JavaScript API

- createServer
- InlineConfig
- ResolvedConfig
- ViteDevServer
- build
- preview
- PreviewServer
- resolveConfig
- mergeConfig
- searchForWorkspaceRoot
- loadEnv
- normalizePath
- transformWithOxc
- transformWithEsbuild
- loadConfigFromFile
- preprocessCSS
- version
- rolldownVersion
- esbuildVersion
- rollupVersion

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 { createServer } from 'vite'

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

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

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

::: tip NOTE When using middleware mode combined with proxy config for WebSocket, the parent http server should be provided in `middlewareMode` to bind the proxy correctly.

Example

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

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

const vite = await createServer({
  server: {
    // Enable middleware mode
    middlewareMode: {
      // Provide the parent http server for proxy WebSocket
```

```
      server: parentServer,
    },
    proxy: {
      '/ws': {
        target: 'ws://localhost:3000',
        // Proxying WebSocket
        ws: true,
      },
    },
  },
})

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

:::

`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.

`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. If `config.server.watch` is set to `null`,
 * it will not watch any files and calling `add` or `unwatch` will have no effect.
 * https://github.com/paulmillr/chokidar/tree/3.6.0#api
 */
watcher: FSWatcher
/**
 * WebSocket 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 (URL-encoded). Returns `null`
 * in middleware mode or if the server is not listening on any port.
 */
resolvedUrls: ResolvedServerUrls | null
```

```
/**
 * Programmatically resolve, load and transform a URL and get the result
 * without going through the http request pipeline.
 */
transformRequest(
  url: string,
  options?: TransformOptions,
): Promise<TransformResult | null>
/**
 * Apply Vite built-in HTML transforms and any plugin HTML transforms.
 */
transformIndexHtml(
  url: string,
  html: string,
  originalUrl?: string,
): Promise<string>
/**
 * Load a given URL as an instantiated module for SSR.
 */
ssrLoadModule(
  url: string,
  options?: { fixStacktrace?: boolean },
): Promise<Record<string, any>>
/**
 * Fix ssr error stacktrace.
 */
ssrFixStacktrace(e: Error): void
/**
 * Triggers HMR for a module in the module graph. You can use the `server.moduleGraph`
 * API to retrieve the module to be reloaded. If `hmr` is false, this is a no-op.
 */
reloadModule(module: ModuleNode): Promise<void>
/**
 * Start the server.
 */
listen(port?: number, isRestart?: boolean): Promise<ViteDevServer>
/**
 * Restart the server.
 *
 * @param forceOptimize - force the optimizer to re-bundle, same as --force cli flag
```

```
   */
  restart(forceOptimize?: boolean): Promise<void>
  /**
   * Stop the server.
   */
  close(): Promise<void>
  /**
   * Bind CLI shortcuts
   */
  bindCLIShortcuts(options?: BindCLIShortcutsOptions<ViteDevServer>): void
  /**
   * Calling `await server.waitForRequestsIdle(id)` will wait until all static imports
   * are processed. If called from a load or transform plugin hook, the id needs to be
   * passed as a parameter to avoid deadlocks. Calling this function after the first
   * static imports section of the module graph has been processed will resolve immediately.
   * @experimental
   */
  waitForRequestsIdle: (ignoredId?: string) => Promise<void>
}
```

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

build

Type Signature:

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

Example Usage:

```
import path from 'node:path'
import { build } from 'vite'

await build({
  root: path.resolve(import.meta.dirname, './project'),
  base: '/foo/',
  build: {
    rolldownOptions: {
      // ...
    },
  },
})
```

preview

Type Signature:

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

Example Usage:

```
import { preview } from 'vite'

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

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

PreviewServer

```
interface PreviewServer {
  /**
   * The resolved vite config object
   */
  config: ResolvedConfig
  /**
   * A connect app instance.
   * - Can be used to attach custom middlewares to the preview server.
   * - Can also be used as the handler function of a custom http server
   *   or as a middleware in any connect-style Node.js frameworks
   *
   * https://github.com/senchalabs/connect#use-middleware
   */
  middlewares: Connect.Server
  /**
   * native Node http server instance
   */
  httpServer: http.Server
  /**
   * The resolved urls Vite prints on the CLI (URL-encoded). Returns `null`
   * if the server is not listening on any port.
   */
  resolvedUrls: ResolvedServerUrls | null
  /**
   * Print server urls
   */
  printUrls(): void
  /**
   * Bind CLI shortcuts
   */
  bindCLIShortcuts(options?: BindCLIShortcutsOptions<PreviewServer>): void
}
```

resolveConfig

Type Signature:

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

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

`mergeConfig`

Type Signature:

```
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.

Note that `null` and `undefined` values in `overrides` are skipped and not merged. If you need to explicitly clear a value from `defaults`, modify the result of `mergeConfig` directly.

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

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

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

// ---cut---
```

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

:::

searchForWorkspaceRoot

Type Signature:

```
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 fallback 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` and merge them with the matching variables already present in `process.env`. 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.

`transformWithOxc`

Type Signature:

```
async function transformWithOxc(
  code: string,
  filename: string,
  options?: OxcTransformOptions,
  inMap?: object,
): Promise<Omit<OxcTransformResult, 'errors'> & { warnings: string[] }>
```

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

`transformWithEsbuild`

Type Signature:

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

Deprecated: Use `transformWithOxc` instead.

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,
  customLogger?: Logger,
): Promise<{
  path: string
  config: UserConfig
  dependencies: string[]
} | null>
```

Load a Vite config file manually with Rolldown.

`preprocessCSS`

- **Experimental:** Give Feedback

Type Signature:

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

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

```
}
```

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

The pre-processor used is inferred from the `filename` extension. If the `filename` ends with `.module.{ext}`, it is inferred as a CSS module and the returned result will include a `modules` object mapping the original class names to the transformed ones.

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

version

Type: `string`

The current version of Vite as a string (e.g. `"8.0.0"`).

rolldownVersion

Type: `string`

The version of Rolldown used by Vite as a string (e.g. `"1.0.0"`). A re-export of `VERSION` from `rolldown`.

esbuildVersion

Type: `string`

Only kept for backward compatibility.

rollupVersion

Type: `string`

Only kept for backward compatibility.

□□□ #2
□ □□□ □□□ 2026-10-31 13:13:58 UTC
□ □□□ □□□ 2026-10-31 13:13:59 UTC