SVELTE • REFERENCE

svelte/compiler

ON THIS PAGE

```
import {
  VERSION,
  compile,
  compileModule,
  migrate,
  parse,
  preprocess,
  walk
} from 'svelte/compiler';
```

VERSION

The current version, as set in package.json.

/docs/svelte-compiler#svelte-version

```
const VERSION: string;
```

compile

`compile` converts your `.svelte` source code into a JavaScript module that exports a component

```
function compile(
  source: string,
```

`compileModule` takes your JavaScript source code containing runes, and turns it into a JavaScript module.

```
function compileModule(
  source: string,
  options: ModuleCompileOptions
): CompileResult;
```

migrate

Does a best-effort migration of Svelte code towards using runes, event attributes and render tags. May throw an error if the code is too complex to migrate automatically.

```
function migrate(
  source: string,
  {
    filename,
    use_ts
  }?:
    | {
        filename?: string;
        use_ts?: boolean;
      }
    | undefined
): {
  code: string;
};
```

parse

The parse function parses a component, returning only its abstract syntax tree.

The `modern` option (`false` by default in Svelte 5) makes the parser return a modern AST instead of the legacy AST. `modern` will become `true` by default in Svelte 6, and the option

```
  source: string,
  options: {
    filename?: string;
    modern: true;
  }
): AST.Root;
```

```
function parse(
  source: string,
  options?:
    | {
        filename?: string;
        modern?: false;
      }
    | undefined
): Record<string, any>;
```

preprocess

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.

```
function preprocess(
  source: string,
  preprocessor: PreprocessorGroup | PreprocessorGroup[],
  options?:
    | {
        filename?: string;
      }
    | undefined
): Promise<Processed>;
```

walk

Docs

```
function walk(): never;
```

AST

```
namespace AST {
  export interface BaseNode {
    type: string;
    start: number;
    end: number;
  }

  export interface Fragment {
    type: 'Fragment';
    nodes: Array<
      Text | Tag | ElementLike | Block | Comment
    >;
  }

  export interface Root extends BaseNode {
    type: 'Root';
    /**
     * Inline options provided by `<svelte:options>` — these override options passed to `
     */
    options: SvelteOptions | null;
    fragment: Fragment;
    /** The parsed `<style>` element, if exists */
    css: Css.StyleSheet | null;
    /** The parsed `<script>` element, if exists */
    instance: Script | null;
    /** The parsed `<script module>` element, if exists */
    module: Script | null;
  }

  export interface SvelteOptions {
    // start/end info (needed for warnings and for our Prettier plugin)
    start: number;
    end: number;
    // options
  namespace?: Namespace;
  css?: 'injected';
  customElement?: {
    tag?: string;
    shadow?: 'open' | 'none';
    props?: Record<
      string,
      {
        attribute?: string;
        reflect?: boolean;
        type?:
          | 'Array'
          | 'Boolean'
          | 'Number'
          | 'Object'
          | 'String';
      }
    >;
    /**
     * Is of type
     * ```ts
     * (ceClass: new () => HTMLElement) => new () => HTMLElement
     * ```
     */
    extend?: ArrowFunctionExpression | Identifier;
  };
  attributes: Attribute[];
}

/** Static text */
export interface Text extends BaseNode {
  type: 'Text';
  /** Text with decoded HTML entities */
  data: string;
  /** The original text, with undecoded HTML entities */
  raw: string;
}

/** A (possibly reactive) template expression — `{...}` */
export interface ExpressionTag extends BaseNode {
  type: 'ExpressionTag';
  expression: Expression;
}
```

```
  expression: Expression;
}

/** An HTML comment */
// TODO rename to disambiguate
export interface Comment extends BaseNode {
  type: 'Comment';
  /** the contents of the comment */
  data: string;
}

/** A `{@const ...}` tag */
export interface ConstTag extends BaseNode {
  type: 'ConstTag';
  declaration: VariableDeclaration & {
    declarations: [
      VariableDeclarator & {
        id: Pattern;
        init: Expression;
      }
    ];
  };
}

/** A `{@debug ...}` tag */
export interface DebugTag extends BaseNode {
  type: 'DebugTag';
  identifiers: Identifier[];
}

/** A `{@render foo(...)} tag */
export interface RenderTag extends BaseNode {
  type: 'RenderTag';
  expression:
    | SimpleCallExpression
    | (ChainExpression & {
        expression: SimpleCallExpression;
      });
}

/** An `animate:` directive */
export interface AnimateDirective extends BaseNode {
  type: 'AnimateDirective';
```

```
}

/** A `bind:` directive */
export interface BindDirective extends BaseNode {
  type: 'BindDirective';
  /** The 'x' in `bind:x` */
  name: string;
  /** The y in `bind:x={y}` */
  expression: Identifier | MemberExpression;
}

/** A `class:` directive */
export interface ClassDirective extends BaseNode {
  type: 'ClassDirective';
  /** The 'x' in `class:x` */
  name: 'class';
  /** The 'y' in `class:x={y}`, or the `x` in `class:x` */
  expression: Expression;
}

/** A `let:` directive */
export interface LetDirective extends BaseNode {
  type: 'LetDirective';
  /** The 'x' in `let:x` */
  name: string;
  /** The 'y' in `let:x={y}` */
  expression:
    | null
    | Identifier
    | ArrayExpression
    | ObjectExpression;
}

/** An `on:` directive */
export interface OnDirective extends BaseNode {
  type: 'OnDirective';
  /** The 'x' in `on:x` */
  name: string;
  /** The 'y' in `on:x={y}` */
  expression: null | Expression;
  modifiers: string[];
}
```

Docs

```
  name: string;
  /** The 'y' in `style:x={y}` */
  value:
    | true
    | ExpressionTag
    | Array<ExpressionTag | Text>;
  modifiers: Array<'important'>;
}

// TODO have separate in/out/transition directives
/** A `transition:`, `in:` or `out:` directive */
export interface TransitionDirective extends BaseNode {
  type: 'TransitionDirective';
  /** The 'x' in `transition:x` */
  name: string;
  /** The 'y' in `transition:x={y}` */
  expression: null | Expression;
  modifiers: Array<'local' | 'global'>;
  /** True if this is a `transition:` or `in:` directive */
  intro: boolean;
  /** True if this is a `transition:` or `out:` directive */
  outro: boolean;
}

/** A `use:` directive */
export interface UseDirective extends BaseNode {
  type: 'UseDirective';
  /** The 'x' in `use:x` */
  name: string;
  /** The 'y' in `use:x={y}` */
  expression: null | Expression;
}

interface BaseElement extends BaseNode {
  name: string;
  attributes: Array<
    Attribute | SpreadAttribute | Directive
  >;
  fragment: Fragment;
}

export interface Component extends BaseElement {
  name: 'title';
}

export interface SlotElement extends BaseElement {
  type: 'SlotElement';
  name: 'slot';
}

export interface RegularElement extends BaseElement {
  type: 'RegularElement';
}

export interface SvelteBody extends BaseElement {
  type: 'SvelteBody';
  name: 'svelte:body';
}

export interface SvelteComponent extends BaseElement {
  type: 'SvelteComponent';
  name: 'svelte:component';
  expression: Expression;
}

export interface SvelteDocument extends BaseElement {
  type: 'SvelteDocument';
  name: 'svelte:document';
}

export interface SvelteElement extends BaseElement {
  type: 'SvelteElement';
  name: 'svelte:element';
  tag: Expression;
}

export interface SvelteFragment extends BaseElement {
  type: 'SvelteFragment';
  name: 'svelte:fragment';
}

export interface SvelteHead extends BaseElement {
  type: 'SvelteHead';
  name: 'svelte:head';
  name: 'svelte:options';
}

export interface SvelteSelf extends BaseElement {
  type: 'SvelteSelf';
  name: 'svelte:self';
}

export interface SvelteWindow extends BaseElement {
  type: 'SvelteWindow';
  name: 'svelte:window';
}

/** An `{#each ...}` block */
export interface EachBlock extends BaseNode {
  type: 'EachBlock';
  expression: Expression;
  context: Pattern;
  body: Fragment;
  fallback?: Fragment;
  index?: string;
  key?: Expression;
}

/** An `{#if ...}` block */
export interface IfBlock extends BaseNode {
  type: 'IfBlock';
  elseif: boolean;
  test: Expression;
  consequent: Fragment;
  alternate: Fragment | null;
}

/** An `{#await ...}` block */
export interface AwaitBlock extends BaseNode {
  type: 'AwaitBlock';
  expression: Expression;
  // TODO can/should we move these inside the ThenBlock and CatchBlock?
  /** The resolved value inside the `then` block */
  value: Pattern | null;
  /** The rejection reason inside the `catch` block */
  error: Pattern | null;
  pending: Fragment | null;
  export interface KeyBlock extends BaseNode {
    type: 'KeyBlock';
    expression: Expression;
    fragment: Fragment;
  }

  export interface SnippetBlock extends BaseNode {
    type: 'SnippetBlock';
    expression: Identifier;
    parameters: Pattern[];
    body: Fragment;
  }

  export interface Attribute extends BaseNode {
    type: 'Attribute';
    name: string;
    /**
     * Quoted/string values are represented by an array, even if they contain a single ex
     */
    value:
      | true
      | ExpressionTag
      | Array<Text | ExpressionTag>;
  }

  export interface SpreadAttribute extends BaseNode {
    type: 'SpreadAttribute';
    expression: Expression;
  }

  export interface Script extends BaseNode {
    type: 'Script';
    context: 'default' | 'module';
    content: Program;
    attributes: Attribute[];
  }
}
```

CompileError

Docs

CompileOptions

```
interface CompileOptions extends ModuleCompileOptions {…}
```

```
name?: string;
```

Sets the name of the resulting JavaScript class (though the compiler will rename it if it would otherwise conflict with other variables in scope). If unspecified, will be inferred from `filename`

```
customElement?: boolean;
```

DEFAULT `false`

If `true` , tells the compiler to generate a custom element constructor instead of a regular Svelte component.

```
accessors?: boolean;
```

DEFAULT `false`

DEPRECATED This will have no effect in runes mode

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` .

```
namespace?: Namespace;
```

```
immutable?: boolean;
```

DEFAULT `false`

DEPRECATED This will have no effect in runes mode

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.

```
css?: 'injected' | 'external';
```

`'injected'`: styles will be included in the `head` when using `render(...)`, and injected into the document (if not already present) when the component mounts. For components compiled as custom elements, styles are injected to the shadow root.

`'external'`: the CSS will only be returned in the `css` field of the compilation result. Most Svelte bundler plugins will set this to `'external'` and use the CSS that is statically generated for better performance, as it will result in smaller JavaScript bundles and the output can be served as cacheable `.css` files. This is always `'injected'` when compiling with `customElement` mode.

```
cssHash?: CssHashGetter;
```

DEFAULT `undefined`

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)}`.

```
preserveComments?: boolean;
```

Docs

stripped out.

```
preserveWhitespace?: boolean;
```

DEFAULT `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.

```
runes?: boolean | undefined;
```

DEFAULT `undefined`

Set to `true` to force the compiler into runes mode, even if there are no indications of runes usage. Set to `false` to force the compiler into ignoring runes, even if there are indications of runes usage. Set to `undefined` (the default) to infer runes mode from the component code. Is always `true` for JS/TS modules compiled with Svelte. Will be `true` by default in Svelte 6. Note that setting this to `true` in your `svelte.config.js` will force runes mode for your entire project, including components in `node_modules` , which is likely not what you want. If you're using Vite, consider using dynamicCompileOptions instead.

```
discloseVersion?: boolean;
```

DEFAULT `true`

If `true` , exposes the Svelte major version in the browser by adding it to a `Set` stored in the global `window.__svelte.v` .

```
compatibility?: {…}
```

DEPRECATED Use these only as a temporary solution before migrating your code

DEFAULT `5`

Applies a transformation so that the default export of Svelte files can still be instantiated the same way as in Svelte 4 — as a class when compiling for the browser (as though using `createClassComponent(MyComponent, {...})` from `svelte/legacy`) or as an object with a `.render(...)` method when compiling for the server

```
sourcemap?: object | string;
```

DEFAULT `null`

An initial sourcemap that will be merged into the final output sourcemap. This is usually the preprocessor sourcemap.

```
outputFilename?: string;
```

DEFAULT `null`

Used for your JavaScript sourcemap.

```
cssOutputFilename?: string;
```

DEFAULT `null`

Used for your CSS sourcemap.

```
hmr?: boolean;
```

DEFAULT `false`

If `true`, compiles components with hot reloading support.

Docs

If `true` , returns the modern version of the AST. Will become `true` by default in Svelte 6, and the option will be removed in Svelte 7.

CompileResult

The return value of `compile` from `svelte/compiler`

```
interface CompileResult {…}
```

```
js: {…}
```

The compiled JavaScript

```
code: string;
```

The generated code

```
map: SourceMap;
```

A source map

```
css: null | {
  /** The generated code */
  code: string;
  /** A source map */
  map: SourceMap;
};
```

The compiled CSS

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

```
metadata: {…}
```

Metadata about the compiled component

```
runes: boolean;
```

Whether the file was compiled in runes mode, either because of an explicit option or inferred from usage. For `compileModule` , this is always `true`

```
ast: any;
```

The AST

MarkupPreprocessor

A markup preprocessor that takes a string of code and returns a processed version.

```
type MarkupPreprocessor = (options: {
  /**
   * The whole Svelte file content
   */
  content: string;
  /**
   * The filename of the Svelte file
```

ModuleCompileOptions

```
interface ModuleCompileOptions {…}
```

```
dev?: boolean;
```

DEFAULT `false`

If `true` , causes extra code to be added that will perform runtime checks and provide debugging information during development.

```
generate?: 'client' | 'server' | false;
```

DEFAULT `'client'`

If `"client"` , Svelte emits code designed to run in the browser. If `"server"` , Svelte emits code suitable for server-side rendering. If `false` , nothing is generated. Useful for tooling that is only interested in warnings.

```
filename?: string;
```

Used for debugging hints and sourcemaps. Your bundler plugin will set it automatically.

```
rootDir?: string;
```

DEFAULT `process.cwd() on node-like environments, undefined elsewhere`

Used for ensuring filenames don't leak filesystem information. Your bundler plugin will set it automatically

A function that gets a `Warning` as an argument and returns a boolean. Use this to filter out warnings. Return `true` to keep the warning, `false` to discard it.

Preprocessor

A script/style preprocessor that takes a string of code and returns a processed version.

```
type Preprocessor = (options: {
  /**
   * The script/style tag content
   */
  content: string;
  /**
   * The attributes on the script/style tag
   */
  attributes: Record<string, string | boolean>;
  /**
   * The whole Svelte file content
   */
  markup: string;
  /**
   * The filename of the Svelte file
   */
  filename?: string;
}) => Processed | void | Promise<Processed | void>;
```

PreprocessorGroup

A preprocessor group is a set of preprocessors that are applied to a Svelte file.

```
interface PreprocessorGroup {…}
```

```
name?: string;
```

```
markup?: MarkupPreprocessor;
```

```
style?: Preprocessor;
```

```
script?: Preprocessor;
```

Processed

The result of a preprocessor run. If the preprocessor does not return a result, it is assumed that the code is unchanged.

```
interface Processed {…}
```

```
code: string;
```

The new code

```
map?: string | object;
```

A source map mapping back to the original code

```
dependencies?: string[];
```

A list of additional files to watch for changes

```
attributes?: Record<string, string | boolean>;
```

Only for script/style preprocessors: The updated attributes to set on the tag. If

Warning

```
interface Warning extends ICompileDiagnostic {}
```

Edit this page on GitHub

PREVIOUS
svelte/animate

NEXT
svelte/easing