Typically, you won't interact with the Svelte compiler directly, but will instead integrate it into your build system using a bundler plugin:

- rollup-plugin-svelte for users of Rollup
- svelte-loader for users of webpack
- or one of the community-maintained plugins

Nonetheless, it's useful to understand how to use the compiler, since bundler plugins generally expose compiler options to you.

`svelte.compile`

```
result: {
    js,
    css,
    ast,
    warnings,
    vars,
    stats
} = svelte.compile(source: string, options?: {...})
```

This is where the magic happens. `svelte.compile` takes your component source code, and turns it into a JavaScript module that exports a class.

```
const svelte = require('svelte/compiler');

const result = svelte.compile(source, {
    // options
});
```

The following options can be passed to the compiler. None are required:

option	default	description
`filename`	`null`	`string` used for debugging hints and sourcemaps. Your bundler plugin will set it automatically.
`name`	`"Component"`	`string` that sets the name of the resulting JavaScript class (though the compiler will rename it if it would otherwise conflict with other variables in scope). It will normally be inferred from `filename`.
`format`	`"esm"`	If `"esm"`, creates a JavaScript module (with `import` and `export`). If `"cjs"`, creates a CommonJS module (with `require` and `module.exports`), which is useful in some server-side rendering situations or for testing.
`generate`	`"dom"`	If `"dom"`, Svelte emits a JavaScript class for mounting to the DOM. If `"ssr"`, Svelte emits an object with a `render` method suitable for server-side rendering. If `false`, no JavaScript or CSS is returned; just metadata.
`errorMode`	`"throw"`	If `"throw"`, Svelte throws when a compilation error occurred. If `"warn"`,

		Svelte will treat errors as warnings and add them to the warning report.
`varsReport`	`"strict"`	If `"strict"`, Svelte returns a variables report with only variables that are not globals nor internals. If `"full"`, Svelte returns a variables report with all detected variables. If `false`, no variables report is returned.
`dev`	`false`	If `true`, causes extra code to be added to components that will perform runtime checks and provide debugging information during development.
`immutable`	`false`	If `true`, tells the compiler that you promise not to mutate any objects. This allows it to be less conservative about checking whether values have changed.
`hydratable`	`false`	If `true` when generating DOM code, enables the `hydrate: true` runtime option, which allows a component to upgrade existing DOM rather than creating new DOM from scratch. When generating SSR code, this adds markers to `<head>` elements so that hydration knows which to replace.
`legacy`	`false`	If `true`, generates code that will work in IE9 and IE10, which don't support things like `element.dataset`.
`accessors`	`false`	If `true`, getters and setters will be created for the component's props. If `false`, they will only be created for readonly exported values (i.e. those declared with `const`, `class` and `function`). If compiling with `customElement: true` this option defaults to `true`.
`customElement`	`false`	If `true`, tells the compiler to generate a custom element constructor instead of a regular Svelte component.
`tag`	`null`	A `string` that tells Svelte what tag name to register the custom element with. It must be a lowercase alphanumeric string with at least one hyphen, e.g. `"my-element"`.
`css`	`true`	If `true`, styles will be included in the JavaScript class and injected at runtime. It's recommended that you set this to `false` and use the CSS that is statically generated, as it will result in smaller JavaScript bundles and better performance.
`cssHash`	See right	A function that takes a `{ hash, css, name, filename }` argument and returns the string that is used as a classname for scoped CSS. It defaults to returning `svelte-${hash(css)}`
`loopGuardTimeout`	0	A `number` that tells Svelte to break the loop if it blocks the thread for more than `loopGuardTimeout` ms. This is useful to prevent infinite loops. **Only available when `dev: true`**
`preserveComments`	`false`	If `true`, your HTML comments will be preserved during server-side rendering. By default, they are stripped out.
`preserveWhitespace`	`false`	If `true`, whitespace inside and between elements is kept as you typed it, rather than removed or collapsed to a single space where possible.

`sourcemap`	`object | string`	An initial sourcemap that will be merged into the final output sourcemap. This is usually the preprocessor sourcemap.
`enableSourcemap`	`boolean | { js: boolean; css: boolean; }`	If `true`, Svelte generate sourcemaps for components. Use an object with `js` or `css` for more granular control of sourcemap generation. By default, this is `true`.
`outputFilename`	`null`	A `string` used for your JavaScript sourcemap.
`cssOutputFilename`	`null`	A `string` used for your CSS sourcemap.
`sveltePath`	`"svelte"`	The location of the `svelte` package. Any imports from `svelte` or `svelte/[module]` will be modified accordingly.
`namespace`	`"html"`	The namespace of the element; e.g., `"mathml"`, `"svg"`, `"foreign"`.

The returned `result` object contains the code for your component, along with useful bits of metadata.

```
const {
    js,
    css,
    ast,
    warnings,
    vars,
    stats
} = svelte.compile(source);
```

- `js` and `css` are objects with the following properties:
  - `code` is a JavaScript string
  - `map` is a sourcemap with additional `toString()` and `toUrl()` convenience methods
- `ast` is an abstract syntax tree representing the structure of your component.
- `warnings` is an array of warning objects that were generated during compilation. Each warning has several properties:
  - `code` is a string identifying the category of warning
  - `message` describes the issue in human-readable terms
  - `start` and `end`, if the warning relates to a specific location, are objects with `line`, `column` and `character` properties
  - `frame`, if applicable, is a string highlighting the offending code with line numbers
- `vars` is an array of the component's declarations, used by eslint-plugin-svelte3 for example. Each variable has several properties:
  - `name` is self-explanatory
  - `export_name` is the name the value is exported as, if it is exported (will match `name` unless you do `export...as`)
  - `injected` is `true` if the declaration is injected by Svelte, rather than in the code you wrote
  - `module` is `true` if the value is declared in a `context="module"` script
  - `mutated` is `true` if the value's properties are assigned to inside the component
  - `reassigned` is `true` if the value is reassigned inside the component

    - `referenced` is `true` if the value is used in the template
    - `referenced_from_script` is `true` if the value is used in the `<script>` outside the declaration
    - `writable` is `true` if the value was declared with `let` or `var` (but not `const`, `class` or `function`)
- `stats` is an object used by the Svelte developer team for diagnosing the compiler. Avoid relying on it to stay the same!

`svelte.parse`

```
ast: object = svelte.parse(
    source: string,
    options?: {
        filename?: string,
        customElement?: boolean
    }
)
```

The `parse` function parses a component, returning only its abstract syntax tree. Unlike compiling with the `generate: false` option, this will not perform any validation or other analysis of the component beyond parsing it. Note that the returned AST is not considered public API, so breaking changes could occur at any point in time.

```
const svelte = require('svelte/compiler');

const ast = svelte.parse(source, { filename: 'App.svelte' });
```

`svelte.preprocess`

A number of community-maintained preprocessing plugins are available to allow you to use Svelte with tools like TypeScript, PostCSS, SCSS, and Less.

You can write your own preprocessor using the `svelte.preprocess` API.

```
result: {
    code: string,
    dependencies: Array<string>
} = await svelte.preprocess(
    source: string,
    preprocessors: Array<{
        markup?: (input: { content: string, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>,
        script?: (input: { content: string, markup: string, attributes:
Record<string, string>, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>,
        style?: (input: { content: string, markup: string, attributes:
```

```
Record<string, string>, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>
    }>,
    options?: {
        filename?: string
    }
)
```

The `preprocess` function provides convenient hooks for arbitrarily transforming component source code. For example, it can be used to convert a `<style lang="sass">` block into vanilla CSS.

The first argument is the component source code. The second is an array of *preprocessors* (or a single preprocessor, if you only have one), where a preprocessor is an object with `markup`, `script` and `style` functions, each of which is optional.

Each `markup`, `script` or `style` function must return an object (or a Promise that resolves to an object) with a `code` property, representing the transformed source code, and an optional array of `dependencies`.

The `markup` function receives the entire component source text, along with the component's `filename` if it was specified in the third argument.

> *Preprocessor functions should additionally return a `map` object alongside `code` and `dependencies`, where `map` is a sourcemap representing the transformation.*

```
const svelte = require('svelte/compiler');
const MagicString = require('magic-string');

const { code } = await svelte.preprocess(source, {
    markup: ({ content, filename }) => {
        const pos = content.indexOf('foo');
        if(pos < 0) {
            return { code: content }
        }
        const s = new MagicString(content, { filename })
        s.overwrite(pos, pos + 3, 'bar', { storeName: true })
        return {
            code: s.toString(),
            map: s.generateMap()
        }
    }
}, {
    filename: 'App.svelte'
});
```

The `script` and `style` functions receive the contents of `<script>` and `<style>` elements respectively (`content`) as well as the entire component source text (`markup`). In addition to `filename`, they get an object of the element's attributes.

If a `dependencies` array is returned, it will be included in the result object. This is used by packages like rollup-plugin-svelte to watch additional files for changes, in the case where your `<style>` tag has an `@import` (for example).

```
const svelte = require('svelte/compiler');
const sass = require('node-sass');
const { dirname } = require('path');

const { code, dependencies } = await svelte.preprocess(source, {
    style: async ({ content, attributes, filename }) => {
        // only process <style lang="sass">
        if (attributes.lang !== 'sass') return;

        const { css, stats } = await new Promise((resolve, reject) => sass.render({
            file: filename,
            data: content,
            includePaths: [
                dirname(filename),
            ],
        }, (err, result) => {
            if (err) reject(err);
            else resolve(result);
        }));

        return {
            code: css.toString(),
            dependencies: stats.includedFiles
        };
    }
}, {
    filename: 'App.svelte'
});
```

Multiple preprocessors can be used together. The output of the first becomes the input to the second. `markup` functions run first, then `script` and `style`.

```
const svelte = require('svelte/compiler');

const { code } = await svelte.preprocess(source, [
    {
        markup: () => {
            console.log('this runs first');
        },
        script: () => {
            console.log('this runs third');
        },
        style: () => {
            console.log('this runs fifth');
        }
    },
```

```
    {
        markup: () => {
            console.log('this runs second');
        },
        script: () => {
            console.log('this runs fourth');
        },
        style: () => {
            console.log('this runs sixth');
        }
    }
], {
    filename: 'App.svelte'
});
```

`svelte.walk`

```
walk(ast: Node, {
    enter(node: Node, parent: Node, prop: string, index: number)?: void,
    leave(node: Node, parent: Node, prop: string, index: number)?: void
})
```

The `walk` function provides a way to walk the abstract syntax trees generated by the parser, using the compiler's own built-in instance of estree-walker.

The walker takes an abstract syntax tree to walk and an object with two optional methods: `enter` and `leave`. For each node, `enter` is called (if present). Then, unless `this.skip()` is called during `enter`, each of the children are traversed, and then `leave` is called on the node.

```
const svelte = require('svelte/compiler');
svelte.walk(ast, {
    enter(node, parent, prop, index) {
        do_something(node);
        if (should_skip_children(node)) {
            this.skip();
        }
    },
    leave(node, parent, prop, index) {
        do_something_else(node);
    }
});
```

`svelte.VERSION`

The current version, as set in package.json.

```
const svelte = require('svelte/compiler');
console.log(`running svelte version ${svelte.VERSION}`);
```