Static Asset Handling

- Importing Asset as URL
  - Explicit URL Imports
  - Explicit Inline Handling
  - Importing Asset as String
  - Importing Script as a Worker
- The public Directory
- new URL(url, import.meta.url)

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

For example, `imgUrl` will be `/src/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.
- TypeScript, by default, does not recognize static asset imports as valid modules. To fix this, include `vite/client`.

::: tip Inlining SVGs through `url()` When passing a URL of SVG to a manually constructed `url()` by JS, the variable should be wrapped within double quotes.

```
import 'vite/client'
// ---cut---
import imgUrl from './img.svg'
document.getElementById('hero-img').style.background = `url("${imgUrl}")`
```

:::

Explicit URL Imports

Assets that are not included in the internal list or in `assetsInclude` can be explicitly imported as a URL using the `?url` suffix. This is useful, for example, to import Houdini Paint Worklets.

```
import 'vite/client'
// ---cut---
import workletURL from 'extra-scalloped-border/worklet.js?url'
CSS.paintWorklet.addModule(workletURL)
```

Explicit Inline Handling

Assets can be explicitly imported with inlining or no inlining using the `?inline` or `?no-inline` suffix respectively.

```
import 'vite/client'
// ---cut---
// prettier-ignore
import imgUrl1 from './img.svg?no-inline'
import imgUrl2 from './img.png?inline'
```

Importing Asset as String

Assets can be imported as strings using the `?raw` suffix.

```
import 'vite/client'
// ---cut---
import shaderString from './shader.glsl?raw'
```

Importing Script as a Worker

Scripts can be imported as web workers with the `?worker` or `?sharedworker` suffix.

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

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

```
import 'vite/client'
// ---cut---
// 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`.

::: tip Choosing between imports and the `public` directory

In general, prefer **importing assets** unless you specifically need the guarantees provided by the `public` directory.

:::

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:

```
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) {
  // note that this does not include files in subdirectories
  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
```

::: details How it works

Vite will transform the `getImageUrl` function to:

```
import __img0png from './dir/img0.png'
import __img1png from './dir/img1.png'

function getImageUrl(name) {
  const modules = {
    './dir/img0.png': __img0png,
    './dir/img1.png': __img1png,
  }
  return new URL(modules[`./dir/${name}.png`], 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` has different semantics in browsers vs. Node.js. The server bundle also cannot determine the client host URL ahead of time. :::

□□□ #2
□ □□□ □□□ 2026-10-31 13:14:03 UTC
□ □□□ □□□ 2026-10-31 13:14:04 UTC