VITE Docs - English

Table of contents

HMR API

Note

This is the client HMR API. For handling HMR update in plugins, see 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:

```
interface ImportMeta {
  readonly hot?: ViteHotContext
}

type ModuleNamespace = Record<string, any> & {
  [Symbol.toStringTag]: 'Module'
}

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
  decline(): void
  invalidate(): void

  // `InferCustomEventPayload` provides types for built-in Vite events
  on<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:

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

`hot.accept(cb)`

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

```
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)
    }
  })
}
```

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

Note that 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.

`hot.accept(deps, cb)`

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

```
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 the updated modules in an Array
    }
  )
}
```

`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:

```
function setupSideEffect() {}

setupSideEffect()

if (import.meta.hot) {
  import.meta.hot.dispose((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.

`hot.decline()`

Calling `import.meta.hot.decline()` indicates this module is not hot-updatable, and the browser should perform a full reload if this module is encountered while propagating HMR updates.

`hot.invalidate()`

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.

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:

```
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)

Custom HMR events can also be sent from plugins. See handleHotUpdate for more details.

`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 for more details.

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:

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

Example Usage:

```
import { fileURLToPath } from 'url'
import { createServer } from 'vite'

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

;(async () => {
  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()
})()
```

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.

`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`

```
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.
   * 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): 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.
   */
  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>
}
```

`build`

Type Signature:

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

Example Usage:

```
import path from 'path'
import { fileURLToPath } from 'url'
import { build } from 'vite'

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

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

`preview`

Type Signature:

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

Example Usage:

```js
import { preview } from 'vite'
;(async () => {
  const previewServer = await preview({
    // any valid user config options, plus `mode` and `configFile`
    preview: {
      port: 8080,
      open: true
    }
  })

  previewServer.printUrls()
})()
```

`resolveConfig`

Type Signature:

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

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

`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.

`searchForWorkspaceRoot`

Type Signature:

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

Related: server.fs.allow

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

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

`loadEnv`

Type Signature:

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

Related: `.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:

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

Related: Path Normalization

Normalizes a path to interoperate between Vite plugins.

`transformWithEsbuild`

Type Signature:

```
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:

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

Load a Vite config file manually with esbuild.

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 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 to see if your need is covered. Also review available community plugins, both in the form of a compatible Rollup plugin and Vite Specific 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.

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, then it is recommended to use the Rollup Plugin naming 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.

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

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

Simple Examples

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

```
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:

```
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 the same way Rollup does it.

The following hooks are called once on server start:

- `options`
- `buildStart`

The following hooks are called on each incoming module request:

- `resolveId`
- `load`
- `transform`

The following hooks are called when the server is closed:

- `buildEnd`
- `closeBundle`

Note that the `moduleParsed` hook is **not** called during dev, because Vite avoids full AST parses for better performance.

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:

```
// 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:

```
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

  Hook for configuring the dev server. The most common use case is adding custom middlewares to the internal connect app:

```
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:

```
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:

```
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: { middlewares: Connect.Server, httpServer: http.Server }) => (() => void) | void | Promise<(() => void) | void>`
- **Kind:** `async`, `sequential`

  Same as `configureServer` but for the preview server. It provides the connect server and its underlying http 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:

```
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 | { enforce?: 'pre' | 'post', transform: 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` 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 }`

Basic Example:

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

Full Hook Signature:

```
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'
}
```

`handleHotUpdate`

- **Type:** `(ctx: HmrContext) => Array<ModuleNode> | void | Promise<Array<ModuleNode> | void>`

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

  ```
  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 complete custom HMR handling by sending custom events to the client:

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

    Client code should register corresponding handler using the HMR API (this could be injected by the same plugin's `transform` hook):

    ```
    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)

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'`:

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

A function can also be used for more precise control:

```
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` 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:

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

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

Check out Vite Rollup Plugins for a list of compatible official Rollup plugins with usage instructions.

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 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.

```
import { normalizePath } from 'vite'

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

Filtering, include/exclude pattern

Vite exposes `@rollup/pluginutils`'s `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 all the clients:

```
// vite.config.js
export default defineConfig({
  plugins: [
    {
      // ...
      configureServer(server) {
        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` to listen to the events:

```
// 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`:

```
// 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:

```
// 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

It is possible to type custom events by extending the `CustomEventMap` interface:

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

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

Static Asset Handling

- Related: Public Base Path
- Related: `assetsInclude` config option

Importing Asset as URL

Importing a static asset will return the resolved public URL when it is served:

```
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.
- 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 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.

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.

```
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.

```
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
// Separate chunk in the production build
import Worker from './shader.js?worker'
const worker = new Worker()
```

```js
// sharedworker
import SharedWorker from './shader.js?sharedworker'
const sharedWorker = new SharedWorker()
```

```js
// Inlined as base64 strings
import InlineWorker from './shader.js?worker&inline'
```

Check out the Web Worker section 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.

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 is a native ESM feature that exposes the current module's URL. Combining it with the native URL constructor, 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:

```
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`

```
// 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.

Backend Integration

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.

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:

```
// vite.config.js
export default defineConfig({
  build: {
    // generate 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, you also need to import the polyfill in your entry

```
// 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):

```
<!-- 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` 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:

```
<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 `manifest.json` file will be generated alongside other asset files. An example manifest file looks like this:

```json
{
  "main.js": {
    "file": "assets/main.4889e940.js",
    "src": "main.js",
    "isEntry": true,
    "dynamicImports": ["views/foo.js"],
    "css": ["assets/main.b82dbe22.css"],
    "assets": ["assets/asset.0ab0f9cd.png"]
  },
  "views/foo.js": {
    "file": "assets/foo.869aea0d.js",
    "src": "views/foo.js",
    "isDynamicEntry": true,
    "imports": ["_shared.83069a53.js"]
  },
  "_shared.83069a53.js": {
    "file": "assets/shared.83069a53.js"
  }
}
```

   - 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).

   You can use this file to render links or preload directives with hashed filenames (note: the syntax here is for explanation only, substitute with your server templating language):

```html
<!-- if production -->
<link rel="stylesheet" href="/assets/{{ manifest['main.js'].css }}" />
<script type="module" src="/assets/{{ manifest['main.js'].file }}"></script>
```

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 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, native ESM dynamic import, and `import.meta`:

- Chrome >=87
- Firefox >=78
- Safari >=13
- Edge >=88

You can specify custom targets via the `build.target` config option, where the lowest target is `es2015`.

Note that by default, Vite only handles syntax transforms and **does not cover polyfills by default**. You can check out Polyfill.io which is a service that automatically generates polyfill bundles based on the user's browser UserAgent string.

Legacy browsers can be supported via @vitejs/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

If you are deploying your project under a nested public path, simply specify the `base` config option 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.

Customizing the Build

The build can be customized via various build config options. Specifically, you can directly adjust the underlying Rollup options via `build.rollupOptions`:

```
// vite.config.js
export default defineConfig({
  build: {
    rollupOptions: {
      // https://rollupjs.org/guide/en/#big-list-of-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). Until Vite 2.8, the default chunking strategy divided the chunks into `index` and `vendor`. It is a good strategy for some SPAs, but it is hard to provide a general solution for every Vite target use case. From Vite 2.9, `manualChunks` is no longer modified by default. You can continue to use the Split Vendor Chunk strategy by adding the `splitVendorChunkPlugin` in your config file:

```
// vite.config.js
import { splitVendorChunkPlugin } from 'vite'
export default defineConfig({
  plugins: [splitVendorChunkPlugin()]
})
```

This strategy is also provided as a `splitVendorChunk({ cache: SplitVendorChunkCache })` factory, in case composition with custom logic is needed. `cache.reset()` needs to be called at `buildStart` for build watch mode to work correctly in this case.

Rebuild on files changes

You can enable rollup watcher with `vite build --watch`. Or, you can directly adjust the underlying `WatcherOptions` via `build.watch`:

```
// vite.config.js
export default defineConfig({
  build: {
    watch: {
      // https://rollupjs.org/guide/en/#watch-options
    }
  }
})
```

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:

```
├── package.json
├── vite.config.js
├── index.html
├── main.js
└── nested
    ├── index.html
    └── 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:

```
// 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`.

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. Make sure to also externalize any dependencies that you do not want to bundle into your library, e.g. `vue` or `react`:

```
// 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:

```
// 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 KiB / gzip: 0.07 KiB
dist/my-lib.umd.cjs 0.30 KiB / gzip: 0.16 KiB
```

Recommended `package.json` for your lib:

```
{
  "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:

```
{
  "name": "my-lib",
  "type": "module",
  "files": ["dist"],
  "main": "./dist/my-lib.cjs",
  "module": "./dist/my-lib.mjs",
  "exports": {
    ".": {
      "import": "./dist/my-lib.mjs",
      "require": "./dist/my-lib.cjs"
    },
    "./secondary": {
      "import": "./dist/secondary.mjs",
      "require": "./dist/secondary.cjs"
    }
  }
}
```

tip Note

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.*` 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.

Advanced Base Options

warning This feature is experimental, the API may change in a future minor without following semver. Please always pin Vite's version to a minor when using it.

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

A single static base isn't enough in these scenarios. Vite provides experimental support for advanced base options during build, using `experimental.renderBuiltUrl`.

```
experimental: {
  renderBuiltUrl(filename: string, { hostType }: { hostType: 'js' | 'css' | 'html' }) {
    if (hostType === 'js') {
      return { runtime: `window.__toCdnUrl(${JSON.stringify(filename)})` }
    } else {
      return { relative: true }
    }
  }
}
```

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.

```
experimental: {
  renderBuiltUrl(filename: string, { hostId, hostType, type }: { hostId: string, hostType: 'j
    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
    }
  }
}
```

Comparisons

WMR

WMR by the Preact team provides a similar feature set, and Vite 2.0's support for Rollup's plugin interface is inspired by it.

WMR is mainly designed for Preact projects, and offers more integrated features such as pre-rendering. In terms of scope, it's closer to a Preact meta framework, with the same emphasis on compact size as Preact itself. If you are using Preact, WMR is likely going to offer a more fine-tuned experience.

@web/dev-server

@web/dev-server (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 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, 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`). Some of the main differences between the two projects are listed in the v2 Comparisons Guide.

Dependency Pre-Bundling

When you run `vite` for the first time, you may notice this message:

```
Pre-bundling dependencies:
  react
  react-dom
(this will be run only when your dependencies or config have changed)
```

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):

   ```
   // 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! 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.

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` and `build.commonjsOptions.include` in your config.

```
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.

warning Deduping Due to differences in linked dependency resolution, transitive dependencies can deduplicate incorrectly, causing issues when used in runtime. If you stumble on this issue, use `npm pack` on the linked dependency to fix it.

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.

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.

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:

- The `dependencies` list in your `package.json`.
- Package manager lockfiles, e.g. `package-lock.json`, `yarn.lock`, or `pnpm-lock.yaml`.
- Relevant fields in your `vite.config.js`, if present.

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. Some built-in variables are available in all cases:

- `import.meta.env.MODE`: {string} the mode 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.
- `import.meta.env.PROD`: {boolean} whether the app is running in 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.

Production Replacement

During production, these env variables are **statically replaced**. It is therefore necessary to always reference them using the full static string. For example, dynamic key access like `import.meta.env[key]` will not work.

It will also replace these strings appearing in JavaScript strings and Vue templates. This should be a rare case, but it can be unintended. You may see errors like `Missing Semicolon` or `Unexpected token` in this case, for example when `"process.env. NODE_ENV"` is transformed to `""development": "`. There are ways to work around this behavior:

- For JavaScript strings, you can break the string up with a Unicode zero-width space, e.g. `'import.meta\u200b.env.MODE'`.
- For Vue templates or other HTML that gets compiled into JavaScript strings, you can use the `<wbr>` tag, e.g. `import.meta.<wbr>env.MODE`.

`.env` Files

Vite uses dotenv to load additional environment variables from the following files in your environment directory:

```
.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
```

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.

```
console.log(import.meta.env.VITE_SOME_KEY) // 123
console.log(import.meta.env.DB_PASSWORD) // undefined
```

Also, Vite uses dotenv-expand to expand variables out of the box. To learn more about the syntax, check out their docs.

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 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`. 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 `env.d.ts` in `src` directory, then augment `ImportMetaEnv` like this:

```
/// <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 and WebWorker, you can update the lib field in `tsconfig.json`.

```
{
  "lib": ["WebWorker"]
}
```

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`.

However, it is important to understand that **mode** is a wider concept than just development vs. production. A typical example is you may want to have a "staging" mode where it should have production-like behavior, but with slightly different env variables from production.

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 our hypothetical staging mode:

```
vite build --mode staging
```

And to get the behavior we want, we need a `.env.staging` file:

```
# .env.staging
NODE_ENV=production
VITE_APP_TITLE=My App (staging)
```

Now your staging app should have production-like behavior, but display a different title from production.

Features

At the very basic level, developing using Vite is not that much 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:

```
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 them to improve page loading speed and convert CommonJS / UMD modules to ESM. The pre-bundling step is performed with esbuild 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.

Hot Module Replacement

Vite provides an HMR API 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 and React Fast Refresh. There are also official integrations for Preact via @prefresh/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.

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 (you can run `tsc --noEmit` in the build script or install `vue-tsc` and run `vue-tsc --noEmit` to also type check your `*.vue` files).

Vite uses 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 syntax to avoid potential problems like type-only imports being incorrectly bundled, for example:

```
import type { T } from 'only/types'
export type { T }
```

TypeScript Compiler Options

Some configuration fields under `compilerOptions` in `tsconfig.json` require special attention.

`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.

However, some libraries (e.g. `vue`) don't work well with `"isolatedModules": true`. You can use `"skipLibCheck": true` to temporarily suppress the errors until it is fixed upstream.

`useDefineForClassFields`

Starting from Vite 2.5.0, the default value will be `true` if the TypeScript target is `ESNext`. It is consistent with the behavior of `tsc` 4.3.2 and later. It is also the standard ECMAScript runtime behavior.

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.

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, Vue Class Components 8.x, etc.

But a few libraries haven't transitioned to this new default yet, including `lit-element`. Please explicitly set `useDefineForClassFields` to `false` in these cases.

Other Compiler Options Affecting the Build Result

- `extends`
- `importsNotUsedAsValues`
- `preserveValueImports`
- `jsxFactory`
- `jsxFragmentFactory`

If migrating your codebase to `"isolatedModules": true` is an unsurmountable effort, you may be able to get around it with a third-party plugin such as rollup-plugin-friendly-type-imports. However, this approach is not officially supported by Vite.

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:

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

Also, you can add `vite/client` to `compilerOptions.types` of your `tsconfig`:

```
{
  "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 on `import.meta.env`
- Types for the HMR API on `import.meta.hot`

tip To override the default typing, declare it before the triple-slash reference. For example, to make the default import of `*.svg` a React component:

```
declare module '*.svg' {
  const content: React.FC<React.SVGProps<SVGElement>>
  export default content
}

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

Vue

Vite provides first-class Vue support:

- Vue 3 SFC support via @vitejs/plugin-vue
- Vue 3 JSX support via @vitejs/plugin-vue-jsx
- Vue 2.7 support via @vitejs/plugin-vue2
- Vue <2.7 support via vite-plugin-vue2

JSX

`.jsx` and `.tsx` files are also supported out of the box. JSX transpilation is also handled via esbuild.

Vue users should use the official @vitejs/plugin-vue-jsx plugin, which provides Vue 3 specific features including HMR, global component resolving, directives and slots.

If not using JSX with React or Vue, custom `jsxFactory` and `jsxFragment` can be configured using the `esbuild` option. For example for Preact:

```
// vite.config.js
import { defineConfig } from 'vite'

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

More details in esbuild docs.

You can inject the JSX helpers using `jsxInject` (which is a Vite-only option) to avoid manual imports:

```
// 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. You can also retrieve the processed CSS as a string as the module's default export.

`@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).

PostCSS

If the project contains valid PostCSS config (any format supported by 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` option.

CSS Modules

Any CSS file ending with `.module.css` is considered a CSS modules file. Importing such a file will return the corresponding module object:

```css
/* example.module.css */
.red {
  color: red;
}
```

```js
import classes from './example.module.css'
document.getElementById('foo').className = classes.red
```

CSS modules behavior can be configured via the `css.modules` option.

If `css.modules.localsConvention` is set to enable camelCase locals (e.g. `localsConvention: 'camelCaseOnly'`), you can also use named imports:

```js
// .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) 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

# .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.

```
import styles from './foo.css' // will be injected into the page
import otherStyles from './bar.css?inline' // will not be injected into the page
```

Static Assets

Importing a static asset will return the resolved public URL when it is served:

```
import imgUrl from './img.png'
document.getElementById('hero-img').src = imgUrl
```

Special queries can modify how assets are loaded:

```
// Explicitly load assets as URL
import assetAsURL from './asset.js?url'
```

```
// Load assets as strings
import assetAsString from './shader.glsl?raw'
```

```
// Load Web Workers
import Worker from './worker.js?worker'
```

```
// Web Workers inlined as base64 strings at build time
import InlineWorker from './worker.js?worker&inline'
```

More details in Static Asset Handling.

JSON

JSON files can be directly imported - named imports are also supported:

```
// 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:

```
const modules = import.meta.glob('./dir/*.js')
```

The above will be transformed into the following:

```
// 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:

```
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:

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

The above will be transformed into the following:

```
// 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
}
```

Glob Import As

`import.meta.glob` also supports importing files as strings (similar to Importing Asset as String) with the Import Reflection syntax:

```
const modules = import.meta.glob('./dir/*.js', { as: 'raw' })
```

The above will be transformed into the following:

```
// code produced by vite
const modules = {
  './dir/foo.js': 'export default "foo"\n',
  './dir/bar.js': 'export default "bar"\n'
}
```

`{ as: 'url' }` is also supported for loading assets as URLs.

Multiple Patterns

The first argument can be an array of globs, for example

```
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:

```
const modules = import.meta.glob(['./dir/*.js', '!**/bar.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.

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

```
// 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.

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

```
// 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.

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

```
// 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 custom queries to imports for other plugins to consume.

```
const modules = import.meta.glob('./dir/*.js', {
  query: { foo: 'bar', bar: true }
})
```

```
// code produced by vite:
const modules = {
  './dir/foo.js': () =>
    import('./dir/foo.js?foo=bar&bar=true').then((m) => m.setup),
  './dir/bar.js': () =>
    import('./dir/bar.js?foo=bar&bar=true').then((m) => m.setup)
}
```

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).
- The glob matching is done via `fast-glob` - check out its documentation for supported glob patterns.
- 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, Vite also supports dynamic import with variables.

```
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 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 wasm instance:

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

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

The init function can also take the `imports` object which is passed along to `WebAssembly.instantiate` as its second argument:

```
init({
  imports: {
    someFunc: () => {
      /* ... */
    }
  }
}).then(() => {
  /* ... */
})
```

In the production build, `.wasm` files smaller than `assetInlineLimit` will be inlined as base64 strings. Otherwise, they will be copied to the dist directory as an asset and fetched on-demand.

warning [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.

Web Workers

Import with Constructors

A web worker script can be imported using `new Worker()` and `new SharedWorker()`. Compared to the worker suffixes, this syntax leans closer to the standards and is the **recommended** way to create workers.

```
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:

```
const worker = new Worker(new URL('./worker.js', import.meta.url), {
  type: 'module'
})
```

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:

```
import MyWorker from './worker?worker'

const worker = new MyWorker()
```

The worker script can also use `import` statements instead of `importScripts()` - note during dev this relies on browser native support and currently only works in Chrome, 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:

```
import MyWorker from './worker?worker&inline'
```

If you wish to retrieve the worker as a URL, add the `url` query:

```
import MyWorker from './worker?worker&url'
```

See Worker Options for details on configuring the bundling of all workers.

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.

If you'd rather have all the CSS extracted into a single file, you can disable CSS code splitting by setting `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:

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.

Getting Started

Overview

Vite (French word for "quick", pronounced `/vit/`, 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 over native ES modules, for example extremely fast Hot Module Replacement (HMR).
- A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production.

Vite is opinionated and comes with sensible defaults out of the box, but is also highly extensible via its Plugin API and JavaScript API with full typing support.

You can learn more about the rationale behind the project in the Why Vite section.

Browser Support

The default build targets browsers that support native ES Modules, native ESM dynamic import, and `import.meta`. Legacy browsers can be supported via the official @vitejs/plugin-legacy - see the Building for Production section for more details.

Trying Vite Online

You can try Vite online on StackBlitz. 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	vanilla-ts
vue	vue-ts
react	react-ts
preact	preact-ts
lit	lit-ts
svelte	svelte-ts

Scaffolding Your First Vite Project

tip Compatibility Note

Vite requires Node.js version 14.18+, 16+. However, some templates require a higher Node.js version to work, please upgrade if your package manager warns about it.

With NPM:

```
$ npm create vite@latest
```

With Yarn:

```
$ yarn create vite
```

With PNPM:

```
$ pnpm create vite
```

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:

```
# npm 6.x
npm create vite@latest my-vue-app --template vue

# npm 7+, extra double-dash is needed:
npm create vite@latest my-vue-app -- --template vue

# yarn
yarn create vite my-vue-app --template vue

# pnpm
pnpm create vite my-vue-app --template vue
```

See create-vite for more details on each supported template: `vanilla`, `vanilla-ts`, `vue`, `vue-ts`, `react`, `react-ts`, `preact`, `preact-ts`, `lit`, `lit-ts`, `svelte`, `svelte-ts`.

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 that include other tools or target different frameworks. You can use a tool like degit to scaffold your project with one of the templates.

```
npx degit user/project my-project
cd my-project

npm install
npm run dev
```

If the project uses `main` as the default branch, suffix the project repo with `#main`

```
npx degit user/project#main my-project
```

`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 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`.

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:

```
{
  "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 `--https`. For a full list of CLI options, run `npx vite --help` in your project.

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 to your local machine and then build and link it yourself (pnpm is required):

```
git clone https://github.com/vitejs/vite.git
cd vite
pnpm install
cd packages/vite
pnpm run build
pnpm link --global # you can 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 and GitHub Discussions.

Migration from v2

Node.js Support

Vite no longer supports Node.js 12 / 13 / 15, which reached its EOL. Node.js 14.18+ / 16+ is now required.

Modern Browser Baseline change

The production bundle assumes support for modern JavaScript. By default, Vite targets browsers which support the native ES Modules, native ESM dynamic import, and `import.meta`:

- Chrome >=87
- Firefox >=78
- Safari >=13
- Edge >=88

A small fraction of users will now require using @vitejs/plugin-legacy, which will automatically generate legacy chunks and corresponding ES language feature polyfills.

Config Options Changes

The following options that were already deprecated in v2 have been removed:

- `alias` (switch to `resolve.alias`)
- `dedupe` (switch to `resolve.dedupe`)
- `build.base` (switch to `base`)
- `build.brotliSize` (switch to `build.reportCompressedSize`)
- `build.cleanCssOptions` (Vite now uses esbuild for CSS minification)
- `build.polyfillDynamicImport` (use `@vitejs/plugin-legacy` for browsers without dynamic import support)
- `optimizeDeps.keepNames` (switch to `optimizeDeps.esbuildOptions.keepNames`)

Architecture Changes and Legacy Options

This section describes the biggest architecture changes in Vite v3. To allow projects to migrate from v2 in case of a compat issue, legacy options have been added to revert to the Vite v2 strategies.

Dev Server Changes

Vite's default dev server port is now 5173. You can use `server.port` to set it to 3000.

Vite's default dev server host is now `localhost`. In Vite v2, Vite was listening to `127.0.0.1` by default. Node.js under v17 normally resolves `localhost` to `127.0.0.1`, so for those versions, the host won't change. For Node.js 17+, you can use `server.host` to set it to `127.0.0.1` to keep the same host as Vite v2.

Note that Vite v3 now prints the correct host. This means Vite may print `127.0.0.1` as the listening host when `localhost` is used. You can set `dns.setDefaultResultOrder('verbatim')` to prevent this. See `server.host` for more details.

SSR Changes

Vite v3 uses ESM for the SSR build by default. When using ESM, the SSR externalization heuristics are no longer needed. By default, all dependencies are externalized. You can use `ssr.noExternal` to control what dependencies to include in the SSR bundle.

If using ESM for SSR isn't possible in your project, you can set `legacy.buildSsrCjsExternalHeuristics` to generate a CJS bundle using the same externalization strategy of Vite v2.

Also `build.rollupOptions.output.inlineDynamicImports` now defaults to `false` when `ssr.target` is `'node'`. `inlineDynamicImports` changes execution order and bundling to a single file is not needed for node builds.

General Changes

- JS file extensions in SSR and lib mode now use a valid extension (`js`, `mjs`, or `cjs`) for output JS entries and chunks based on their format and the package type.
- Terser is now an optional dependency. If you are using `build.minify: 'terser'`, you need to install it.

```
npm add -D terser
```

`import.meta.glob`

- Raw `import.meta.glob` switched from `{ assert: { type: 'raw' }}` to `{ as: 'raw' }`
- Keys of `import.meta.glob` are now relative to the current module.

```
// file: /foo/index.js
const modules = import.meta.glob('../foo/*.js')

// transformed:
const modules = {
-  '../foo/bar.js': () => {}
+  './bar.js': () => {}
}
```

- When using an alias with `import.meta.glob`, the keys are always absolute.
- `import.meta.globEager` is now deprecated. Use `import.meta.glob('*', { eager: true })` instead.

WebAssembly Support

`import init from 'example.wasm'` syntax is dropped to prevent future collision with "ESM integration for Wasm". You can use `?init` which is similar to the previous behavior.

```diff
-import init from 'example.wasm'
+import init from 'example.wasm?init'

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

Automatic https Certificate Generation

A valid certificate is needed when using `https`. In Vite v2, if no certificate was configured, a self-signed certificate was automatically created and cached. Since Vite v3, we recommend manually creating your certificates. If you still want to use the automatic generation from v2, this feature can be enabled back by adding @vitejs/plugin-basic-ssl to the project plugins.

```js
import basicSsl from '@vitejs/plugin-basic-ssl'

export default {
  plugins: [basicSsl()]
}
```

Experimental

Using esbuild deps optimization at build time

In v3, Vite allows the use of esbuild to optimize dependencies during build time. If enabled, it removes one of the most significant differences between dev and prod present in v2. `@rollup/plugin-commonjs` is no longer needed in this case since esbuild converts CJS-only dependencies to ESM.

If you want to try this build strategy, you can use `optimizeDeps.disabled: false` (the default in v3 is `disabled: 'build'`). `@rollup/plugin-commonjs` can be removed by passing `build.commonjsOptions: { include: [] }`

Advanced

There are some changes which only affect plugin/tool creators.

- [#5868] refactor: remove deprecated api for 3.0
  - `printHttpServerUrls` is removed
  - `server.app`, `server.transformWithEsbuild` are removed
  - `import.meta.hot.acceptDeps` is removed
- [#6901] fix: sequential injection of tags in transformIndexHtml
  - `transformIndexHtml` now gets the correct content modified by earlier plugins, so the order of the injected tags now works as expected.
- [#7995] chore: do not fixStacktrace
  - `ssrLoadModule`'s `fixStacktrace` option's default is now `false`
- [#8178] feat!: migrate to ESM
  - `formatPostcssSourceMap` is now async

    - `resolvePackageEntry`, `resolvePackageData` are no longer available from CJS build (dynamic import is needed to use in CJS)
- [#8626] refactor: type client maps
    - Type of callback of `import.meta.hot.accept` is now stricter. It is now `(mod: (Record<string, any> & { [Symbol.toStringTag]: 'Module' }) | undefined) => void` (was `(mod: any) => void`).

Also there are other breaking changes which only affect few users.

- [#5018] feat: enable `generatedCode: 'es2015'` for rollup build
    - Transpile to ES5 is now necessary even if the user code only includes ES5.
- [#7877] fix: vite client types
    - `/// <reference lib="dom" />` is removed from `vite/client.d.ts`. `{ "lib": ["dom"] }` or `{ "lib": ["webworker"] }` is necessary in `tsconfig.json`.
- [#8090] feat: preserve process env vars in lib build
    - `process.env.*` is now preserved in library mode
- [#8280] feat: non-blocking esbuild optimization at build time
    - `server.force` option was removed in favor of `optimizeDeps.force` option.
- [#8550] fix: dont handle sigterm in middleware mode
    - When running in middleware mode, Vite no longer kills process on `SIGTERM`.

Migration from v1

Check the Migration from v1 Guide in the Vite v2 docs first to see the needed changes to port your app to Vite v2, and then proceed with the changes on this page.

Server-Side Rendering

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 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.

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). The Vite playground contains example SSR setups for Vue 3 and React, which can be used as references for this guide:

- Vue 3
- React

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

```
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:

server.js

```
import fs from 'fs'
import path from 'path'
import { fileURLToPath } from '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
  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. `vite.middlewares` is a 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
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. vite.ssrLoadModule automatically transforms
    //    your 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.
    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:

```
{
  "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 instead, 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')` instead (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 Vue and React demos for a working setup.

Generating Preload Directives

`vite build` supports the `--ssrManifest` flag which will generate `ssr-manifest.json` in build output directory:

```
- "build:client": "vite build --outDir dist/client",
+ "build:client": "vite build --outDir dist/client --ssrManifest",
```

The above script will now generate `dist/client/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:

```
// 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 for a full example.

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 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`.

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`.

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.js.org/en/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:

```
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.

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

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` and to the preview server with `configurePreviewServer`.

Note

Use a post hook so that your SSR middleware runs *after* Vite's middlewares.

SSR Format

By default, Vite generates the SSR bundle in ESM. There is experimental support for configuring `ssr.format`, but it isn't recommended. Future efforts around SSR development will be based on ESM, and CommonJS remains available for backward compatibility. If using ESM for SSR isn't possible in your project, you can set `legacy.buildSsrCjsExternalHeuristics: true` to generate a CJS bundle using the same externalization heuristics of Vite v2.

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` , 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:

```
{
  "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.

```
$ 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.

```
$ npm run build
$ 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/`, you can omit `base` as it defaults to `'/'`.

   If you are deploying to `https://<USERNAME>.github.io/<REPO>/`, for example your repository is at `https://github.com/<USERNAME>/<REPO>`, then set `base` to `'/<REPO>/'`.

2. Inside your project, create `deploy.sh` with the following content (with highlighted lines uncommented appropriately), and run it to deploy:

   ```sh
   #!/usr/bin/env sh

   # abort on errors
   set -e

   # build
   npm run build

   # navigate into the build output directory
   cd dist

   # place .nojekyll to bypass Jekyll processing
   echo > .nojekyll

   # if you are deploying to a custom domain
   # echo 'www.example.com' > CNAME

   git init
   git checkout -B main
   git add -A
   git commit -m 'deploy'

   # if you are deploying to https://<USERNAME>.github.io
   # git push -f git@github.com:<USERNAME>/<USERNAME>.github.io.git main

   # if you are deploying to https://<USERNAME>.github.io/<REPO>
   # git push -f git@github.com:<USERNAME>/<REPO>.git main:gh-pages

   cd -
   ```

tip You can also run the above script in your CI setup to enable automatic deployment on each push.

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.
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 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, and all changes made to the Production Branch (commonly "main") will result in a Production Deployment.

Vercel

Vercel CLI

1. Install the Vercel 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)

```
$ 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 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, and all changes made to the Production Branch (commonly "main") will result in a Production Deployment.

Learn more about Vercel's Git Integration.

Cloudflare Pages

Cloudflare Pages via Wrangler

1. Install Wrangler CLI.
2. Authenticate Wrangler with your Cloudflare account using `wrangler login`.
3. Run your build command.
4. Deploy using `npx wrangler pages publish dist`.

```
# Install Wrangler CLI
$ npm install -g wrangler

# Login to Cloudflare account from CLI
$ wrangler login

# Run your build command
$ npm run build

# Create new deployment
$ npx wrangler pages publish 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 unless specified not to in your branch build controls. All changes to the Production Branch (commonly "main") 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.

Google Firebase

1. Make sure you have firebase-tools installed.

2. Create `firebase.json` and `.firebaserc` at the root of your project with the following content:

   `firebase.json`:

   ```
   {
     "hosting": {
       "public": "dist",
       "ignore": [],
       "rewrites": [
         {
           "source": "**",
           "destination": "/index.html"
         }
       ]
     }
   }
   ```

`.firebaserc`:

```
{
  "projects": {
    "default": "<YOUR_FIREBASE_ID>"
  }
}
```

3. After running `npm run build`, deploy using the command `firebase deploy`.

Surge

1. First install surge, if you haven't already.

2. Run `npm run build`.

3. Deploy to surge by typing `surge dist`.

You can also deploy to 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 service. You need:

- An Azure account and a subscription key. You can create a free Azure account here.
- Your app code pushed to GitHub.
- The SWA Extension in Visual Studio Code.

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.

1. Create a Render account.

2. In the Dashboard, 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 can be configured in the project settings.

You can also add a custom domain to your project.

Troubleshooting

See Rollup's troubleshooting guide for more information too.

If the suggestions here don't work, please try posting questions on GitHub Discussions or in the `#help` channel of Vite Land Discord.

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).

You will need to either:

- Switch to another package manager (e.g. `pnpm`, `yarn`)
- Remove `&` from the path to your project

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`

```
# 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`

```
# 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

Note that these settings persist but a **restart is required**.

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.

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` 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:

```
import './Foo.js' // should be './foo.js'
```

Related issue: #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.

A full reload happens instead of HMR

If HMR is not handled by Vite or a plugin, a full reload will happen.

Also if there is a dependency loop, a full reload will happen. To solve this, try removing the loop.

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 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`.

Others

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 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 code are used inside dependencies, you could use `patch-package` (or `yarn patch` or `pnpm patch`) for an escape hatch.

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 can be used:

```
$ npm add -D @vitejs/plugin-legacy
```

```
// 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

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 for information about official plugins. Community plugins are listed in awesome-vite. For compatible Rollup plugins, check out Vite Rollup Plugins for a list of compatible official Rollup plugins with usage instructions or the Rollup Plugin Compatibility section in case it is not listed there.

You can also find plugins that follow the recommended conventions using a npm search for vite-plugin for Vite plugins or a npm search for rollup-plugin 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

```
// 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 for detailed information, and look out for the `enforce` label and usage instructions for popular plugins in the Vite Rollup Plugins compatibility listing.

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'`:

```
// 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 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, Rollup and Parcel, 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 using esbuild. 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. 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.

Slow Updates

When a file is edited in a bundler-based build setup, it is inefficient to rebuild the whole bundle for obvious reasons: 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 that bakes in many performance optimizations out of the box.

Why Not Bundle with esbuild?

While `esbuild` is extremely fast and is already a very capable bundler for libraries, some of the important features needed for bundling *applications* are still work in progress - in particular code-splitting and CSS handling. For the time being, Rollup is more mature and flexible in these regards. That said, we won't rule out the possibility of using `esbuild` for production builds when it stabilizes these features in the future.

How is Vite Different from X?

You can check out the Comparisons section for more details on how Vite differs from other similar tools.

Colophon

This book is created by using the following sources:

- Vite - English
- GitHub source: vitejs/vite/docs
- Created: 2022-11-10
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- GitHub repo: https://github.com/shinokada/js-framwork-docs