderToString(app, ctx)
// ctx.modules is now a Set of module IDs that were used during the render
```
~~~

In the production branch of `server.js` we need to read and pass the manifest to the `render` function exported by `src/entry-server.js`. This would provide us with enough information to render preload directives for files used by async routes! See [demo source](https://github.com/vitejs/vite-plugin-vue/blob/main/playground/ssr-vue/src/entry-server.js) for a full example. You can also use this information for [103 Early Hints](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/103).

## Pre-Rendering / SSG

If the routes and the data needed for certain routes are known ahead of time, we can pre-render these routes into static HTML using the same logic as production SSR. This can also be considered a form of Static-Site Generation (SSG). See [demo pre-render script](https://github.com/vitejs/vite-plugin-vue/blob/main/playground/ssr-vue/prerender.js) for working example.

## SSR Externals

Dependencies are "externalized" from Vite's SSR transform module system by default when running SSR. This speeds up both dev and build.

If a dependency needs to be transformed by Vite's pipeline, for example, because Vite features are used untranspiled in them, they can be added to [`ssr.noExternal`](../config/ssr-options.md#ssr-noexternal).

For linked dependencies, they are not externalized by default to take advantage of Vite's HMR. If this isn't desired, for example, to test dependencies as if they aren't linked, you can add it to [`ssr.external`](../config/ssr-options.md#ssr-external).

:::warning Working with Aliases
If you have configured aliases that redirect one package to another, you may want to alias the actual `node_modules` packages instead to make it work for SSR externalized dependencies. Both [Yarn](https://classic.yarnpkg.com/en/docs/cli/add/#toc-yarn-add-alias) and [pnpm](https://pnpm.io/aliases/) support aliasing via the `npm:` prefix.
:::

## SSR-specific Plugin Logic

Some frameworks such as Vue or Svelte compile components into different formats based on client vs. SSR. To support conditional transforms, Vite passes an additional `ssr` property in the `options` object of the following plugin hooks:

- `resolveId`
- `load`
- `transform`

Example:

```js twoslash
/** @type {() => import('vite').Plugin} */
// ---cut---
export function mySSRPlugin() {
  return {
    name: 'my-ssr',
    transform(code, id, options) {
      if (options?.ssr) {
        // perform ssr-specific transform...
      }
    },
  }
}
```

The options object in `load` and `transform` is optional, rollup is not currently using this object but may extend these hooks with additional metadata in the future.

:::tip Note
Before Vite 2.7, this was informed to plugin hooks with a positional `ssr` param instead of using the `options` object. All major frameworks and plugins are updated but you may find outdated posts using the previous API.
:::

## SSR Target

The default target for the SSR build is a node environment, but you can also run the server in a Web Worker. Packages entry resolution is different for each platform. You can configure the target to be Web Worker using the `ssr.target` set to `'webworker'`.

## SSR Bundle

In some cases like `webworker` runtimes, you might want to bundle your SSR build into a single JavaScript file. You can enable this behavior by setting `ssr.noExternal` to `true`. This will do two things:

- Treat all dependencies as `noExternal`
- Throw an error if any Node.js built-ins are imported

## SSR Resolve Conditions

By default package entry resolution will use the conditions set in [`resolve.conditions`](../config/shared-options.md#resolve-conditions) for the SSR build. You can use [`ssr.resolve.conditions`](../config/ssr-options.md#ssr-resolve-conditions) and [`ssr.resolve.externalConditions`](../config/ssr-options.md#ssr-resolve-externalconditions) to customize this behavior.

## Vite CLI

The CLI commands `$ vite dev` and `$ vite preview` can also be used for SSR apps. You can add your SSR middlewares to the development server with [`configureServer`](/guide/api-plugin#configureserver) and to the preview server with [`configurePreviewServer`](/guide/api-plugin#configurepreviewserver).

:::tip Note
Use a post hook so that your SSR middleware runs _after_ Vite's middlewares.
:::

# Deploying a Static Site

The following guides are based on some shared assumptions:

- You are using the default build output location (`dist`). This location [can be changed using `build.outDir`](/config/build-options.md#build-outdir), and you can extrapolate instructions from these guides in that case.
- You are using npm. You can use equivalent commands to run the scripts if you are using Yarn or other package managers.
- Vite is installed as a local dev dependency in your project, and you have setup the following npm scripts:

```json
{
  "scripts": {
    "build": "vite build",
    "preview": "vite preview"
  }
}
```

It is important to note that `vite preview` is intended for previewing the build locally and not meant as a production server.

::: tip NOTE
These guides provide instructions for performing a static deployment of your Vite site. Vite also supports Server Side Rendering. SSR refers to front-end frameworks that support running the same application in Node.js, pre-rendering it to HTML, and finally hydrating it on the client. Check out the [SSR Guide](./ssr) to learn about this feature. On the other hand, if you are looking for integration with traditional server-side frameworks, check out the [Backend Integration guide](./backend-integration) instead.
:::

## Building the App

You may run `npm run build` command to build the app.

```bash
$ npm run build
```

By default, the build output will be placed at `dist`. You may deploy this `dist` folder to any of your preferred platforms.

### Testing the App Locally

Once you've built the app, you may test it locally by running `npm run preview` command.

```bash
$ npm run preview
```

The `vite preview` command will boot up a local static web server that serves the files from `dist` at `http://localhost:4173`. It's an easy way to check if the production build looks OK in your local environment.

You may configure the port of the server by passing the `--port` flag as an argument.

```json
{
  "scripts": {
    "preview": "vite preview --port 8080"
  }
}
```

Now the `preview` command will launch the server at `http://localhost:8080`.

## GitHub Pages

1. Set the correct `base` in `vite.config.js`.

   If you are deploying to `https://<USERNAME>.github.io/`, or to a custom domain through GitHub Pages (eg. `www.example.com`), set `base` to `'/'`. Alternatively, you can remove `base` from the configuration, as it defaults to `'/'`.

   If you are deploying to `https://<USERNAME>.github.io/<REPO>/` (eg. your repository is at `https://github.com/<USERNAME>/<REPO>`), then set `base` to `'/<REPO>/'`.

2. Go to your GitHub Pages configuration in the repository settings page and choose the source of deployment as "GitHub Actions", this will lead you to create a workflow that builds and deploys your project, a sample workflow that installs dependencies and builds using npm is provided:

   ```yml
   # Simple workflow for deploying static content to GitHub Pages
   name: Deploy static content to Pages

   on:
     # Runs on pushes targeting the default branch
   ```

~~~~
     push:
       branches: ['main']

     # Allows you to run this workflow manually from the Actions tab
     workflow_dispatch:

   # Sets the GITHUB_TOKEN permissions to allow deployment to GitHub Pages
   permissions:
     contents: read
     pages: write
     id-token: write

   # Allow one concurrent deployment
   concurrency:
     group: 'pages'
     cancel-in-progress: true

   jobs:
     # Single deploy job since we're just deploying
     deploy:
       environment:
         name: github-pages
         url: ${{ steps.deployment.outputs.page_url }}
       runs-on: ubuntu-latest
       steps:
         - name: Checkout
           uses: actions/checkout@v4
         - name: Set up Node
           uses: actions/setup-node@v4
           with:
             node-version: 20
             cache: 'npm'
         - name: Install dependencies
           run: npm ci
         - name: Build
           run: npm run build
         - name: Setup Pages
           uses: actions/configure-pages@v4
         - name: Upload artifact
           uses: actions/upload-pages-artifact@v3
           with:
             # Upload dist folder
             path: './dist'
         - name: Deploy to GitHub Pages
           id: deployment
           uses: actions/deploy-pages@v4
   ```

## GitLab Pages and GitLab CI

1. Set the correct `base` in `vite.config.js`.

   If you are deploying to `https://<USERNAME or GROUP>.gitlab.io/`, you can omit `base` as it defaults to `'/'`.

   If you are deploying to `https://<USERNAME or GROUP>.gitlab.io/<REPO>/`, for example your repository is at
`https://gitlab.com/<USERNAME>/<REPO>`, then set `base` to `'/<REPO>/'`.

2. Create a file called `.gitlab-ci.yml` in the root of your project with the content below. This will build and deploy your
site whenever you make changes to your content:

   ```yaml
   image: node:16.5.0
   pages:
     stage: deploy
     cache:
       key:
         files:
           - package-lock.json
         prefix: npm
       paths:
         - node_modules/
     script:
       - npm install
       - npm run build
       - cp -a dist/. public/
     artifacts:
       paths:
         - public
     rules:
       - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
   ```

## Netlify
~~~~

### Netlify CLI

1. Install the [Netlify CLI](https://cli.netlify.com/).
2. Create a new site using `ntl init`.
3. Deploy using `ntl deploy`.

```bash
# Install the Netlify CLI
$ npm install -g netlify-cli

# Create a new site in Netlify
$ ntl init

# Deploy to a unique preview URL
$ ntl deploy
```

The Netlify CLI will share with you a preview URL to inspect. When you are ready to go into production, use the `prod` flag:

```bash
# Deploy the site into production
$ ntl deploy --prod
```

### Netlify with Git

1. Push your code to a git repository (GitHub, GitLab, BitBucket, Azure DevOps).
2. [Import the project](https://app.netlify.com/start) to Netlify.
3. Choose the branch, output directory, and set up environment variables if applicable.
4. Click on **Deploy**.
5. Your Vite app is deployed!

After your project has been imported and deployed, all subsequent pushes to branches other than the production branch along with pull requests will generate [Preview Deployments](https://docs.netlify.com/site-deploys/deploy-previews/), and all changes made to the Production Branch (commonly â€œmainâ€) will result in a [Production Deployment](https://docs.netlify.com/site-deploys/overview/#definitions).

## Vercel

### Vercel CLI

1. Install the [Vercel CLI](https://vercel.com/cli) and run `vercel` to deploy.
2. Vercel will detect that you are using Vite and will enable the correct settings for your deployment.
3. Your application is deployed! (e.g. vite-vue-template.vercel.app)

```bash
$ npm i -g vercel
$ vercel init vite
Vercel CLI
> Success! Initialized "vite" example in ~/your-folder.
- To deploy, `cd vite` and run `vercel`.
```

### Vercel for Git

1. Push your code to your git repository (GitHub, GitLab, Bitbucket).
2. [Import your Vite project](https://vercel.com/new) into Vercel.
3. Vercel will detect that you are using Vite and will enable the correct settings for your deployment.
4. Your application is deployed! (e.g. vite-vue-template.vercel.app)

After your project has been imported and deployed, all subsequent pushes to branches will generate [Preview Deployments](https://vercel.com/docs/concepts/deployments/environments#preview), and all changes made to the Production Branch (commonly â€œmainâ€) will result in a [Production Deployment](https://vercel.com/docs/concepts/deployments/environments#production).

Learn more about Vercelâ€™s [Git Integration](https://vercel.com/docs/concepts/git).

## Cloudflare Pages

### Cloudflare Pages via Wrangler

1. Install [Wrangler CLI](https://developers.cloudflare.com/workers/wrangler/get-started/).
2. Authenticate Wrangler with your Cloudflare account using `wrangler login`.
3. Run your build command.
4. Deploy using `npx wrangler pages deploy dist`.

```bash
# Install Wrangler CLI
$ npm install -g wrangler

# Login to Cloudflare account from CLI
$ wrangler login
```

~~~~markdown
# Run your build command
$ npm run build

# Create new deployment
$ npx wrangler pages deploy dist
```

After your assets are uploaded, Wrangler will give you a preview URL to inspect your site. When you log into the Cloudflare
Pages dashboard, you will see your new project.

### Cloudflare Pages with Git

1. Push your code to your git repository (GitHub, GitLab).
2. Log in to the Cloudflare dashboard and select your account in **Account Home** > **Pages**.
3. Select **Create a new Project** and the **Connect Git** option.
4. Select the git project you want to deploy and click **Begin setup**
5. Select the corresponding framework preset in the build setting depending on the Vite framework you have selected.
6. Then save and deploy!
7. Your application is deployed! (e.g `https://<PROJECTNAME>.pages.dev/`)

After your project has been imported and deployed, all subsequent pushes to branches will generate [Preview Deployments]
(https://developers.cloudflare.com/pages/platform/preview-deployments/) unless specified not to in your [branch build
controls](https://developers.cloudflare.com/pages/platform/branch-build-controls/). All changes to the Production Branch
(commonly â€œmainâ€) will result in a Production Deployment.

You can also add custom domains and handle custom build settings on Pages. Learn more about [Cloudflare Pages Git
Integration](https://developers.cloudflare.com/pages/get-started/#manage-your-site).

## Google Firebase

1. Make sure you have [firebase-tools](https://www.npmjs.com/package/firebase-tools) installed.

2. Create `firebase.json` and `.firebaserc` at the root of your project with the following content:

   `firebase.json`:

   ```json
   {
     "hosting": {
       "public": "dist",
       "ignore": [],
       "rewrites": [
         {
           "source": "**",
           "destination": "/index.html"
         }
       ]
     }
   }
   ```

   `.firebaserc`:

   ```js
   {
     "projects": {
       "default": "<YOUR_FIREBASE_ID>"
     }
   }
   ```

3. After running `npm run build`, deploy using the command `firebase deploy`.

## Surge

1. First install [surge](https://www.npmjs.com/package/surge), if you havenâ€™t already.

2. Run `npm run build`.

3. Deploy to surge by typing `surge dist`.

You can also deploy to a [custom domain](http://surge.sh/help/adding-a-custom-domain) by adding `surge dist yourdomain.com`.

## Azure Static Web Apps

You can quickly deploy your Vite app with Microsoft Azure [Static Web Apps](https://aka.ms/staticwebapps) service. You need:

- An Azure account and a subscription key. You can create a [free Azure account here](https://azure.microsoft.com/free).
- Your app code pushed to [GitHub](https://github.com).
- The [SWA Extension](https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurestaticwebapps) in [Visual
Studio Code](https://code.visualstudio.com).

Install the extension in VS Code and navigate to your app root. Open the Static Web Apps extension, sign in to Azure, and
~~~~

click the '+' sign to create a new Static Web App. You will be prompted to designate which subscription key to use.

Follow the wizard started by the extension to give your app a name, choose a framework preset, and designate the app root (usually `/`) and built file location `/dist`. The wizard will run and will create a GitHub action in your repo in a `.github` folder.

The action will work to deploy your app (watch its progress in your repo's Actions tab) and, when successfully completed, you can view your app in the address provided in the extension's progress window by clicking the 'Browse Website' button that appears when the GitHub action has run.

## Render

You can deploy your Vite app as a Static Site on [Render](https://render.com/).

1. Create a [Render account](https://dashboard.render.com/register).

2. In the [Dashboard](https://dashboard.render.com/), click the **New** button and select **Static Site**.

3. Connect your GitHub/GitLab account or use a public repository.

4. Specify a project name and branch.

   - **Build Command**: `npm run build`
   - **Publish Directory**: `dist`

5. Click **Create Static Site**.

   Your app should be deployed at `https://<PROJECTNAME>.onrender.com/`.

By default, any new commit pushed to the specified branch will automatically trigger a new deployment. [Auto-Deploy]
(https://render.com/docs/deploys#toggling-auto-deploy-for-a-service) can be configured in the project settings.

You can also add a [custom domain](https://render.com/docs/custom-domains) to your project.

```
<!--
  NOTE: The sections below are reserved for more deployment platforms not listed above.
  Feel free to submit a PR that adds a new section with a link to your platform's
  deployment guide, as long as it meets these criteria:

  1. Users should be able to deploy their site for free.
  2. Free tier offerings should host the site indefinitely and are not time-bound.
     Offering a limited number of computation resource or site counts in exchange is fine.
  3. The linked guides should not contain any malicious content.

  The Vite team may change the criteria and audit the current list from time to time.
  If a section is removed, we will ping the original PR authors before doing so.
-->
```

## Flightcontrol

Deploy your static site using [Flightcontrol](https://www.flightcontrol.dev/?ref=docs-vite) by following these [instructions]
(https://www.flightcontrol.dev/docs/reference/examples/vite?ref=docs-vite).

## Kinsta Static Site Hosting

Deploy your static site using [Kinsta](https://kinsta.com/static-site-hosting/) by following these [instructions]
(https://kinsta.com/docs/react-vite-example/).

## xmit Static Site Hosting

Deploy your static site using [xmit](https://xmit.co) by following this [guide](https://xmit.dev/posts/vite-quickstart/).

# Troubleshooting

See [Rollup's troubleshooting guide](https://rollupjs.org/troubleshooting/) for more information too.

If the suggestions here don't work, please try posting questions on [GitHub Discussions](https://github.com/vitejs/vite/discussions) or in the `#help` channel of [Vite Land Discord](https://chat.vitejs.dev).

## CJS

### Vite CJS Node API deprecated

The CJS build of Vite's Node API is deprecated and will be removed in Vite 6. See the [GitHub discussion](https://github.com/vitejs/vite/discussions/13928) for more context. You should update your files or frameworks to import the ESM build of Vite instead.

In a basic Vite project, make sure:

1. The `vite.config.js` file content is using the ESM syntax.
2. The closest `package.json` file has `"type": "module"`, or use the `.mjs`/`.mts` extension, e.g. `vite.config.mjs` or `vite.config.mts`.

For other projects, there are a few general approaches:

- **Configure ESM as default, opt-in to CJS if needed:** Add `"type": "module"` in the project `package.json`. All `*.js` files are now interpreted as ESM and need to use the ESM syntax. You can rename a file with the `.cjs` extension to keep using CJS instead.
- **Keep CJS as default, opt-in to ESM if needed:** If the project `package.json` does not have `"type": "module"`, all `*.js` files are interpreted as CJS. You can rename a file with the `.mjs` extension to use ESM instead.
- **Dynamically import Vite:** If you need to keep using CJS, you can dynamically import Vite using `import('vite')` instead. This requires your code to be written in an `async` context, but should still be manageable as Vite's API is mostly asynchronous.

If you're unsure where the warning is coming from, you can run your script with the `VITE_CJS_TRACE=true` flag to log the stack trace:

```bash
VITE_CJS_TRACE=true vite dev
```

If you'd like to temporarily ignore the warning, you can run your script with the `VITE_CJS_IGNORE_WARNING=true` flag:

```bash
VITE_CJS_IGNORE_WARNING=true vite dev
```

Note that postcss config files do not support ESM + TypeScript (`.mts` or `.ts` in `"type": "module"`) yet. If you have postcss configs with `.ts` and added `"type": "module"` to package.json, you'll also need to rename the postcss config to use `.cts`.

## CLI

### `Error: Cannot find module 'C:\foo\bar&baz\vite\bin\vite.js'`

The path to your project folder may include `&`, which doesn't work with `npm` on Windows ([npm/cmd-shim#45](https://github.com/npm/cmd-shim/issues/45)).

You will need to either:

- Switch to another package manager (e.g. `pnpm`, `yarn`)
- Remove `&` from the path to your project

## Config

### This package is ESM only

When importing a ESM only package by `require`, the following error happens.

> Failed to resolve "foo". This package is ESM only but it was tried to load by `require`.

> "foo" resolved to an ESM file. ESM file cannot be loaded by `require`.

ESM files cannot be loaded by [`require`](<https://nodejs.org/docs/latest-v18.x/api/esm.html#require:~:text=Using%20require%20to%20load%20an%20ES%20module%20is%20not%20supported%20because%20ES%20modules%20have%20asynchronous%20execution.%20Instead%2C%20use%20import()%20to%20load%20an%20ES%20module%20from%20a%20CommonJS%20module.>).

We recommend converting your config to ESM by either:

- adding `"type": "module"` to the nearest `package.json`
- renaming `vite.config.js`/`vite.config.ts` to `vite.config.mjs`/`vite.config.mts`

## Dev Server

### Requests are stalled forever

If you are using Linux, file descriptor limits and inotify limits may be causing the issue. As Vite does not bundle most of the files, browsers may request many files which require many file descriptors, going over the limit.

To solve this:

- Increase file descriptor limit by `ulimit`

  ```shell
  # Check current limit
  $ ulimit -Sn
  # Change limit (temporary)
  $ ulimit -Sn 10000 # You might need to change the hard limit too
  # Restart your browser
  ```

- Increase the following inotify related limits by `sysctl`

  ```shell
  # Check current limits
  $ sysctl fs.inotify
  # Change limits (temporary)
  $ sudo sysctl fs.inotify.max_queued_events=16384
  $ sudo sysctl fs.inotify.max_user_instances=8192
  $ sudo sysctl fs.inotify.max_user_watches=524288
  ```

If the above steps don't work, you can try adding `DefaultLimitNOFILE=65536` as an un-commented config to the following files:

- /etc/systemd/system.conf
- /etc/systemd/user.conf

For Ubuntu Linux, you may need to add the line `* - nofile 65536` to the file `/etc/security/limits.conf` instead of updating systemd config files.

Note that these settings persist but a **restart is required**.

### Network requests stop loading

When using a self-signed SSL certificate, Chrome ignores all caching directives and reloads the content. Vite relies on these caching directives.

To resolve the problem use a trusted SSL cert.

See: [Cache problems](https://helpx.adobe.com/mt/experience-manager/kb/cache-problems-on-chrome-with-SSL-certificate-errors.html), [Chrome issue](https://bugs.chromium.org/p/chromium/issues/detail?id=110649#c8)

#### macOS

You can install a trusted cert via the CLI with this command:

```
security add-trusted-cert -d -r trustRoot -k ~/Library/Keychains/login.keychain-db your-cert.cer
```

Or, by importing it into the Keychain Access app and updating the trust of your cert to "Always Trust."

### 431 Request Header Fields Too Large

When the server / WebSocket server receives a large HTTP header, the request will be dropped and the following warning will be shown.

> Server responded with status code 431. See https://vitejs.dev/guide/troubleshooting.html#_431-request-header-fields-too-large.

This is because Node.js limits request header size to mitigate [CVE-2018-12121](https://www.cve.org/CVERecord?id=CVE-2018-12121).

To avoid this, try to reduce your request header size. For example, if the cookie is long, delete it. Or you can use [`--max-http-header-size`](https://nodejs.org/api/cli.html#--max-http-header-sizesize) to change max header size.

## HMR

### Vite detects a file change but the HMR is not working

You may be importing a file with a different case. For example, `src/foo.js` exists and `src/bar.js` contains:

```js
import './Foo.js' // should be './foo.js'
```

Related issue: [#964](https://github.com/vitejs/vite/issues/964)

### Vite does not detect a file change

If you are running Vite with WSL2, Vite cannot watch file changes in some conditions. See [`server.watch` option](/config/server-options.md#server-watch).

### A full reload happens instead of HMR

If HMR is not handled by Vite or a plugin, a full reload will happen as it's the only way to refresh the state.

If HMR is handled but it is within a circular dependency, a full reload will also happen to recover the execution order. To solve this, try breaking the loop. You can run `vite --debug hmr` to log the circular dependency path if a file change triggered it.

## Build

### Built file does not work because of CORS error

If the HTML file output was opened with `file` protocol, the scripts won't run with the following error.

> Access to script at 'file:///foo/bar.js' from origin 'null' has been blocked by CORS policy: Cross origin requests are only supported for protocol schemes: http, data, isolated-app, chrome-extension, chrome, https, chrome-untrusted.

> Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at file:///foo/bar.js. (Reason: CORS request not http).

See [Reason: CORS request not HTTP - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSRequestNotHttp) for more information about why this happens.

You will need to access the file with `http` protocol. The easiest way to achieve this is to run `npx vite preview`.

## Optimized Dependencies

### Outdated pre-bundled deps when linking to a local package

The hash key used to invalidate optimized dependencies depends on the package lock contents, the patches applied to dependencies, and the options in the Vite config file that affects the bundling of node modules. This means that Vite will detect when a dependency is overridden using a feature as [npm overrides](https://docs.npmjs.com/cli/v9/configuring-npm/package-json#overrides), and re-bundle your dependencies on the next server start. Vite won't invalidate the dependencies when you use a feature like [npm link](https://docs.npmjs.com/cli/v9/commands/npm-link). In case you link or unlink a dependency, you'll need to force re-optimization on the next server start by using `vite --force`. We recommend using overrides instead, which are supported now by every package manager (see also [pnpm overrides](https://pnpm.io/package_json#pnpmoverrides) and [yarn resolutions](https://yarnpkg.com/configuration/manifest/#resolutions)).

## Performance bottlenecks

If you suffer any application performance bottlenecks resulting in slow load times, you can start the built-in Node.js inspector with your Vite dev server or when building your application to create the CPU profile:

::: code-group

```bash [dev server]
vite --profile --open
```

```bash [build]
vite build --profile
```

:::

::: tip Vite Dev Server
Once your application is opened in the browser, just await finish loading it and then go back to the terminal and press `p` key (will stop the Node.js inspector) then press `q` key to stop the dev server.
:::

Node.js inspector will generate `vite-profile-0.cpuprofile` in the root folder, go to https://www.speedscope.app/, and upload the CPU profile using the `BROWSE` button to inspect the result.

You can install [vite-plugin-inspect](https://github.com/antfu/vite-plugin-inspect), which lets you inspect the intermediate state of Vite plugins and can also help you to identify which plugins or middlewares are the bottleneck in your applications. The plugin can be used in both dev and build modes. Check the readme file for more details.

## Others

### Module externalized for browser compatibility

When you use a Node.js module in the browser, Vite will output the following warning.

> Module "fs" has been externalized for browser compatibility. Cannot access "fs.readFile" in client code.

This is because Vite does not automatically polyfill Node.js modules.

We recommend avoiding Node.js modules for browser code to reduce the bundle size, although you can add polyfills manually. If the module is imported from a third-party library (that's meant to be used in the browser), it's advised to report the issue to the respective library.

### Syntax Error / Type Error happens

Vite cannot handle and does not support code that only runs on non-strict mode (sloppy mode). This is because Vite uses ESM and it is always [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) inside ESM.

For example, you might see these errors.

> [ERROR] With statements cannot be used with the "esm" output format due to strict mode

> TypeError: Cannot create property 'foo' on boolean 'false'

If these codes are used inside dependencies, you could use [`patch-package`](https://github.com/ds300/patch-package) (or [`yarn patch`](https://yarnpkg.com/cli/patch) or [`pnpm patch`](https://pnpm.io/cli/patch)) for an escape hatch.

### Browser extensions

Some browser extensions (like ad-blockers) may prevent the Vite client from sending requests to the Vite dev server. You may see a white screen without logged errors in this case. Try disabling extensions if you have this issue.

### Cross drive links on Windows

If there's a cross drive links in your project on Windows, Vite may not work.

An example of cross drive links are:

- a virtual drive linked to a folder by `subst` command
- a symlink/junction to a different drive by `mklink` command (e.g. Yarn global cache)

Related issue: [#10802](https://github.com/vitejs/vite/issues/10802)

# Using Plugins

Vite can be extended using plugins, which are based on Rollup's well-designed plugin interface with a few extra Vite-specific options. This means that Vite users can rely on the mature ecosystem of Rollup plugins, while also being able to extend the dev server and SSR functionality as needed.

## Adding a Plugin

To use a plugin, it needs to be added to the `devDependencies` of the project and included in the `plugins` array in the `vite.config.js` config file. For example, to provide support for legacy browsers, the official [@vitejs/plugin-legacy](https://github.com/vitejs/vite/tree/main/packages/plugin-legacy) can be used:

```
$ npm add -D @vitejs/plugin-legacy
```

```js twoslash
// vite.config.js
import legacy from '@vitejs/plugin-legacy'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [
    legacy({
      targets: ['defaults', 'not IE 11'],
    }),
  ],
})
```

`plugins` also accepts presets including several plugins as a single element. This is useful for complex features (like framework integration) that are implemented using several plugins. The array will be flattened internally.

Falsy plugins will be ignored, which can be used to easily activate or deactivate plugins.

## Finding Plugins

:::tip NOTE
Vite aims to provide out-of-the-box support for common web development patterns. Before searching for a Vite or compatible Rollup plugin, check out the [Features Guide](../guide/features.md). A lot of the cases where a plugin would be needed in a Rollup project are already covered in Vite.
:::

Check out the [Plugins section](../plugins/) for information about official plugins. Community plugins are listed in [awesome-vite](https://github.com/vitejs/awesome-vite#plugins).

You can also find plugins that follow the [recommended conventions](./api-plugin.md#conventions) using a [npm search for vite-plugin](https://www.npmjs.com/search?q=vite-plugin&ranking=popularity) for Vite plugins or a [npm search for rollup-plugin](https://www.npmjs.com/search?q=rollup-plugin&ranking=popularity) for Rollup plugins.

## Enforcing Plugin Ordering

For compatibility with some Rollup plugins, it may be needed to enforce the order of the plugin or only apply at build time. This should be an implementation detail for Vite plugins. You can enforce the position of a plugin with the `enforce` modifier:

- `pre`: invoke plugin before Vite core plugins
- default: invoke plugin after Vite core plugins
- `post`: invoke plugin after Vite build plugins

```js twoslash
// vite.config.js
import image from '@rollup/plugin-image'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [
    {
      ...image(),
      enforce: 'pre',
    },
  ],
})
```

Check out [Plugins API Guide](./api-plugin.md#plugin-ordering) for detailed information.

## Conditional Application

By default, plugins are invoked for both serve and build. In cases where a plugin needs to be conditionally applied only during serve or build, use the `apply` property to only invoke them during `'build'` or `'serve'`:

```js twoslash
```

```js
// vite.config.js
import typescript2 from 'rollup-plugin-typescript2'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [
    {
      ...typescript2(),
      apply: 'build',
    },
  ],
})
```

## Building Plugins

Check out the [Plugins API Guide](./api-plugin.md) for documentation about creating plugins.

# Why Vite

## The Problems

Before ES modules were available in browsers, developers had no native mechanism for authoring JavaScript in a modularized fashion. This is why we are all familiar with the concept of "bundling": using tools that crawl, process and concatenate our source modules into files that can run in the browser.

Over time we have seen tools like [webpack](https://webpack.js.org/), [Rollup](https://rollupjs.org) and [Parcel](https://parceljs.org/), which greatly improved the development experience for frontend developers.

However, as we build more and more ambitious applications, the amount of JavaScript we are dealing with is also increasing dramatically. It is not uncommon for large scale projects to contain thousands of modules. We are starting to hit a performance bottleneck for JavaScript based tooling: it can often take an unreasonably long wait (sometimes up to minutes!) to spin up a dev server, and even with Hot Module Replacement (HMR), file edits can take a couple of seconds to be reflected in the browser. The slow feedback loop can greatly affect developers' productivity and happiness.

Vite aims to address these issues by leveraging new advancements in the ecosystem: the availability of native ES modules in the browser, and the rise of JavaScript tools written in compile-to-native languages.

### Slow Server Start

When cold-starting the dev server, a bundler-based build setup has to eagerly crawl and build your entire application before it can be served.

Vite improves the dev server start time by first dividing the modules in an application into two categories: **dependencies** and **source code**.

- **Dependencies** are mostly plain JavaScript that do not change often during development. Some large dependencies (e.g. component libraries with hundreds of modules) are also quite expensive to process. Dependencies may also be shipped in various module formats (e.g. ESM or CommonJS).

  Vite [pre-bundles dependencies](./dep-pre-bundling) using [esbuild](https://esbuild.github.io/). esbuild is written in Go and pre-bundles dependencies 10-100x faster than JavaScript-based bundlers.

- **Source code** often contains non-plain JavaScript that needs transforming (e.g. JSX, CSS or Vue/Svelte components), and will be edited very often. Also, not all source code needs to be loaded at the same time (e.g. with route-based code-splitting).

  Vite serves source code over [native ESM](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). This is essentially letting the browser take over part of the job of a bundler: Vite only needs to transform and serve source code on demand, as the browser requests it. Code behind conditional dynamic imports is only processed if actually used on the current screen.

```
<script setup>
import bundlerSvg from '../images/bundler.svg?raw'
import esmSvg from '../images/esm.svg?raw'
</script>
<svg-image :svg="bundlerSvg" />
<svg-image :svg="esmSvg" />
```

### Slow Updates

When a file is edited in a bundler-based build setup, it is inefficient to rebuild the whole bundle for an obvious reason: the update speed will degrade linearly with the size of the app.

In some bundlers, the dev server runs the bundling in memory so that it only needs to invalidate part of its module graph when a file changes, but it still needs to re-construct the entire bundle and reload the web page. Reconstructing the bundle can be expensive, and reloading the page blows away the current state of the application. This is why some bundlers support Hot Module Replacement (HMR): allowing a module to "hot replace" itself without affecting the rest of the page. This greatly improves DX - however, in practice we've found that even HMR update speed deteriorates significantly as the size of the application grows.

In Vite, HMR is performed over native ESM. When a file is edited, Vite only needs to precisely invalidate the chain between the edited module and its closest HMR boundary (most of the time only the module itself), making HMR updates consistently fast regardless of the size of your application.

Vite also leverages HTTP headers to speed up full page reloads (again, let the browser do more work for us): source code module requests are made conditional via `304 Not Modified`, and dependency module requests are strongly cached via `Cache-Control: max-age=31536000,immutable` so they don't hit the server again once cached.

Once you experience how fast Vite is, we highly doubt you'd be willing to put up with bundled development again.

## Why Bundle for Production

Even though native ESM is now widely supported, shipping unbundled ESM in production is still inefficient (even with HTTP/2) due to the additional network round trips caused by nested imports. To get the optimal loading performance in production, it is still better to bundle your code with tree-shaking, lazy-loading and common chunk splitting (for better caching).

Ensuring optimal output and behavioral consistency between the dev server and the production build isn't easy. This is why Vite ships with a pre-configured [build command](./build) that bakes in many [performance optimizations](./features#build-optimizations) out of the box.

## Why Not Bundle with esbuild?

Vite's current plugin API isn't compatible with using `esbuild` as a bundler. In spite of `esbuild` being faster, Vite's adoption of Rollup's flexible plugin API and infrastructure heavily contributed to its success in the ecosystem. For the time being, we believe that Rollup offers a better performance-vs-flexibility tradeoff.

Rollup has also been working on performance improvements, [switching its parser to SWC in v4](https://github.com/rollup/rollup/pull/5073). And there is an ongoing effort to build a Rust-port of Rollup called Rolldown. Once Rolldown is ready, it could replace both Rollup and esbuild in Vite, improving build performance significantly and removing inconsistencies between development and build. You can watch [Evan You's ViteConf 2023 keynote for more details](https://youtu.be/hrdwQHoAp0M).

## How is Vite Different from X?

You can check out the [Comparisons](./comparisons) section for more details on how Vite differs from other similar tools.