SVELTEKIT • ADVANCED

Packaging

ON THIS PAGE

You can use SvelteKit to build apps as well as component libraries, using the `@sveltejs/package` package (`npx sv create` has an option to set this up for you).

When you're creating an app, the contents of `src/routes` is the public-facing stuff; `src/lib` contains your app's internal library.

A component library has the exact same structure as a SvelteKit app, except that `src/lib` is the public-facing bit, and your root `package.json` is used to publish the package. `src/routes` might be a documentation or demo site that accompanies the library, or it might just be a sandbox you use during development.

Running the `svelte-package` command from `@sveltejs/package` will take the contents of `src/lib` and generate a `dist` directory (which can be configured) containing the following:

- All the files in `src/lib` . Svelte components will be preprocessed, TypeScript files will be transpiled to JavaScript.
- Type definitions (`d.ts` files) which are generated for Svelte, JavaScript and TypeScript files. You need to install `typescript >= 4.0.0` for this. Type definitions are placed next to their implementation, hand-written `d.ts` files are copied over as is. You can disable generation, but we strongly recommend against it — people using your library might use TypeScript, for which they require these type definition files.

`@sveltejs/package` version 1 generated a `package.json` . This is no longer the case and it will now use the `package.json` from your project and validate that it is correct instead. If you're

Since you're now building a library for public use, the contents of your `package.json` will become more important. Through it, you configure the entry points of your package, which files are published to npm, and which dependencies your library has. Let's go through the most important fields one by one.

name

This is the name of your package. It will be available for others to install using that name, and visible on `https://npmjs.com/package/<name>`.

```
{
  "name": "your-library"
}
```

Read more about it here.

license

Every package should have a license field so people know how they are allowed to use it. A very popular license which is also very permissive in terms of distribution and reuse without warranty is `MIT`.

```
{
  "license": "MIT"
}
```

Read more about it here. Note that you should also include a `LICENSE` file in your package.

files

This tells npm which files it will pack up and upload to npm. It should contain your output folder (`dist` by default). Your `package.json` and `README` and `LICENSE` will always

```
  "files": ["dist"]
}
```

To exclude unnecessary files (such as unit tests, or modules that are only imported from `src/routes` etc) you can add them to an `.npmignore` file. This will result in smaller packages that are faster to install.

Read more about it here.

exports

The `"exports"` field contains the package's entry points. If you set up a new library project through `npx sv create`, it's set to a single export, the package root:

```
{
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
      "svelte": "./dist/index.js"
    }
  }
}
```

This tells bundlers and tooling that your package only has one entry point, the root, and everything should be imported through that, like this:

```
import { Something } from 'your-library';
```

The `types` and `svelte` keys are export conditions. They tell tooling what file to import when they look up the `your-library` import:

- TypeScript sees the `types` condition and looks up the type definition file. If you don't publish type definitions, omit this condition.
- Svelte-aware tooling sees the `svelte` condition and knows this is a Svelte component

replace this condition with `default`.

> Previous versions of `@sveltejs/package` also added a `package.json` export. This is no longer part of the template because all tooling can now deal with a `package.json` not being explicitly exported.

You can adjust `exports` to your liking and provide more entry points. For example, if instead of a `src/lib/index.js` file that re-exported components you wanted to expose a `src/lib/Foo.svelte` component directly, you could create the following export map...

```
{
  "exports": {
    "./Foo.svelte": {
      "types": "./dist/Foo.svelte.d.ts",
      "svelte": "./dist/Foo.svelte"
    }
  }
}
```

...and a consumer of your library could import the component like so:

```
import Foo from 'your-library/Foo.svelte';
```

> Beware that doing this will need additional care if you provide type definitions. Read more about the caveat here

In general, each key of the exports map is the path the user will have to use to import something from your package, and the value is the path to the file that will be imported or a map of export conditions which in turn contains these file paths.

Read more about `exports` here.

svelte

Docs

longer necessary when using the `svelte` export condition, but for backwards compatibility with outdated tooling that doesn't yet know about export conditions it's good to keep it around. It should point towards your root entry point.

```
{
  "svelte": "./dist/index.js"
}
```

sideEffects

The `sideEffects` field in `package.json` is used by bundlers to determine if a module may contain code that has side effects. A module is considered to have side effects if it makes changes that are observable from other scripts outside the module when it's imported. For example, side effects include modifying global variables or the prototype of built-in JavaScript objects. Because a side effect could potentially affect the behavior of other parts of the application, these files/modules will be included in the final bundle regardless of whether their exports are used in the application. It is a best practice to avoid side effects in your code.

Setting the `sideEffects` field in `package.json` can help the bundler to be more aggressive in eliminating unused exports from the final bundle, a process known as tree-shaking. This results in smaller and more efficient bundles. Different bundlers handle `sideEffects` in various manners. While not necessary for Vite, we recommend that libraries state that all CSS files have side effects so that your library will be compatible with webpack. This is the configuration that comes with newly created projects:

package.json

```
{
  "sideEffects": ["**/*.css"]
}
```

> If the scripts in your library have side effects, ensure that you update the `sideEffects` field. All

package.json

```json
{
  "sideEffects": [
    "**/*.css",
    "./dist/sideEffectfulFile.js"
  ]
}
```

This will treat only the specified files as having side effects.

TypeScript

You should ship type definitions for your library even if you don't use TypeScript yourself so that people who do get proper intellisense when using your library. `@sveltejs/package` makes the process of generating types mostly opaque to you. By default, when packaging your library, type definitions are auto-generated for JavaScript, TypeScript and Svelte files. All you need to ensure is that the `types` condition in the exports map points to the correct files. When initialising a library project through `npx sv create`, this is automatically setup for the root export.

If you have something else than a root export however — for example providing a `your-library/foo` import — you need to take additional care for providing type definitions. Unfortunately, TypeScript by default will *not* resolve the `types` condition for an export like `{ "./foo": { "types": "./dist/foo.d.ts", ... }}`. Instead, it will search for a `foo.d.ts` relative to the root of your library (i.e. `your-library/foo.d.ts` instead of `your-library/dist/foo.d.ts`). To fix this, you have two options:

The first option is to require people using your library to set the `moduleResolution` option in their `tsconfig.json` (or `jsconfig.json`) to `bundler` (available since TypeScript 5, the best and recommended option in the future), `node16` or `nodenext`. This opts TypeScript into actually looking at the exports map and resolving the types correctly.

types. This is a field inside `package.json` TypeScript uses to check for different type definitions depending on the TypeScript version, and also contains a path mapping feature for that. We leverage that path mapping feature to get what we want. For the mentioned `foo` export above, the corresponding `typesVersions` looks like this:

```
{
  "exports": {
    "./foo": {
      "types": "./dist/foo.d.ts",
      "svelte": "./dist/foo.js"
    }
  },
  "typesVersions": {
    ">4.0": {
      "foo": ["./dist/foo.d.ts"]
    }
  }
}
```

`>4.0` tells TypeScript to check the inner map if the used TypeScript version is greater than 4 (which should in practice always be true). The inner map tells TypeScript that the typings for `your-library/foo` are found within `./dist/foo.d.ts`, which essentially replicates the `exports` condition. You also have `*` as a wildcard at your disposal to make many type definitions at once available without repeating yourself. Note that if you opt into `typesVersions` you have to declare all type imports through it, including the root import (which is defined as `"index.d.ts": [..]`).

You can read more about that feature here.

Best practices

You should avoid using SvelteKit-specific modules like `$app/environment` in your packages unless you intend for them to only be consumable by other SvelteKit projects. E.g. rather than using `import { browser } from '$app/environment'` you could use `import { BROWSER }`

`$app/navigation` , etc. Writing your app in this more generic fashion will also make it easier to setup tools for testing, UI demos and so on.

Ensure that you add aliases via `svelte.config.js` (not `vite.config.js` or `tsconfig.json`), so that they are processed by `svelte-package` .

You should think carefully about whether or not the changes you make to your package are a bug fix, a new feature, or a breaking change, and update the package version accordingly. Note that if you remove any paths from `exports` or any `export` conditions inside them from your existing library, that should be regarded as a breaking change.

```
{
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
// changing `svelte` to `default` is a breaking change:
      "svelte": "./dist/index.js"
      "default": "./dist/index.js"
    },
// removing this is a breaking change:
    "./foo": {
      "types": "./dist/foo.d.ts",
      "svelte": "./dist/foo.js",
      "default": "./dist/foo.js"
    },
// adding this is ok:
    "./bar": {
      "types": "./dist/bar.d.ts",
      "svelte": "./dist/bar.js",
      "default": "./dist/bar.js"
    }
  }
}
```

Options

`svelte-package` accepts the following options:

Docs

Defaults to `src/lib`

`-o` / `--output` — the output directory where the processed files are written to. Your `package.json`'s `exports` should point to files inside there, and the `files` array should include that folder. Defaults to `dist`

`-t` / `--types` — whether or not to create type definitions (`d.ts` files). We strongly recommend doing this as it fosters ecosystem library quality. Defaults to `true`

`--tsconfig` - the path to a tsconfig or jsconfig. When not provided, searches for the next upper tsconfig/jsconfig in the workspace path.

Publishing

To publish the generated package:

```
npm publish
```

Caveats

All relative file imports need to be fully specified, adhering to Node's ESM algorithm. This means that for a file like `src/lib/something/index.js`, you must include the filename with the extension:

```
import { something } from './something/index.js';
```

If you are using TypeScript, you need to import `.ts` files the same way, but using a `.js` file ending, *not* a `.ts` file ending. (This is a TypeScript design decision outside our control.) Setting `"moduleResolution": "NodeNext"` in your `tsconfig.json` or `jsconfig.json` will help you with this.

All files except Svelte files (preprocessed) and TypeScript files (transpiled to JavaScript) are