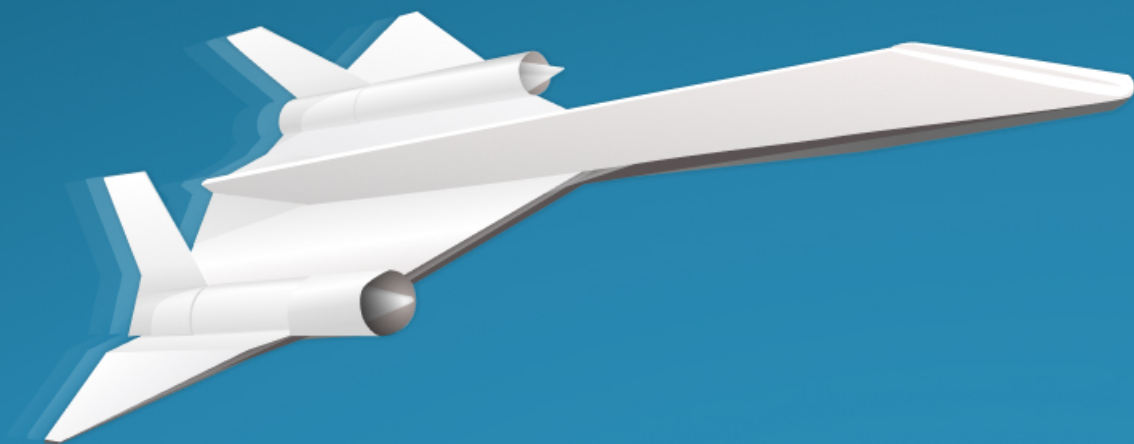




# SVELTE

## A BEGINNER'S GUIDE

BY SIMON HOLTHAUSEN



# Svelte: A Beginner's Guide

Copyright © 2022 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-48-6

- **Author:** Simon Holthausen-Kircher
- **Series Editor:** Oliver Lindberg
- **Product Manager:** Simon Mackie
- **Ignatius Bagus:** Tim Boronczyk
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

## Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of

the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

10-12 Gwynne St, Richmond, VIC, 3121

Australia

Web: [www.sitepoint.com](http://www.sitepoint.com)

Email: [books@sitepoint.com](mailto:books@sitepoint.com)

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

## About the Author

Simon is passionate about web frontends. He has expert knowledge in Angular and Svelte and is also proficient in other frameworks, leading several projects to success. He is also part of the Svelte maintainer team, his most significant contribution being the VS Code extension.

# Preface

## Who Should Read This Book?

This book is for developers with experience of JavaScript. If you've already used a JavaScript framework such as React, you'll find this book an easy read, but it's also suitable for readers with no prior experience of such frameworks.

## Conventions Used

### Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.
</p>
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

### Tips, Notes, and Warnings

#### Hey, You!

Tips provide helpful little pointers.

#### Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

### Make Sure You Always ...

... pay attention to these important points.

### Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

## Supplementary Materials

- <https://www.sitepoint.com/community/> are SitePoint's forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

# Chapter 1: Getting Started with Svelte

Svelte is a relatively new JavaScript frontend framework for developing websites and web apps.

The praise that Svelte has received over the last two years is testament to it not being “just another frontend framework”. It won “breakthrough of the year” on the State of JS survey 2019, followed by topping the satisfaction rating in 2020. It was also voted the most loved web framework in the Stack Overflow 2021 survey.

Svelte appeals to developers with its combination of a small bundle size, very good performance, and ease of use. At the same time, it comes packed with a lot of goodies. A simple state management solution to build upon is already provided, as well as ready-to-use transitions and animations. This introductory tutorial will shed light on how does Svelte achieves this. The following tutorials in the series will go into more detail on how to implement applications with Svelte using the various possibilities Svelte provides.

## The Svelte Backstory

But first, a little back story on Svelte. Though it only entered the mainstream in the early 2020s, Svelte has been around for much longer.

The first commit to GitHub was in late 2016. Its creator is Rich Harris, an open-source wizard whose most prominent other invention is Rollup, a modern bundler. Rich Harris worked at the news magazine *The Guardian* as a graphics editor at the time. His daily routine was to create interactive visualizations for the website, and he wanted to have a tool that easily let him write these without compromising on bundle size or speed. At the same time, he wanted something approachable so other less tech-savvy colleagues would be able to create visualizations fast.

Out of these needs, Svelte was born. Starting from the news room, Svelte quickly gathered a small following in the open-source community. But it wasn't until April 2019 where Svelte really got known to the world. This date marked the release of version 3, which was a complete rewrite with a focus on developer experience and approachability. Since then, Svelte's popularity has risen a lot, more maintainers have joined the team, and Rich Harris has even joined Vercel to work on Svelte full-time.

## Building a Simple Book List

Let's dive into Svelte! We'll build a small book list that allows us to add and remove books from our reading list. The final result will look something like the image below.



We'll start by scaffolding our project from a project template. We'll use the official [Svelte template](#). Alternatives would be to use a [Vite-powered template](#) or to use [SvelteKit](#), a framework on top of Svelte for building full-fledged apps with built-in routing—but we'll keep it as barebones as possible for this tutorial.

After downloading the template, switch to its folder and run `npm install`, which downloads all packages we need to get going. Then we'll switch to `App.svelte`, where we'll replace the contents with an HTML-only version to lay out the visuals we want:

```
<h4>Add Book</h4>
<input type="text" />
<h4>My Books</h4>
<ul>
  <li>A book</li>
</ul>
```



We can write the above code directly at the top level of the Svelte file; we don't need to add any wrapper elements. Svelte's syntax is a superset of HTML, so anything that is valid inside an HTML file is valid inside a Svelte file.

The question now is how to get the dynamic parts in there. We'll start by adding a static list to the script and render that through a loop:

```
<script>
  let books = ['Learning Svelte', 'The Zen of Cooking Tea'];
</script>

<label>
  <h4>Add Book</h4>
  <input type="text" />
</label>
<h4>My Books</h4>
<ul>
  {#each books as book}
    <li>{book}</li>
  {/each}
</ul>
```

We added a `script` tag in which we put our JavaScript logic related to the component. That logic is executed each time the component mounts. We also enhance the HTML with special Svelte syntax to create a loop and print the title of each book. As you can see, Svelte has distinct syntax for control flow blocks, unlike Vue or Angular, which add such functionality in the form of special attributes. This makes the code more readable, as you can more easily spot it. It also makes it unnecessary to create wrapper elements if you want to contain more than one top-level item within the control flow block.

The title of a book is outputted by surrounding the variable with curly braces. In general, whenever you encounter a curly brace within the template, you know you are entering something Svelte-related. We'll look into the template syntax in more detail in Part 2 of this tutorial series.

## Reacting to User Input

We can now render an arbitrary list of book titles, defined by our `books` variable. What about adding a new book? To do this, we need to enhance our

logic in the `<script>` tag and connect it to the `<input>` element:

```
<script>
  let books = ['Learning Svelte', 'The Zen of Cooking Tea'];
  let newBook = '';

  function addBook(evt) {
    if (evt.key === 'Enter') {
      books = [...books, newBook];
      newBook = '';
    }
  }
</script>

<label>
  <h4>Add Book</h4>
  <input type="text" bind:value={newBook} on:keydown={addBook}
/>
</label>
<h4>My Books</h4>
<ul>
  {#each books as book}
    <li>{book}</li>
  {/each}
</ul>
```

We added a new variable called `newBook`, which should mirror the input value. To do that, we bind it to the `<input>` by writing `bind:value={newBook}`. This establishes a two-way binding, so every time the user enters text into the `<input>`, `newBook` updates, and if `newBook` is updated in the `<script>` tag, the display value of `<input>` changes. We could have done the same with simple dynamic attributes, but this way saves us some code—a thought pattern you’ll come across often in Svelte.

When the user presses `enter`, we want to add the new book title to the list. To do this, we add a DOM event listener. To tell Svelte to hook into the event, we just add a colon between `on` and the rest of the event name—so in this case it’s `on:keydown`. After that, we use the curly braces and place the name of the function inside. The function is called each time the event fires off. More on this template syntax can be found in Part 2 of this tutorial series.

The function to call in this case is `addBook`, in which we check the keyboard event, and if the user indeed pressed `enter`, we update the `books` variable. Notice the lack of a `this` context like we find in Angular or Vue 2, or the lack of special value objects like in Vue 3, or the lack of `setState` in React. Svelte doesn't need extra syntax in this case to know that the variable has updated. This might feel like magic, but also like "just plain simple JavaScript" at the same time.

To understand how Svelte achieves this, we need to look under the hood. What does Svelte actually do with a `.svelte` file, and when does it process it? The answer: *Svelte is actually a compiler!* It does most of the work before your code is even loaded in the browser. Svelte parses the code and transforms it into regular JavaScript. During parsing, it's able to see that variables like `newBook` are used in the template, so assignments to it will cause rerenders. The compilation output will therefore wrap these assignments with calls to a `$$invalidate` function, which will schedule a rerender of this exact component for the next browser paint. This is the secret to Svelte's great performance: it knows in advance which parts could trigger rerenders and then only needs to do work in these exact places, surgically updating the DOM. It's also the reason why the bundle sizes of Svelte applications are so small: everything that's not needed just won't be part of the output, so Svelte can leave out every part of its tiny runtime that isn't needed. A Svelte Hello World! app has a bundle size of just 2.5KB!

The only thing to watch out for is that Svelte does only look for assignments. That's why we need to do `books = [...books, newBook];` or `books.push(newBook); books = books;` Otherwise, Svelte wouldn't know that `books` has updated.

```
App.svelte* +
1 <script>
2   let books = ['Learning Svelte', 'The Zen of cooking Tea'];
3   let newBook = '';
4
5   function addBook(evt) {
6     if (evt.which === 13 /* ENTER KEY */) {
7       books = [...books, newBook];
8       newBook = '';
9     }
10  }
11 </script>
12
13 <h4>Add Book</h4>
14 <input type="text" bind:value={newBook} on:keydown={addBook} />
15 <h4>My Books</h4>
16 <ul>
17   {#each books as book}
18     <li>{book}</li>
19   {/each}
20 </ul>

```

Result JS output CSS output

```
151 }
152 }
153
154 function instance($$self, $$props, $$invalidate) {
155   let books = ['Learning Svelte', 'The Zen of cooking Tea'];
156   let newBook = '';
157
158   function addBook(evt) {
159     if (evt.which === 13) {
160       $$invalidate(0, books = [...books, newBook]); /* ENTER KEY */
161       $$invalidate(1, newBook = '');
162     }
163   }
164
165   function input_input_handler() {
166     newBook = this.value;
167     $$invalidate(1, newBook);
168   }
169
170   return [books, newBook, addBook, input_input_handler];
171 }
172

```

## Finishing Touches

We did it! We can now view and add books to our list! It doesn't look that pretty, though, so let's put some finishing touches to our UI. First, we'll add some CSS to style our elements:

```
<!-- script and html code... -->
<style>
  input {
    padding: 5px 10px;
  }
  li {
    list-style: none;
  }
  ul {
    padding: 5px 0;
  }
</style>

```

As you can see, we just add a `<style>` tag to our `.svelte` file and continue to write regular CSS in it. If you're fearing that the code above will style all `<input>`, `<li>` or `<ul>` tags in the entire application, be assured that it won't. Svelte scopes styles by default, so they only apply to the component they're defined in. If you want to define something globally, wrap the selector with the `:global` function. If, for example, you'd like to style all `<input>`s in the application, the code would be `:global(input) { padding: 5px 10px; }`.

The styling is better now. Let's finish it off with a transition for better UX: we want new list elements to fade in. To do that, we just need to reach for one of Svelte's built-in transitions and animations and apply them:

```
<script>
  import { fade } from 'svelte/transition';
  // ..
</script>

<!-- input ... -->
<h4>My Books</h4>
<ul>
  {#each books as book}
    <li transition:fade>{book}</li>
  {/each}
</ul>

<!-- styling ... -->
```

And that's it! Just by importing one of the built-in transitions and applying it by adding `transition:fade` to the element, we get that smooth fade-in transition. Our mini app is now finished. This doesn't contain the topbar and the background gradient yet, but it should be easy now for you to add this as well. This is the end result:

```
<script>
  import { fade } from 'svelte/transition';

  let books = ['Learning Svelte', 'The Zen of Cooking Tea'];
  let newBook = '';

  function addBook(evt) {
    if (evt.key === 'Enter') {
      books = [...books, newBook];
      newBook = '';
    }
  }
</script>

<label>
  <h4>Add Book</h4>
  <input type="text" bind:value={newBook} on:keydown={addBook} />
</label>
<h4>My Books</h4>
```

```
<ul>
  {#each books as book}
    <li transition:fade>{book}</li>
  {/each}
</ul>

<style>
  input {
    padding: 5px 10px;
  }
  li {
    list-style: none;
  }
  ul {
    padding: 5px 0;
  }
</style>
```

## Architectural Considerations

We've seen how to write a little app in Svelte with just 32 lines of code. We've only scratched the surface, of course. A full-blown app needs some kind of state management, multiple components, and ways to integrate these components with each other.

For example, it would make sense to split out the display of one to-do item into a separate component, as we'll add features like editing the name in-place or marking it as done. Having this all in one component would become hard to maintain over time. Luckily, using other components is as easy as importing it as a default import from another Svelte file and interacting with it in a similar way to what we've already seen with regular DOM elements. We'll look into component interaction in more detail in Part 5 of this series.

Another example would be the management of to-dos. Right now, they're handled inside the component and there's no connection to a backend. If we were to add API calls, we would mix UI logic with backend interaction, which is generally better handled outside of components for better separation of concerns. We can use Svelte stores for this, which we'll look at in Part 4.

As you can see, Svelte has solutions to all of our requirements, and we'll look at them over the course of this series.

## Ready, Set ... Svelte?

So, is it safe to use Svelte for your next project? Your manager might ask if Svelte will be around in the years to come or burn out like previous frontend framework stars. There isn't one big company backing Svelte's entire development as there is for Angular and React, but Vue has already shown that this isn't a problem. Moreover, as stated at the beginning, Rich Harris, the creator of Svelte, is now working on it full-time. With Svelte's continuous rise in popularity, there's no sign of it going anywhere in the years to come.

Another aspect of choosing a framework is the ecosystem and its tooling. The ecosystem is still small compared to React, but new libraries are coming out every day, and there are already a handful of very good component libraries. At the same time, since Svelte is so close to vanilla HTML and JavaScript, it's very easy to integrate any existing regular HTML/JavaScript library into your codebase, with no need for wrapper libraries.

Regarding tooling, Svelte is looking pretty good. There's an official VS Code extension that's actively maintained, as well as an underlying language server that can be used by many other IDEs to integrate Intellisense. IntelliJ also has a plugin for Svelte and recently hired the creator behind it to work at JetBrains. There are also various tools for integrating Svelte with various bundlers. And yes, you can also use TypeScript with Svelte.

If you're looking to build a full-blown website or web app, you might also be interested in checking out [SvelteKit](#). It provides a stellar development experience and comes with a flexible filesystem-based router. It also enables you to deploy to many different platforms like Vercel, Netlify, your own Node server, or just a good old static file server, depending on the features and needs of your application.

### Svelte Sumamry

In brief, here are the important points to remember about Svelte:

- it has a full-time maintainer

- it has good tooling
- its features are stable
- its ecosystem is growing
- SvelteKit is available for building apps fast

To summarize: Svelte is definitely ready to use for your next project! If you want to give it a try and you want to learn more, this tutorial series has you covered. In Chapter 2, we'll take a close look at the template syntax. In Chapter 3, we'll look at reactive statements and how they help us react to variable changes or derive computed variables. In Chapter 4, we'll look at stores, which will help us with logic outside and across Svelte files, and which we can also use for state management. In Chapter 5, we'll look at various component interaction concepts. Finally, in Chapter 6, we'll look into testing Svelte apps.

We hope to have sparked your interest in Svelte!



# Chapter 2: Template Syntax

In the first chapter, we got introduced to the history of Svelte and made our first little steps at writing a simple component. In this and the following chapters, we'll have a closer look at specific aspects of writing Svelte code. In the end, you'll be able to create applications in Svelte that scale. We'll start with taking a deep dive into Svelte's template syntax.

Similar to Angular or Vue, Svelte enhances the HTML syntax with several features for expressing dynamic properties, loops, and more. We'll take a look at each of them and see how they help you write readable code. One of the nice things about Svelte is that you can start with “just HTML” and gradually enhance your markup with dynamic features. You do so by using special syntax—called HTMLx in Svelte—that acts as an enhancement of HTML. We'll use this syntax to gradually enhance the following code, which represents the initial draft of a to-do app:

```
<label>
  New To-Do
  <!-- To implement: make this update the list -->
  <input type="text" />
</label>

<!-- To implement: only show this if we don't have a to-do yet -->
<p>What do you want to work on?</p>

<ul>
  <!-- To implement: make this a dynamic list -->
  <li>An item</li>
</ul>
```

The snippet above shows what we want to accomplish with our little app: to be able to enter new to-do items and show them in a list. Other things like marking them as done is out of scope for now, but you are of course free to add them yourself after having finished reading this tutorial.

The final app will look something like the image below.

## New To-Do

Learn Svelte

---

Create great things

---

## Control Flow Syntax

To start, we want to make the list dynamic and render an array of text entries as our to-do list. To achieve that, we'll need to add a `<script>` block to be able to put our list of to-dos in a variable. We've seen in the first chapter that we're able to write regular JavaScript code into the contents of a `<script>` tag, and its variables and functions are then available in the template. We then reach for the `#each` block—one of currently four control flow blocks in Svelte—to loop over the list of to-dos in the template:

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
</script>

<!-- ... -->

<ul>
  <!-- syntax: {#each <arrayname> as <entry>} loop content
  {/each} -->
  {#each todos as todo}
    <li>{todo}</li>
  {/each}
</ul>
```

As you can see, Svelte chooses a different approach to its template syntax compared to Vue or Angular. While the latter two add control flow syntax such as loops in the form of special attributes, Svelte creates distinct syntax for it. This makes the code more readable, as you can more easily spot it. It

also makes it unnecessary to create wrapper elements if you want to contain more than one top-level item within the control flow block.

The text of one to-do item is outputted by surrounding the variable with curly braces. In general, whenever you encounter a curly brace within the template, you know you're entering something Svelte-related.

## Look for Braces!

Any time you're looking through the markup and see braces, you immediately know that you're getting into Svelte's territory.

Let's use another control flow block to solve the second task of showing a message if we have no to-dos yet. We'll use an `#if` block for that:

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
</script>

<!-- ... -->

{#if todos.length === 0}
  <p>What do you want to work on?</p>
{:else}
  <ul>
    {#each todos as todo}
      <li>{todo}</li>
    {/each}
  </ul>
{/if}
```

The `if` block reads very much like a regular JavaScript `if` block. The block starts with `{#if ..}`, and can have multiple `{:else if ..}` branches as well as one final `{:else}` branch. Just as with regular `if` blocks, the latter two are optional. The block is closed by writing `{/if}`. In general, the control flow syntax is `{#<name> ..} .. {:<possibly something>} .. {/<name>}`. The hash marks the start of a block, the colon marks optional other branches where the text behind the colon depends on the control flow block, and the slash marks the end of a block.

`#if` and `#each` are probably what you'll reach out for the most, but there are two more control flow blocks that also come in handy from time to time.

The first one is the `#await` block, which lets you use promises inside the markup:

```
{#await somePromise}
  Loading..
{:then result}
  The result is {result}
{:catch error}
  Oh no! An error occurred: {error}
{/await}
```

As shown above, it's possible to handle all states of a promise, from loading the result to catching errors. Just like we've seen with `#if`, the syntax and semantics closely match that of using promises in regular JavaScript: you `#await` the promise, `{:then ...}` is very similar to the `.then(...)` function of a promise object, and `{:catch ...}` is very similar to the corresponding `.catch(...)` function of a promise object. The branches are all optional, and you can also skip showing the loading branch by doing `{#await somePromise then result}..{/await}`.

```
if (something) {  
  // ...  
} else if (somethingElse) {  
  // ...  
} else {  
  // ...  
}
```

```
{#if something}  
  <!-- ... -->  
{:else if somethingElse}  
  <!-- ... -->  
{:else}  
  <!-- ... -->  
{/if}
```

```
for (const item of array) {  
  // ...  
}
```

```
{#each array as item}  
  <!-- ... -->  
{/each}
```

```
somePromise  
  .then(result => {  
    // ...  
  })  
  .catch(error => {  
    // ...  
  });
```

```
{#await somePromise}  
  <!-- ... -->  
{:then result}  
  <!-- ... -->  
{:catch error}  
  <!-- ... -->  
{/await}
```

The last control flow block is `#key`, which was added more recently. It destroys and recreates everything inside it if the value passed to it changes:

```
{#key someValue}  
  I get destroyed and recreated every time someValue changes  
{/key}
```

This is useful mainly in two scenarios. The first scenario is when you have component where it's easier to recreate its state and contents from scratch rather than updating it—for example, when you wrap a third-party library inside that doesn't provide a good update mechanism. The second scenario is when you want to replay a transition. A transition is a kind of animation that's only triggered when an element is created or destroyed.

Let's get back to our example, which now looks like this:

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
</script>

<label>
  New To-Do
  <!-- To implement: make this update the list -->
  <input type="text" />
</label>

{#if todos.length === 0}
  <p>What do you want to work on?</p>
{:else}
  <ul>
    {#each todos as todo}
      <li>{todo}</li>
    {/each}
  </ul>
{/if}
```

We're now able to differentiate between having an empty list and having todos, and we can render each of them into a list. Awesome!

## Adding New To-dos Using Events and Bindings

What's left is to be able to add new to-do items to the list. When we type into the text box and press `enter`, we want to add that text as a new item to the list. A first version looks like this:

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
  let newTodo = '';
  function handleKeyup(evt) {
    if (evt.key === 'Enter') {
      todos = [...todos, newTodo];
      newTodo = '';
    }
    newTodo = evt.target.value;
  }
</script>
```

```
<label>
  New To-Do
  <input type="text" value={newTodo} on:keyup={handleKeyup} />
</label>

<!-- ... -->
```

There are a few things going on here, let's go through them one by one.

First, we set the `value` attribute of the input by assigning it to the value of `newTodo`. Whenever `newTodo` is updated, the input text will also be updated. If we had given the variable the same name as the attribute, we could have written `{value}`, which is the shorthand of `value={value}`. These shorthands can be used whenever the variable and attribute name are the same.

Next, we listen to the `keyup` event of the input. There are numerous DOM events, all starting with `on`—in this case `onkeyup`. To tell Svelte to hook into the event, we just add a colon between `on` and the rest of the event name—so in this case it's `on:keyup`. After that, we use the curly braces and place the name of the function inside. The function is called each time the event fires off. You could also place an anonymous function in there, if you like.

Inside `handleKeyup`, we then listen to the `keyup` event and act accordingly. We update `newTodo` whenever we enter a character, and when we press `enter`, we add the text to the list of to-dos and then reset the input value. Notice how we didn't need any extra function like `setState` to tell Svelte to rerender in reaction to a variable change. Svelte knows when to rerender by looking for assignment operators, so it knows which variable is dirty. When compiling a Svelte component, it will wrap these assignments with a function call that tells the runtime to rerender on the next browser paint. Be careful, though, as this means that only doing `todos.push(newTodo)` won't work: Svelte will only mark variables dirty that were updated through assignments. Writing `todos.push(newTodo); todos = todos;` would therefore be okay.

The code above works, but we can do a little better. Instead of manually keeping `newTodo` in sync with the input value by listening to the `keyup` event, we can also use a binding. The result looks like this:

```

<script>
  let todos = ['Learn Svelte', 'Create great things'];
  let newTodo = '';
  function handleKeyup(evt) {
    if (evt.key === 'Enter') {
      todos = [...todos, newTodo];
      newTodo = '';
    }
  }
</script>

<label>
  New To-Do
  <input type="text" bind:value={newTodo} on:keyup=
{handleKeyup} />
</label>

<!-- ... -->

```

Notice the `bind:` that we prepended to the `value` attribute. This establishes a two-way-binding between the `value` of the input and the variable `newTodo`. We can change it from the outside, but it's also automatically updated whenever the user types something into the input. Similar to the shorthand for attribute inputs, there's a shorthand if the name of the variable we bind and the attribute we bind it to are the same. So in this case, if the variable name was `value`, we could just write `bind:value`. Svelte allows a limited set of these bindings on DOM elements, mostly around input elements. This makes interacting with forms really easy. It's also possible to bind to properties of components (you'll learn more about component interaction in Part 5 of this series). If you were to write `<Component bind:property />`, this would mean that `Component` would re-render every time you changed `property` from the outside, and it would also propagate back changes to `property` it made on the inside. But beware: too many two-way bindings might get confusing to reason about, so use them with caution.

You may have noticed by now that code inside control blocks or curly braces is regular JavaScript—and indeed, you can use all the JavaScript features you like inside them. This relieves you from learning any extra template syntax aside from what we've seen above.

**It's Just JavaScript!**



Everything inside curly braces and control flow blocks in markup is just JavaScript.

## Wrapping Up

In this second Svelte tutorial, we learned the Svelte template syntax. We learned how to output dynamic content by putting variables in `{brackets}`. We got to know the four control flow blocks, of which `#if` and `#each` are probably what you'll reach for most of the time. We learned how to read and write attributes and how to listen to events. Finally, we learned that we can use two-way bindings in some situations to make our lives easier.

The final code—after adding all the syntax elements we learned about above—looks like this:

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
  let newTodo = '';
  function handleKeyup(evt) {
    if (evt.key === 'Enter') {
      todos = [...todos, newTodo];
      newTodo = '';
    }
  }
</script>

<label>
  New To-Do
  <input type="text" bind:value={newTodo} on:keyup=
{handleKeyup} />
</label>

{#if todos.length === 0}
  <p>What do you want to work on?</p>
{:else}
  <ul>
    {#each todos as todo}
      <li>{todo}</li>
    {/each}
  </ul>
{/if}
```

Notice how close we are to vanilla JavaScript and simple HTML. Svelte doesn't distort but enhances existing code. The following colorization visualizes this. Everything that's blue is regular JavaScript, everything that's green is regular HTML, and everything that's orange is Svelte syntax.

```
<script>
  let todos = ['Learn Svelte', 'Create great things'];
  let newTodo = '';
  function handleKeyup(evt) {
    if (evt.key === 'Enter') {
      todos = [...todos, newTodo];
      newTodo = '';
    }
  }
</script>

<label>
  New Todo
  <input type="text" bind:value={newTodo} on:keyup={handleKeyup} />
</label>

{#if todos.length === 0}
  <p>What do you want to work on?</p>
{:else}
  <ul>
    {#each todos as todo}
      <li>{todo}</li>
    {/each}
  </ul>
{/if}
```

With these tools at your disposal, you're now ready to enrich your HTML with dynamic content and behavior! In the next chapter, we'll look into reactive statements and how they can help us derive computed variables or react to variable changes, and how Svelte again uses its unique advantage—

being a compiler—to make the developer experience of writing these as good as possible.

# Chapter 3: Reactive Statements

So far, we've learned that Svelte is able to compile regular variable assignments inside components into something that will automatically update the view. Whenever you assign something to a variable, Svelte will notice that and schedule a rerender for the next browser paint. We then had a closer look at the component syntax to see how to add dynamic behavior to our HTML code. In this chapter, we'll focus on the contents of the `<script>` tag and take a look at how we can run other code in reaction to a variable change.

Have you ever needed to react to a variable change? Probably every day, in some form or the other. Svelte already reacts nicely to changes of variables by rerendering the view. But what if we need to explicitly react to a variable change inside our `<script>` tag? Doubling a count, fetching a new object from the backend when an ID changes—there are many use cases. This is where reactive statements come in. They look something like this:

```
<script>
  // reactive declaration
  $: foo = bar;
  // reactive statement
  $: {
    something = somethingElse;
  }
</script>
```

Syntactically, the reactive statement is expressed by using the dollar sign followed by a colon—`$.` This is actually valid JavaScript syntax. Originally, it expresses a labelled statement—`labelname : statement`—to which you can tell JavaScript to jump when using `continue` or `break`. But since no one uses that, Svelte has filled the void to syntactically express its reactive statement. This means you can't use labelled statements of the name `$` inside Svelte files, but as you're most likely not using labelled statements of any kind anyway, this won't make a difference.

With the syntax out of the way, let's look at the semantic meaning of reactive statements, which come in two forms: reactive declarations and reactive statements.

## Reactive Declarations

A **reactive declaration** defines a variable whose value is dependent on other variables. A simple reactive declaration may look like this:

```
<script>
  let count = 0;
  $: doubled = count * 2;
</script>

<p>Count: {count}</p>
<p>Doubled: {doubled}</p>
<button on:click={() => count += 1}>Increase count</button>
```

This reactive declaration declares the variable `doubled`, which is expressed as being the double of the variable `count`. Whenever `count` changes, Svelte knows to also compute the new value of `doubled`. Think of reactive declarations like a formula of an Excel cell: you declaratively define what the result is, and Svelte takes care of the rest—but note the caveats that we'll get to later. Reactive declarations can also depend on other reactive declarations, so you could write `$. quadrupled = doubled * 2`.

## Reactive Statements

A **reactive statement** doesn't define a computed variable. Instead, defines a block that should be rerun whenever its dependencies change. Every variable that appears inside the block of a reactive statement and is read (*not* written to)—in order to compute the outcome—is a dependency. A reactive statement looks like this:

```
<script>
  let count = 0;
  let prevCount = 0;
  let countWentUp = false;
  // Simple statement
  $: console.log('Count is' + count);
```

```

// Block statement
$: {
  countWentUp = prevCount < count;
  prevCount = count;
}
</script>

<p>Count: {count}</p>
<p>Count went {countWentUp ? 'up' : 'down'}</p>
<button on:click={() => count += 1}>Increase count</button>
<button on:click={() => count -= 1}>Decrease count</button>

```

As you can see, a reactive statement is essentially an arbitrary bit of code that should be rerun at certain times. In the above example, the `console.log` statement is rerun every time the `count` changes. Inside the block statement, we update two variables in reaction to a `count` change. First, we compare the previous count with the current count to determine whether the count went up or down, and then we store the current count as the new previous count for the next run. You can use reactive statements to express side effects or run computations that don't fit into a simple declaration statement, or which touch more than one variable. Essentially, reactive statements are a superset of reactive declarations: you could rewrite every reactive declaration as a reactive statement. The following code is identical from a semantic point of view:

```

<script>
  let something = 'something';
  // This ...
  $: declaration = something;
  // Is the same as this
  let statement;
  $: {
    statement = something;
  }
</script>

```

## Order of Execution

You might ask now in which order these reactive statements are run. Intuitively, it makes sense that it's somehow ordered in relation to the things they use, but how exactly? The answer is simple: at compile time, Svelte looks at the *direct* code of the reactive statements and checks each variable

that's *read*. These are the dependencies we already touched on briefly in the previous section. From these dependencies, Svelte determines the order of the statements. That means you could write `$: quadrupled = doubled * 2` before writing `$: doubled = count * 2` and Svelte would execute the `doubled` declaration first.

Mathematically speaking, this is called **topological ordering** with respect to the variables. If Svelte isn't able to determine an ordering that way—for example, because the statements have no dependencies between them—they're executed in the order you've written them down. In the following code example, `doubled` and `quadrupled` can be reordered correctly, because Svelte sees the dependency between them. The `console.log` statement at the end stays untouched and runs last, because it has no dependency to `doubled` or `quadrupled` or something that needs to run before they run:

```
<script>
  let count = 0;
  $: quadrupled = doubled * 2; // runs 2nd
  $: doubled = count * 2; // runs 1st
  $: console.log(quadrupled); // runs 3rd
  $: console.log(count); // runs last
</script>
```

There may be situations where we want to hide a variable from the dependencies. In the following somewhat arbitrary (but simple) example, we want `multiplied` to only update when we increase `count`, but not when we change `multiplyBy`. We do this by hiding the variable `multiplyBy` from the direct dependencies, by moving it into a function:

```
<script>
  let count = 0;
  let multiplyBy = 2;

  $: multiplied = multiply(count);

  function multiply(value) {
    return value * multiplyBy;
  }
</script>
```

Now `multiplied` will only change if `count` changes, but not when `multiplyBy` changes. This works because Svelte only looks at the *direct*

code of the reactive statement; it doesn't follow function calls.

The inverse might be true, too: we may want to retrigger a reactive statement even if a variable changes that doesn't directly contribute to the result. In this case, we can just add the variable in question by making it appear within the reactive statement:

```
<script>
  // ...
  $: variableWhichShouldTriggerRecalculation,
  retriggerCalculation();
  // ...
</script>
```

The above shows one possible way to do this by using the comma operator. Another way would be to make it part of the function parameters but never use it in the function itself. How exactly you make the variable appear is up to you. Anything that's valid JavaScript is allowed.

Since reactive statements are regular JavaScript, you can also use `if`-blocks to have more control over when a statement is rerun by only executing its body when a certain condition is met:

```
<script>
  // ...
  $: if (someVariable < 100) {
    // ...
  }
  // ...
</script>
```

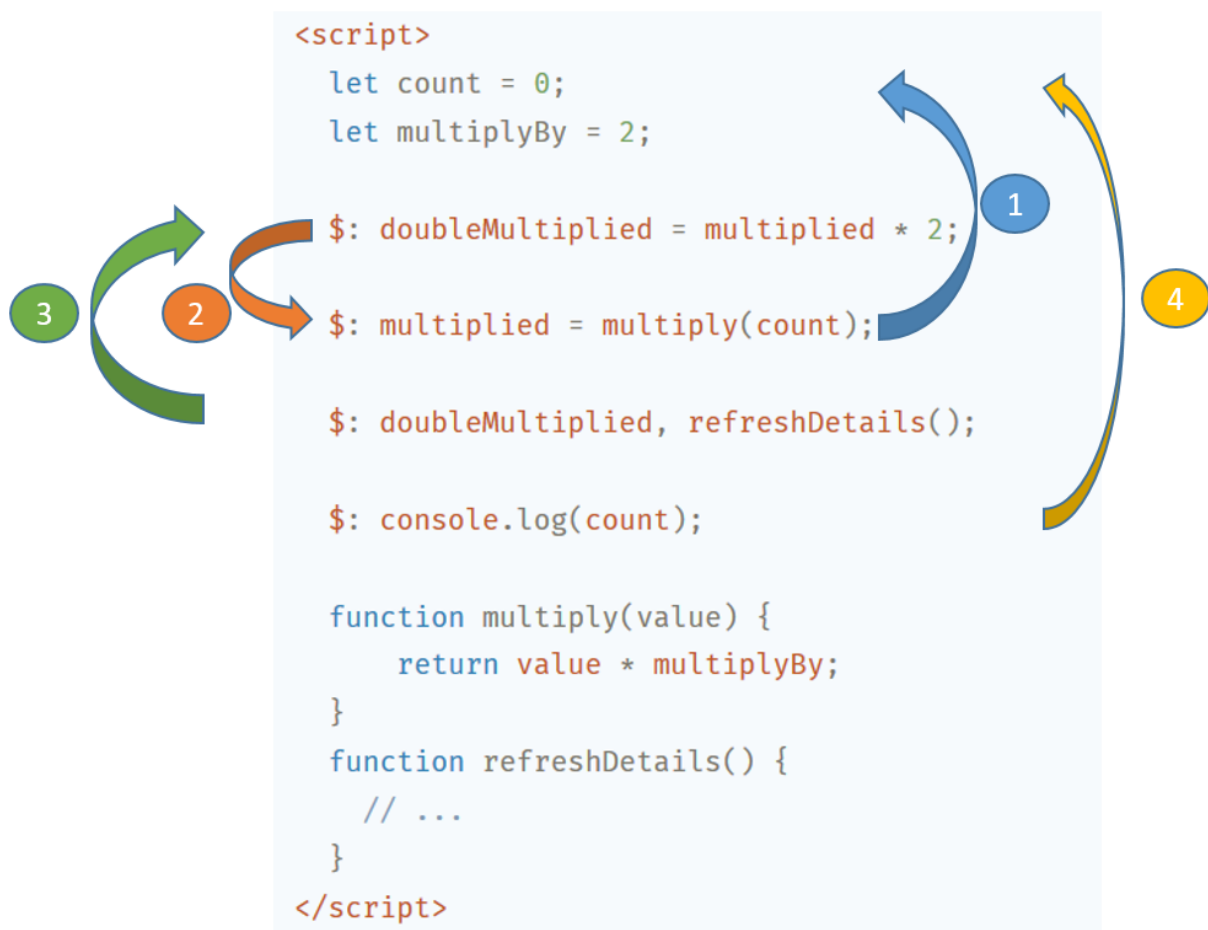
The above code executes whenever `someVariable` or anything inside the reactive statement is updated, but will only go into the body of the statement when the `if` condition is met. You can use this as an alternative to hiding variables from the statement, and also use it to rerun the statement when a variable not used in the body changes by placing it as a dummy condition in the `if` statement.

Most of the time, you don't need to care about the dependencies. This is a relief in comparison to React, where you need to be very explicit about the dependencies in the context of hooks. In Svelte, most of the time it "just



works” like you expect it to, and if you need to fine-tune execution behavior, you can easily adapt your code by using one of the techniques shown above.

Let’s summarize these findings by looking at one more example. The screenshot below shows in which order the reactive statements are run. The arrows show the dependency on other reactive statements. `multiplied` runs first because it depends on `count`. `doubleMultiplied` runs second because it depends on `multiplied`. `refreshDetails()` runs third because it depends on `doubleMultiplied`. `console.log(count)` runs last because it’s written after all other reactive statements and none of the other statements needs it to run first. None of the statements rerun when `multiplyBy` changes because none of them has a direct dependency on it.



## Reactive Statements Are Run Once


One thing that didn't become clear from the previous sections but which might become important is the execution timing of reactive statements. If a regular variable is reassigned, Svelte will schedule an update for the next browser paint. Between this scheduling and the paint is the time where reactive statements are executed—but they're only executed once. This means that, if a reactive statement writes to a variable which a previously run reactive statement is reading, that reactive statement will *not* be rerun. This prevents endless loops, performance issues, and also makes certain reactive statements possible to write. Let's have a look at the reactive statement from above again:

```
<script>
  let count = 0;
  let prevCount = 0;
  let countWentUp = false;
  $: {
    countWentUp = prevCount < count;
    prevCount = count;
  }
</script>
```

This reactive statement reads `prevCount`, but it also *writes* to `prevCount`. If the statement were to run more than once, this would mean it would get invalidated immediately again, resulting in an infinite loop, and would also always assign `false` to `countWentUp` in the end, which defeats its purpose. Just like with the dependencies, most of the time this is nothing to care about, because reactive statements feel very natural to write. But it's good to keep in mind once you get into more complex use cases.

```
<script>
  let count = 0;
  let prevCount = 0;
  let countWentUp = false;

  $: {
    countWentUp = prevCount < count;
    prevCount = count;
  }
</script>
```

The diagram shows two orange arrows indicating dependencies. One arrow points from the expression 'prevCount < count' in the first line of the reactive block to the variable 'count' in the first line of the script. A second arrow points from the same expression to the variable 'prevCount' in the second line of the reactive block. A red circle with a diagonal slash is placed over the second arrow, indicating that this dependency is invalid or problematic.

## Wrapping Up

This tutorial introduced reactive statements. We can use them to react to assignments of other variables to execute side effects or create computed variables. Reactive statements are ordered in Svelte by looking at direct dependencies, and then scheduled to run after a variable assignment. If an order can't be determined, they're run in the order they're declared. We can use this to our advantage to hide variables from the dependencies by moving them into functions or by adding additional variables to the statement to retrigger the execution of that statement. Reactive statements won't be rerun if a reactive statement executed later during the same run updates a variable they depend on. With that in mind, you're now prepared to react to all the things!

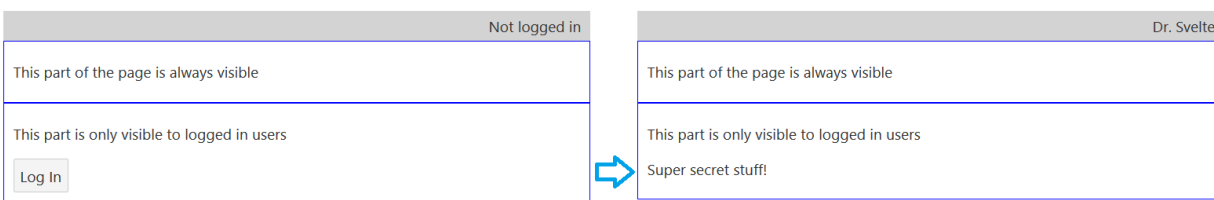
This marks the end of us solely focusing on the insides of one component. In the first chapter, we learned that Svelte is a compiler offering some unique advantages, which we also saw in this chapter, too. In the second chapter, we had a closer look at the component syntax to see how to add dynamic behavior to our HTML code. The next chapters will look at what we can do to integrate components with each other. We'll start by looking at Svelte stores, which provide us with a simple yet powerful API for handling cross-component state in our application.

# Chapter 4: Stores

Svelte's reactivity inside Svelte components is intuitive and easy. Because Svelte is a compiler, it can transform regular variable assignments into something the UI will rerender when it changes (as we saw in Chapter 2). Reactive assignments make derived or computed variables very easy (as we saw in Chapter 3).

Outside of Svelte components, we have a different mental model, there's no inherently connected UI layer to our code, and the Svelte compiler won't touch code outside of Svelte files. But what if we also want the goodies of reactivity outside of Svelte components? What if we have global state that lives inside a JavaScript file that many components should react to when it changes? This is where Svelte stores come in. In this chapter, we'll have a close look at the store API, how to use it to build neatly encapsulated modules, and how to easily use them inside Svelte components.

As a practical example throughout this tutorial, let's pick something that's probably globally available in almost all applications: user state. In this example, this state will consist of whether or not the user is logged in, and, if they are logged in, their username. There's an area of the page that's only visible to authenticated users, and if they're logged in, their username is visible in the top bar at all times.



The app surely won't win us any design awards, but it helps us showcase the problem we have with global state in this application. The root of the application looks like this:

```
<script>
  import Topbar from './Topbar.svelte';
  import AuthOnly from './AuthOnly.svelte';
```

```

</script>

<Topbar />
<div>
  <p>
    This part of the page is always visible
  </p>
</div>
<div>
  <p>
    This part is only visible to logged in users
  </p>
  <AuthOnly />
</div>

<style>
  div {
    border: 1px solid blue;
    padding: 5px 10px
  }
</style>

```

That component uses other components. To do so, we import them as a default import in the `<script>` tag. We can then use them by the same name as the default import in the template. (We'll look into component interaction more deeply in Part 5 of this series.)

`Topbar.svelte` looks like this:

```

<div>
  How to get the username in here?
</div>

<style>
  div {
    background: lightgrey;
    padding: 5px 10px;
    text-align: right;
  }
</style>

```

And `AuthOnly.svelte` looks like this:

```

<script>
  let authenticated = false;
</script>

```

```

{#if authenticated}
  <p>
    Super secret stuff!
  </p>
{:else}
  <button on:click={() => authenticated = true}>Log
In</button>
{/if}

```

The components `Topbar` and `AuthOnly` both need access to the user state; passing it via properties is not an option. In this simple example, we only need to pass the user state down one component, but in reality this will almost never be the case. So we need some other place to share the state. Additionally, `AuthOnly` needs to update the state to log in the user. Where do we put the state now? It's probably best to put it inside a JavaScript file that exports the state so everyone can access it. Let's try that!

## A First Attempt

We create a `user.js` with the following content:

```

export let user = { loggedIn: false };
export function login() {
  user = { loggedIn: true, name: "Dr. Svelte" };
}

```

We also import the `user` state in `Topbar.svelte`:

```

<script>
  import { user } from './user';
</script>

<div>
  {user.name || 'Not logged in'}
</div>

<!-- style... -->

```

We do the same in `AuthOnly.svelte`, where we also call the `login` function:

```

<script>
  import { user, login } from './user';

```

```
    let authenticated = user.loggedIn;
  </script>

  {#if authenticated}
    <p>
      Super secret stuff!
    </p>
  {:else}
    <button on:click={() => login()}>Log In</button>
  {/if}
```

This will correctly show a logged-out state in the beginning, but we somehow can't get the login to propagate back to the components. Why? After all, doesn't this work the same inside Svelte files? The reason is that the Svelte compiler only looks at Svelte files to do its transformation. JavaScript files are left as they are. Furthermore, the semantics inside a JavaScript file are different; there's no direct connection to a view layer. Therefore, we somehow need to explicitly express the state as an object that can change over time, which Svelte components can react to. This is where Svelte stores come in.

## Introducing Writable Stores

Svelte provides a very simple API for initializing and updating a store. A **store** is something that stores (hence the name) a value (also called state) to which others can subscribe, and get notified about updates to the value of that store.

At the heart of Svelte's store API is the `writable` function, which provides this functionality. The `writable` function is called with the initial state and returns an object with `set`, `update` and `subscribe` methods. `set` is called with the new desired state. `update` is called with a function that gets the current state and is expected to return the next state. `subscribe` is called with a callback function that's invoked every time the state changes, and returns a function that can be called to unsubscribe:

```
import { writable } from "svelte/store";

const store = writable("initial value");

const unsubscribe = store.subscribe((value) =>
```



```
    console.log("the current value is " + value)
  );
store.set("new value");
store.update((value) => value + " 2");
unsubscribe();
store.set("another value");
```

In the code snippet above, the value is "initial value" first, then "new value", then "new value 2". All these strings are logged, because the `subscribe` callback function is called each time and logs them out. The `subscribe` function returns an `unsubscribe` function. After it's called, updates to the store—in this case, "another value"—are no longer logged. If you know RxJS, this API may look familiar to you, and indeed, the API closely follows the principle of observables.

Let's apply the `writable` to our use case:

```
import { writable } from "svelte/store";
export let user = writable({ loggedIn: false });
export function login() {
  user.set({ loggedIn: true, name: "Dr. Svelte" });
}
```

We change the code in our JavaScript file and wrap the state with the `writable`. This means we need to adjust our Svelte components to subscribe to the state. Here's a first attempt in `Topbar.svelte`:

```
<script>
  import { user } from './user';
  import { onDestroy } from 'svelte';
  let _user;
  const unsubscribe = user.subscribe(u => _user = u);
  onDestroy(unsubscribe);
</script>

<div>
  {_user.name || 'Not logged in'}
</div>

<!-- style... -->
```

This works! If we press `login` inside `AuthOnly.svelte`, we'll see the name "Dr. Svelte" appear in the top bar. The solution is a little boilerplate-y, though: we have to subscribe to the store, assign it to a local variable, and remember to unsubscribe when the component is destroyed. Can we do better? This is where Svelte's big advantage comes in again: the fact that it's a compiler. Since the compiler will transform Svelte files to JavaScript files anyway, it also can do transformations to get rid of the subscription boilerplate. All we have to do is to put a dollar sign in front of the store and Svelte will take care of the rest:

```
<script>
  import { user } from './user';
</script>

<div>
  {$user.name || 'Not logged in'}
</div>

<!-- style... -->
```

Just like with reactive statements (which we covered in Part 3), the dollar sign is used for some reactive Svelte magic. Declaring (*not* using) variable names with a dollar sign in front is prohibited in Svelte files. Instead, they're a sign to the compiler to generate all the subscription boilerplate for us that we previously wrote by hand. This even works for updating the store. If we were to write `$user.name = "Dr. Dollarsign"` or `$user = { loggedIn: false }` inside a Svelte component, this would be transformed to a store update by the Svelte compiler. Neat! But note that it's not possible to use the `$` syntax to subscribe to stores that aren't created at the top level (imports are top level, so that works). We also can't subscribe to a store that's nested inside an object: we first need to extract it into a top-level variable. These limitations are worth keeping in mind, but you'll probably only rarely encounter them.

```
1 <script>
2   import { user } from './user';
3
4   console.log('current value of ' + $user);
5
6   $user = { loggedIn: false };
7 </script>
8
9
10
11
40
41
42
43
44 }
45
46 function instance($$self, $$props, $$invalidate) {
47   let $user;
48   component_subscribe($$self, user, $$value => $$invalidate(0, $user = $$value));
49   console.log('current value of ' + $user);
50   set_store_value(user, $user = { loggedIn: false }, $user);
51   return [$user];
52 }
53
```

Subscribing to a store through the dollar sign syntax doesn't only work for Svelte stores. Every object that satisfies the store contract (in other words, providing a `subscribe` method) can be used that way. If you prefer to use RxJS, for example, you could use the exact same `$user` syntax to subscribe to the observable. We can use this to our advantage to create our own encapsulated user API. Right now, everyone is able to update the user state directly, because we expose the whole `writable`. To ensure everyone using the user state has to go through the API to update the state, we can create our own store like this:

```
import { writable } from "svelte/store";

let _user = writable({ loggedIn: false });

function login() {
  _user.set({ loggedIn: true, name: "Dr. Svelte" });
}

export const user = {
  subscribe: _user.subscribe,
  login,
};
```

We then need to adjust our `AuthOnly.svelte` component like this:

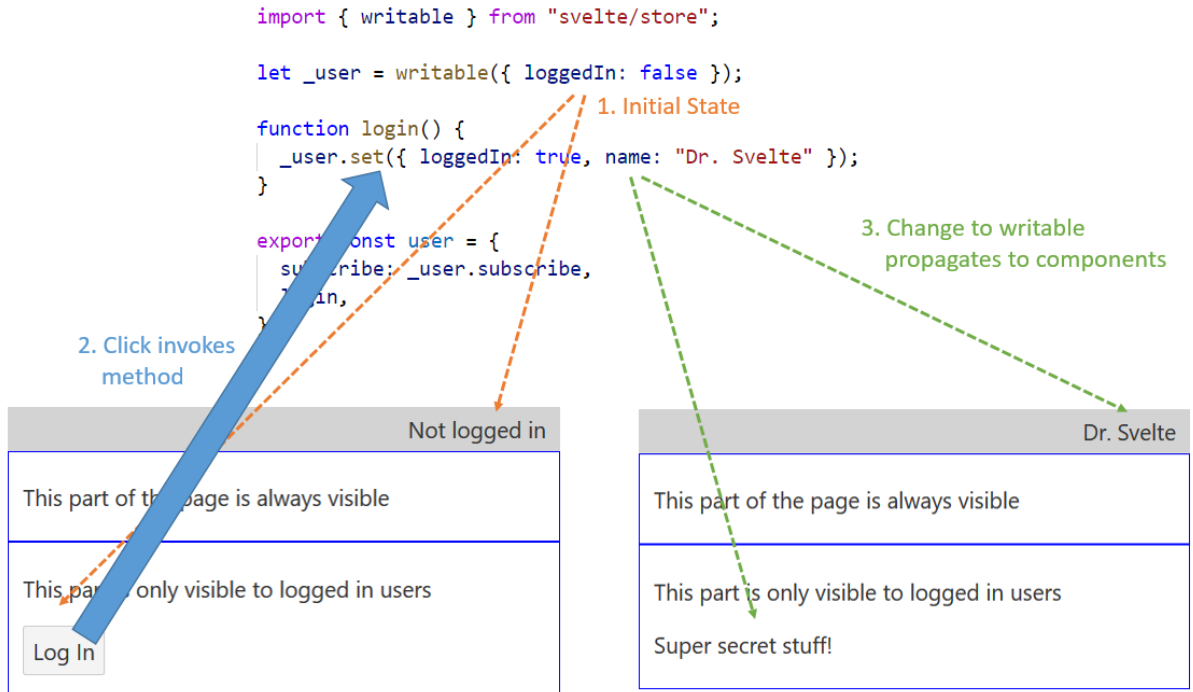
```
<script>
  import { user } from './user';
  $: authenticated = $user.loggedIn;
</script>

{#if authenticated}
  <p>
    Super secret stuff!
  </p>
{:else}
  <button on:click={() => user.login()}>Log In</button>
{/if}
```

We can keep using the `$user` syntax, and update the state through a method on that object. Doing something like `$user.loggedIn = true` or `user.set({ loggedIn: true })` is no longer possible, because the `set` method of the `writable` is no longer exposed. Also note how we can use

`$user` inside the `script` tag and within reactive statements. It's just a variable that we can use anywhere in the component.

The following screenshot visualizes the state flow of our example app.



## Readable Stores

Sometimes we may want to create a store that's not updatable from the outside at all. One use case is to make sure that no one (not even within the same file) updates the store from the outside, because it should be self-contained—for example, a clock that updates every second and that can't be reset or adjusted from the outside. In this case, we can reach for `readable`:

```
import { readable } from "svelte/store";

const time = readable(null, (set) => {
  set(new Date());
});

const interval = setInterval(() => {
  set(new Date());
}, 1000);
```

```
return () => clearInterval(interval);
});
```

The code above creates a store that doesn't have a `set` or `update` method, but only a `subscribe` method. We can't set the state from the outside. The initial state of the `readable` is set through the first parameter; the second parameter is a function that's called every time the subscriber count goes from zero to one. It's handed a `set` function that we can use to update the state from the inside—in this case, the current time each second. We can return a function that's called when the subscriber count goes from one to zero. In this case, we clean up our interval.

## Derived Stores

Once we start using stores, we may also want to combine some of them or just derive a different value from a store. In this case, we can reach for `derived`. Let's suppose our username is split into `surname` and `lastname`, but we want to show the combined name in various places of the app. In order not to repeat ourselves, we use a `derived` store to put them together:

```
import { derived } from "svelte/store";
import { user } from "./user";

export const username = derived(
  user,
  (_user) => _user.surname + " " + _user.lastname
);
```

In its simplest form, `derived` accepts a store or an array of stores as the first argument and a function that gets the current value of this or these stores as the second argument. For more complex use cases, the function also provides a second parameter—`set`—which we can use to update the `derived` state when appropriate. In this case, the return value can be a function that's called when the subscriber count goes from one to zero. We can also pass a third argument to `derived` in this case, which marks the initial value.

Here's an example:

```
import { derived } from "svelte/store";
// ...

export const itemDetails = derived(
  itemId,
  (_itemId, set) => {
    getItemDetails(_itemId).then((details) => set(details));
  },
  "Loading..."
);
```

The code above creates a `derived` store that retrieves details about an item from its ID every time that ID changes, and shows `"Loading..."` initially.

## Wrapping Up

That's it! In this fourth tutorial of the series, we took a deep dive into Svelte stores. We learned that we can use `writable` to make our state available for use across Svelte components, which then can react to updates by just appending a dollar sign to the variable name.

We saw that it doesn't have to be a Svelte store. Instead, every object that implements the subscribe contract can be used like that in a Svelte component. This allows us to use other libraries like RxJS or create custom objects. Svelte stores help us manage state, providing the primitives for a robust state management solution.

We also took a look at `readable` stores, which help us to create state that's not changeable from the outside, and `derived` stores, which provide options for creating a derived state from incoming stores. The core of the Svelte store API is dead simple to understand, but scales for more complex use cases.

You're now ready to create stores to handle state that's shared across different components! In the next chapter, we'll look at more component interactions apart from using stores to create robust applications.

# Chapter 5: Component Interaction Concepts

Components are at the core of frontend frameworks like Svelte. They are the primary unit for organizing and implementing your view layer. Components encapsulate a specific UI and/or behavior—such as the look of a button, a list containing arbitrary elements, or a specific section of a page.

In the first half of this book, we focused on the ins and outs of one component. We looked at template syntax in Chapter 2, and at reactive statements in Chapter 3. In Chapter 4, we looked at Svelte stores, which provide a handy API for implementing state management and which do cross-component updates. There are, of course, more ways to integrate components with each other—and we'll look into them in this tutorial. After reading through this chapter, you'll be able to use the right component API for the right job.

## Using Other Components

The key part to organizing your code is to put each part in the most appropriate location. This means writing and reusing components. We can use another component by importing it as a default import and then use it like a regular element tag inside the template:

```
<script>
  import SomeOtherComponent from
  "../somewhere/else/SomeOtherComponent.svelte";
</script>

<SomeOtherComponent></SomeOtherComponent>
<!-- if the component has no children, you can also selfclose
it: -->
<SomeOtherComponent />
```

## Naming Imports

Since you import the component through its default export, you can name the import any way you want. It doesn't have to be the same as the file name, but it's best practice to do so.

## Passing Stuff to Components through Properties

Many components receive some kind of input that they need in order to function. Consider a list that should show text entries below each other and which you want to use in various places. The list then would be a component property, or in other words an input. In Svelte, these properties are defined through `export let <propertyName>`, as seen here:

```
<script>
  export let items;
</script>

<ul>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ul>
```

At first, this may take some getting used to, since `export` is normally used to make something importable somewhere else, but it will soon feel like second nature.

Properties without an initializer are treated as mandatory. Optional properties have an initializer. Let's say we want to optionally add support for reloading the list through the click of a button, but only if the component user explicitly turns it on. With an optional property, it would look like this:

```
<script>
  export let items;
  export let showReload = false;
</script>

{#if showReload}
  <button>Reload list</button>
{/if}
<ul>
```



```
{#each items as item}
  <li>{item}</li>
{/each}
</ul>
```

The usage would look like this:

```
<script>
  import List from "../List.svelte";
</script>

<List items={['Learn Svelte', 'Create great things']}
  showReload={true} />
```

## Reacting to Component Events

We now have a list with an optional reload button. The logic for reloading the list should be part of the consuming component. We therefore need to somehow react to a click event of the button.

The first option would be to use callback props like you'd use in React—passing a function as input:

```
<script>
  export let items;
  export let showReload = false;
  export let reload = undefined;
</script>

{#if showReload}
  <button on:click={reload}>Reload list</button>
{/if}
<ul>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ul>
```

The second option is to create an event dispatcher and dispatch a custom event from it. For this, we import `createEventDispatcher` from `svelte` and call the resulting function where appropriate:

```

<script>
  import { createEventDispatcher } from "svelte";

  export let items;
  export let showReload = false;

  const dispatch = createEventDispatcher();
</script>

{#if showReload}
  <button on:click={() => dispatch("reload")}>Reload
list</button>
{/if}
<ul>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ul>

```

The first argument to the `dispatch` function is the event name. The second (optional) argument is the payload. Listening to this event is similar to listening to DOM events; we add `on:` in front of the event name:

```

<script>
  import List from "./List.svelte";
  let list = ['Learn Svelte', 'Create great things'];
  function reloadList() {
    // ...
  }
</script>

<List items={list} showReload={true} on:reload={reloadList} />

```

The last option—in this case—is to just bubble the `click` event of the button. An event is bubbled if you write `on:<eventName>` without handling the event:

```

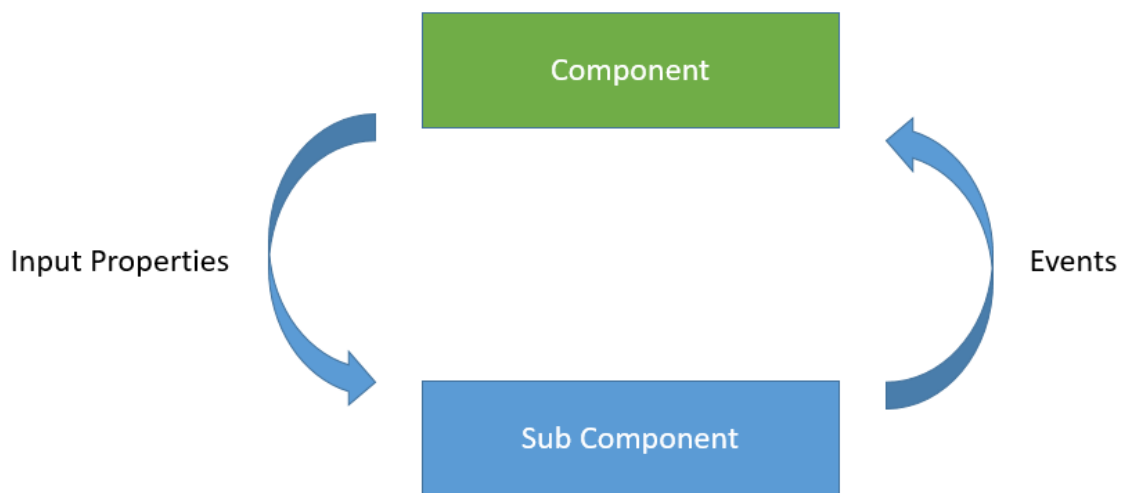
<script>
  export let items;
  export let showReload = false;
  export let reload = undefined;
</script>

{#if showReload}
  <button on:click>Reload list</button>
{/if}

```

```
<ul>
  {#each items as item}
    <li>{item}</li>
  {/each}
</ul>
```

You then would listen to `on:click` on the `List` component in the parent. While this works, in this case it's better to use one of the first two options, as an event name like `reload` better communicates what was clicked and what should happen in response to it. Whether you use callback props or custom events comes down to personal preference.



## Composing the UI with Slots

The list is now displayed and we can reload it, but the look of a list entry is fixed; we can't change it from the outside. To change that, we use slots. A **slot** is like a placeholder where its content is determined by the parent. Let's first define our slot in our list component:

```
<script>
  // ...
</script>

{#if showReload}
  <button on:click>Reload list</button>
{/if}
<ul>
  {#each items as item}
```

```
<li>
  <slot {item}>
    {item}
  </slot>
</li>
{/each}
</ul>
```

The `<slot>` tag marks the section as the destination for the parent's content. The `{item}` property means that the parent is able to use the `item` when providing the UI. Otherwise, we would have no way to output the item's text in the parent. The content inside the `<slot>` is its fallback content, in case the parent component doesn't provide its own UI. The fallback content is optional; you don't need to provide it.

Let's use the component slot in the parent component:

```
<script>
  import List from "../List.svelte";
  // ...
</script>

<List items={list} showReload={true} on:reload={reloadList}
let:item>
  <b>{item}</b>
</List>
```

We chose to simply put the item's text into a `<b>` tag, making it bold. The UI for the slot is inside the component's tag. We can use the `item` property passed into the slot by writing `let:item`. If we wanted to rename the property, we would write `let:item={anotherVariableName}`.

The slot above is the default slot. We can also use named slots to provide more than one of them, which can appear in different areas. Giving the slot a name is as easy as writing `<slot name="desiredName" />`, and using it is just as easy by adding a `slot="desiredName` attribute to the parent's element, which should go into a specific slot. If there was an "empty list placeholder" slot, it could look like this:

```
<!-- ... -->
<List items={list} showReload={true} on:reload={reloadList}
let:item>
  <b>{item}</b>
```

```
<!-- you can use let:xx on the named slot as well if the
component passes slot properties -->
  <p slot="empty">List is empty</p>
</List>
```

The following screenshot visualizes the slot feature.

```
1 <script>
2   import List from './List.svelte';
3   let items = ['I get rendered by the parent using a slot', 'Me too'];
4 </script>
5
6 <List {items} let:item>
7   <div>
8     {item}
9   </div>
10 </List>
11
12 <style>
13   div {
14     border: 2px solid green;
15     padding: 5px;
16     margin: 5px;
17   }
18 </style>
```

I get rendered by the List component

- I get rendered by the parent using a slot
- Me too

## Using the Module Script to Manage Instances of the Same Component

The reload feature of the list is a big success—so big, indeed, that the boss now asks for a “Reload all” button somewhere at the top of the page. For situations like this—where we need to manage and/or coordinate multiple component instances of the same type, or just run code outside of the component lifecycle but colocate it with the component—we can use the `module script` tag. This is another script tag with an additional `context="module"` attribute on it. The code inside this script tag runs only once and right away. Think of it as a separate JavaScript file, but colocated with the Svelte component. Let’s use this to implement the “Reload all” behavior:

```
<script context="module">
  let reloads = new Set();
  export function reloadAll() {
    reloads.forEach((reload) => reload());
  }
</script>
```

```
<script>
  import { createEventDispatcher, onMount } from "svelte";

  export let items;
  export let showReload = false;

  const dispatch = createEventDispatcher();

  onMount(() => {
    const reload = () => dispatch("reload");
    reloads.add(reload);
    return () => reloads.delete(reload);
  });
</script>

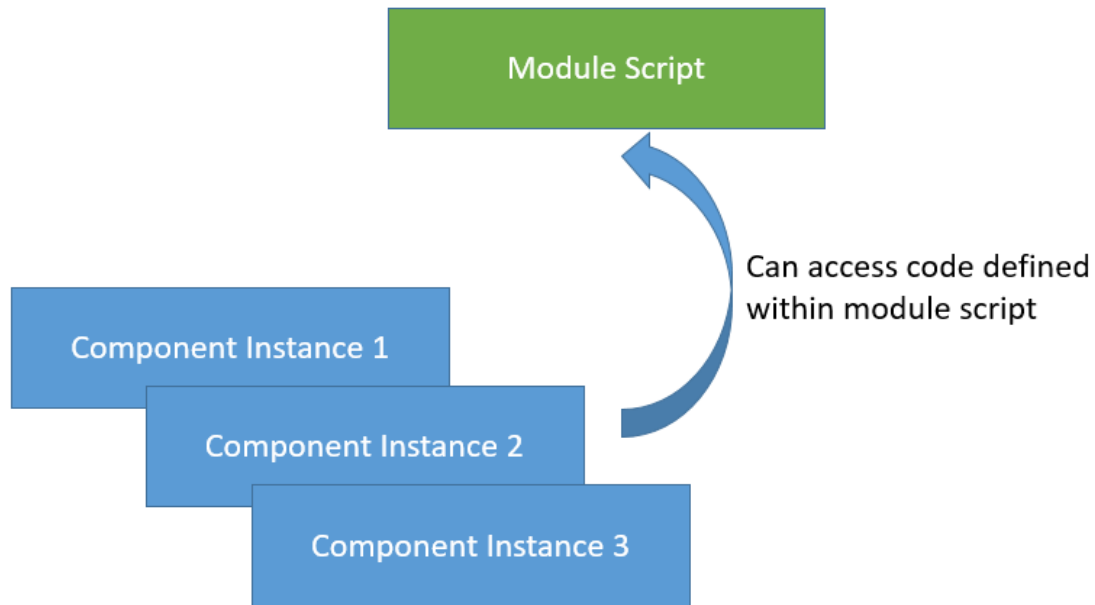
<!-- ... -->
```

We create a `Set` that contains all “reload” `dispatch` functions. Every time a list instance is created, we encapsulate our `dispatch` function and add it to the `Set`. When the component instance is destroyed, we remove it. What’s left to do now is to use the exported `reloadAll` function where the “Reload all” button is implemented:

```
<script>
  import { reloadAll } from "../somewhere/List.svelte";
</script>

<button on:click={reloadAll}>Reload all</button>
```

The default import of a Svelte component is the component constructor itself. Named imports are those that are exported from the module script. That way, we can easily import `reloadAll` and call it when appropriate.



You'll need module scripts only occasionally, but when you do, they'll come in very handy to allow imperative interaction with your component(s) or for managing and coordinating multiple instances of the same type. Another example—from Svelte's own tutorial—deals with having an audio player component rendered multiple times, but only allowing one of them to play at the same time. This is also easily done with module scripts.

## Using Context to Provide State to Component Trees

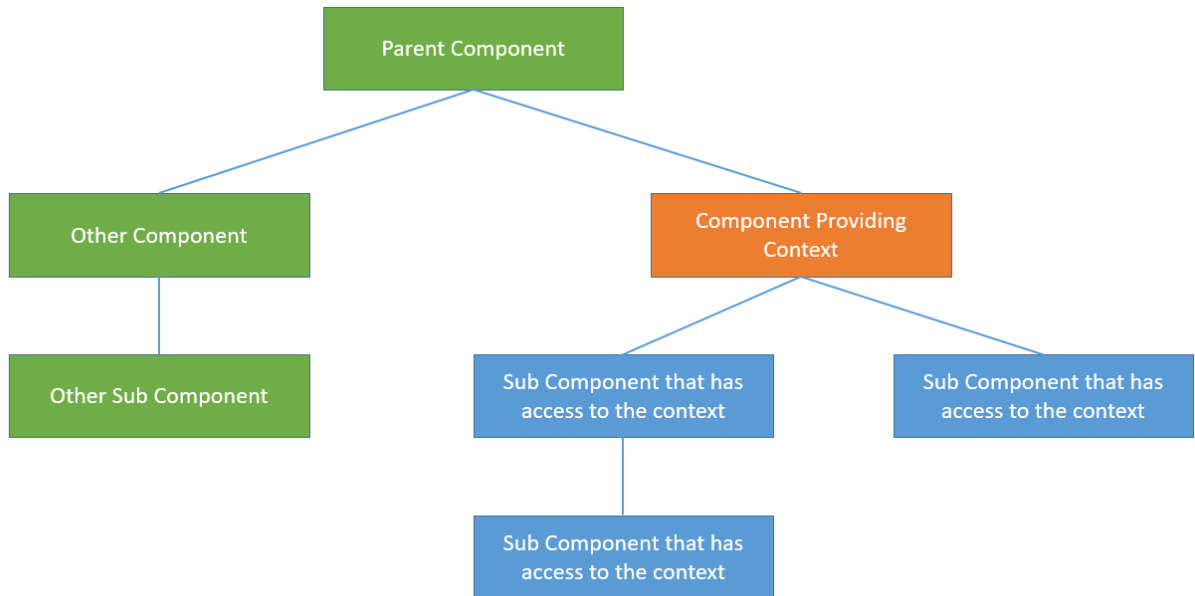
One day after the “Reload all” feature goes live, the boss comes to realize that it's probably not a good idea to actually reload all lists, but rather just specific ones. The calculation whether or not a certain area of the page with lists in it should support reloading is done near the root of these page areas, but the lists themselves are deep children of these components. What now? Should we add a new property to the list component to specify if it should be part of the reload party? This would mean we need to pass down that property through many unrelated components that only need to know about that property in order to pass it down to the next component. This is called **property drilling**. For situations like this, providing context to child component comes in handy. This is what it looks like:

```
<!-- somewhere in a parent component -->
<script>
  import { setContext } from 'svelte';
  setContext('partOfReloadAll',
someFancyComputationThatReturnsABoolean());
  // ...
</script>

<!-- in List.svelte -->
<script>
  import { getContext, onMount } from 'svelte';
  // ...
  const isPartOfReloadAll = getContext('partOfReloadAll');
  onMount(() => {
    if (!isPartOfReloadAll) {
      return;
    }
    const reload = () => dispatch("reload");
    reloads.add(reload);
    return () => reloads.delete(reload);
  });
});
```

These two functions—`setContext` and `getContext`—allow us to provide specific things to all children of the component where that thing is provided. This is similar to `useContext` in React, `provide/inject` in Vue, or a service provided on a component in Angular. Note that you need to call these functions during component initialization. You can't call them later on in the life cycle or asynchronously.





In the example above, we passed a string as the key, but you could also use a `Symbol` or something else that's guaranteed to be unique. The value can be anything you want. In this case, it's a simple Boolean, but it could also be a function or a store. The combination of context and stores is a nice way of providing a value that can change over time and to which the children should be able to react. Here's an example:

```

<!-- somewhere in a parent component -->
<script>
  import { setContext } from 'svelte';
  import { writable } from 'svelte/store';
  const count = writable(1);
  setContext('count', count);
  // ...
  function updateCount() {
    count.update(c => c + 1);
  }
</script>

<!-- somewhere in a child component -->
<script>
  import { getContext } from 'svelte';
  const count = getContext('count');
</script>
<p>The count is {$count}</p>

```

If you have state that's available globally, you can of course skip using context and instead just import that state from a central location. This is what we did in the previous part of the application using stores.

## Wrapping Up

This was a long one! We've seen quite a few component interaction concepts in this tutorial. To summarize:

- Use component properties and events to implement basic component interaction.
- Use slots when you need to provide a custom UI from the parent at a certain place in the child.
- Use the `<script context="module">` tag when you need to manage or coordinate multiple instances of the same component or export imperative logic related to them.
- Use `setContext` and `getContext` when you need to provide certain values for a certain sub part of the app and you want to avoid property drilling.

With all these tools at your disposal, you're now ready to structure and organize your components in the best way possible.

You're now ready to write full-blown applications in Svelte. In the last chapter, we'll look into testing these applications. We'll look into the different levels at which we can test, as well as what it means to make your apps testable.

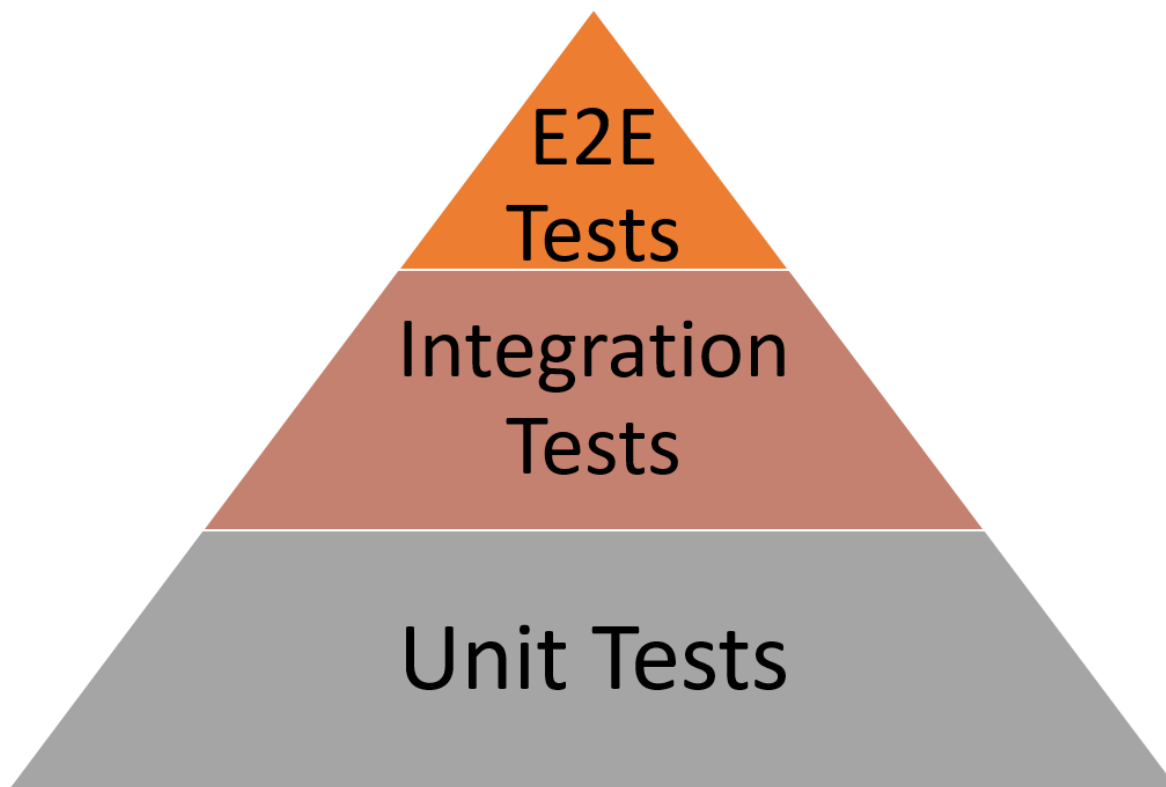
# Chapter 6: Testing Applications

We've come quite a long way. We started by looking at Svelte's history, as well as the first steps of writing an application with it. The tools for building such an application were then introduced in more detail. We took a deep dive into the template syntax of Svelte, which is an extension of HTML. We saw how to use reactive statements to react to variable changes—for example, in order to create computed variables. We then looked at Svelte stores, which can be used to share state across multiple components, and we learned about various ways to integrate components with each other.

All this empowers us to create Svelte applications that scale. But how do we test those applications? This is what we'll look at in this final chapter.

## The Testing Pyramid

Before we jump into the technical details, let's first take a step back and think about the ways we can test an application. As it turns out, there's not one true way to do this. Rather, there are different layers of testing. There's actually a name for this, which you may have come across already: “the testing pyramid”.



The **testing pyramid** helps us classify our various ways of testing.

At the bottom, there are **unit tests**, which look only at a tiny part of our application at a time—such as testing a specific function in isolation.

Above these are **integration tests**, which test a small to medium part of our application in combination. While unit tests are great for testing things like whether or not a function returns the right thing, they don't tell you whether that function is actually used in the right way. All your unit tests could pass, but the interaction of the parts with each other you tested might be wrong. Integration tests help surface these bugs.

The top-most layer in the pyramid are **end-to-end tests** (abbreviated with E2E), which take the integration tests one step further by not only running parts of the application, but also running the whole application and testing through the eyes of the end user. The challenge is to get the whole application running under test conditions. The advantage is that you need to mock less: only the backend, or if you want, even that can run as part of an E2E test.

It's named the "testing pyramid" and not just "testing levels" because the idea is that there should be more tests at the lower level than the higher level. Some people say you should strive for 60% unit tests, 30% integration tests and 10% E2E tests. There's no single right way to do this, however, and for testing the frontend especially there are people who even challenge the pyramid completely, saying that E2E tests are the easiest and most robust way—especially with respect to implementation changes—to test web apps, so you should focus on these the most.

In the end, how to organize your tests is up to you, and we'll focus on giving an example for each layer in the rest of this tutorial.

## Test Setup

We'll be using [Jest](#) for unit and integration tests. Jest is currently the most popular testing library for frontend code. If you're using SvelteKit, you can set up and install all the required dependencies for testing with Jest via [svelte-add-jest](#), by running the following in the root of your SvelteKit project:

```
npx apply rossyman/svelte-add-jest
npm install @testing-library/svelte -D
```

The first command does all the installation of the minimum required dependencies for you, as well as creating the required config. We only need to install one more package to use after that.

If you aren't using SvelteKit, there are some more manual steps to take. First, install all the required dependencies. If you're using JavaScript, these are as follows:

```
npm install jest svelte-jester @babel/core @babel/preset-env
babel-jest @testing-library/svelte -D
```

`jest` is the core testing framework and `svelte-jester` is the corresponding plugin in order to support Svelte. Babel is needed in order to transform newer ES module code. The `testing-library` package will help us write more robust tests, as we'll see later on. Then create a `jest.config.js` with the following contents in order to tell Jest to handle Svelte files correctly:

```
module.exports = {
  testEnvironment: "jsdom",
  transform: {
    "^.+\\.svelte$": "svelte-jester",
  },
  moduleFileExtensions: ["js", "svelte"],
};
```

Then create a `.babelrc` and add the following code, which is needed for transpiling more recent JavaScript syntax into something that Jest understands:

```
{
  "presets": [["@babel/preset-env", { "targets": { "node": "current" } }]]
}
```

These are the steps for creating the test setup when using JavaScript. If you're using TypeScript, the setup looks a little different. First, you need to install the following packages:

```
npm install typescript svelte-preprocess jest ts-jest svelte-jester @testing-library/svelte -D`
```

You probably already have installed `typescript` and `svelte-preprocess`, as these are needed for developing Svelte applications in TypeScript. The Babel dependencies are replaced with `ts-jest`, which takes care of transpiling the code correctly. After that, add the following config files to the root of your project:

In `svelte.config.js`, add this:

```
const sveltePreprocess = require("svelte-preprocess");

module.exports = {
  preprocess: sveltePreprocess(),
};
```

In `jest.config.js`, add this:

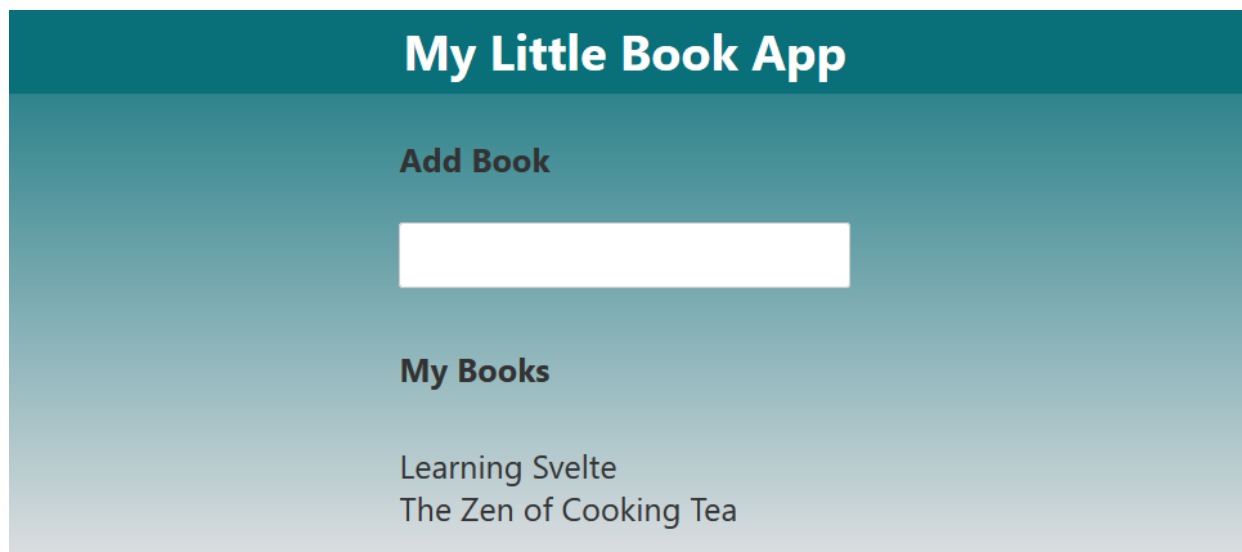
```
module.exports = {
  testEnvironment: "jsdom",
  transform: {
```

```
    "^.+\\.svelte$": [  
      "svelte-jester",  
      {  
        preprocess: true,  
      },  
    ],  
    "^.+\\.ts$": "ts-jest",  
  },  
  moduleFileExtensions: ["js", "ts", "svelte"],  
};
```

These files ensure that Jest knows how to preprocess TypeScript and Svelte files correctly before running the tests. The `svelte.config.js` tells various tools how to preprocess Svelte files before handing them off to the compiler, and the Jest config tells Jest how to transform Svelte and TypeScript files. Again, if you're already using TypeScript, you already might have a `svelte.config.js`.

## Writing Your First Test

With all that setup out of the way, let's get into writing our first test! The application we'll be testing is our very simple book list you already know from Chapter 1. As a reminder, the image below shows what it looks like.



And this is the code behind it:

```
<script>
  import { fade } from 'svelte/transition';

  let books = ['Learning Svelte', 'The Zen of cooking Tea'];
  let newBook = '';

  function addBook(evt) {
    if (evt.key === 'Enter') {
      books = [...books, newBook];
      newBook = '';
    }
  }
</script>

<h2>
  My Little Book App
</h2>
<div>
  <label>
    <h4>Add Book</h4>
    <input type="text" bind:value={newBook} on:keydown=
{addBook} />
  </label>

  <h4>My Books</h4>
  <ul>
    {#each books as book}
      <li transition:fade>{book}</li>
    {/each}
  </ul>
</div>

<style>
  div {
    display: inline-block;
    margin-left: auto;
    margin-right: auto;
  }
  h2 {
    background: rgb(9,111,121);
    padding: 5px 10px;
    text-align: center;
    color:white;
    font-weight: bold;
    margin:0;
  }
  input {
    padding: 5px 10px;
```



```
}
li {
  list-style: none;
}
ul {
  padding: 5px 0;
}
:global(body) {
  background: rgb(9,111,121);
  background: linear-gradient(180deg, rgba(9,111,121,1)
0%, rgba(255,241,242,1) 60%, rgba(255,241,242,1) 100%);
  display: flex;
  flex-direction: column;
}
</style>
```

We want to test that adding a book works correctly. This means ensuring that the typed text appears as an item in the list, and that the input is cleared after adding the book.

How to approach this? The first idea might be to somehow access the compiled version of the component and then update and invoke the needed variables and functions. This is very brittle, however, since we're closely tied to the implementation details of our component. It would be better if we could test this one abstraction level higher, through the rendered HTML. This means that we simulate interactions with the HTML and then check if it was updated correctly.

In our case, this means we want to simulate typing into the input, press enter, and then check the updated HTML for a new book entry. `@testing-library/svelte` will help us with this. It's a wrapper around the `Testing Library` API, which provides an opinionated set of utilities for interacting with the HTML. If you already have written tests with other wrappers around this library—for example, using `React` or `Vue`—the following code snippet will look familiar to you:

```
import { render, fireEvent } from "@testing-library/svelte";
import App from "../App.svelte";

describe("Book App", () => {
  test("can add book", async () => {
    // Instantiate and render component
    const app = render(App);
```

```
// Check that there are no list items at first
expect(() => app.getAllByRole("listitem")).toThrow();
// Simulate user input
const input = await app.findByLabelText("Add Book");
await fireEvent.input(input, {
  target: { value: "Writing Svelte Tests" },
});
await fireEvent.keyDown(input, { key: "Enter" });
// Check that there is one list item now
expect(app.getAllByRole("listitem").length).toBe(1);
expect(app.getByText("Writing Svelte Tests")).toBeDefined();
});
});
```

We first instantiate the component using the provided utility function named `render`. After checking that the list is empty initially, we simulate user input, and then test that the new book was added after pressing the `enter` key. These methods return promises that resolve as soon as the Svelte compiler has rerendered, which happens asynchronously. As you can see, we don't do that by looking up specific HTML tags or classes (though we could fall back to that if needed). Instead, we've chosen a more semantic approach and access the HTML by looking up (for example) specific text or a specific role. This makes our tests more robust to changes in the Svelte template.

The example above is a test for a rather simple component. Testing can get more complex when the component in question is connected to context or global state or has many components below it. If you only want to test that specific component, you need to mock away all unwanted dependencies, which can be easy to hard depending on what needs to be mocked.

If it's very hard to mock things away or to test in general, this hints at your code maybe being too wired with all its dependencies. You may need to split up and modularize or reorganize some of the code. For example, if you need to create a complex test setup because you want to test specific business logic, it may be beneficial to split the computation out into a function in a separate JavaScript file that's independent of components, which makes testing that function much easier.

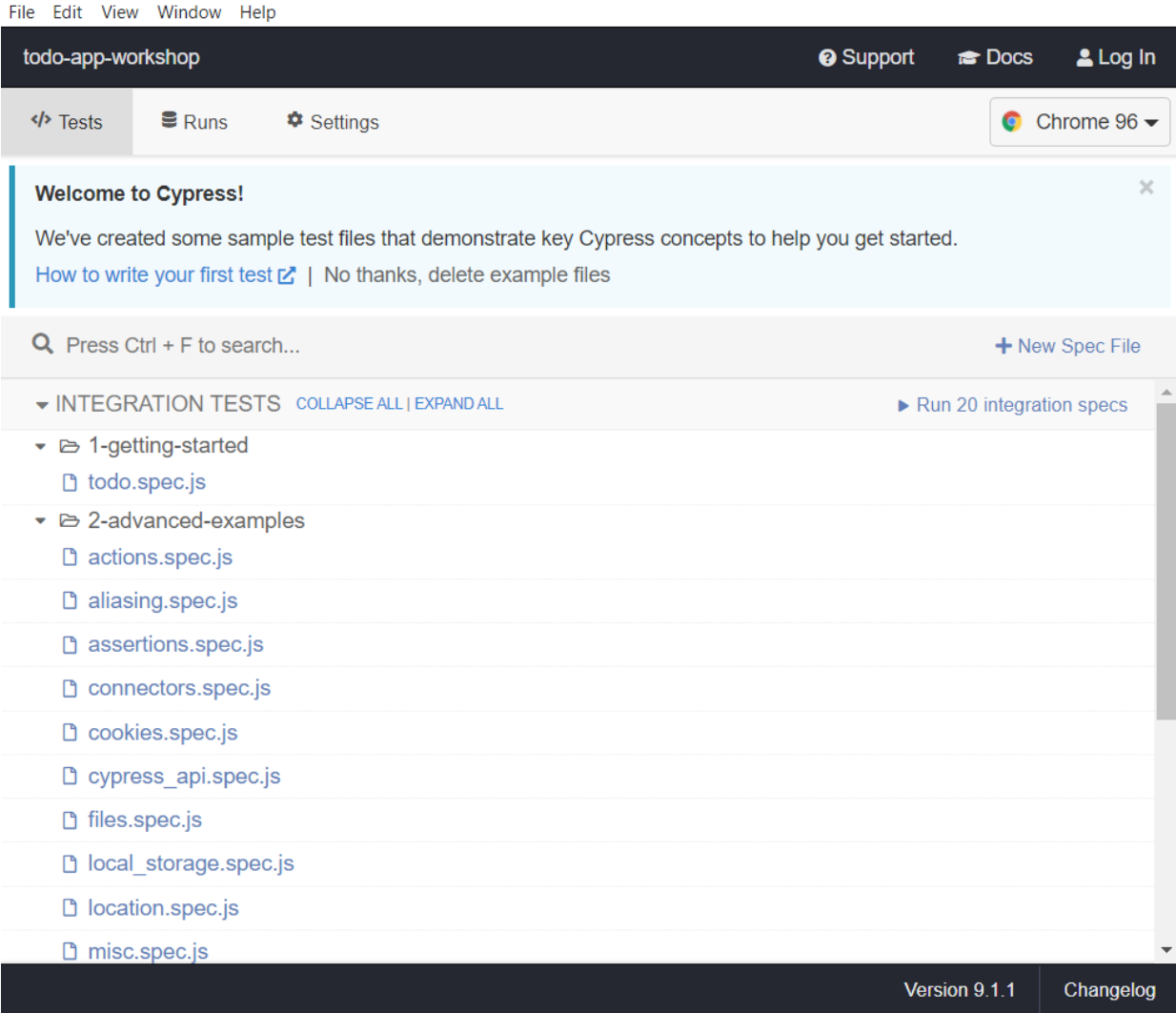
Preparing your code for better testability will often lead to cleaner code overall, so a feel-good vibe when testing will mostly go hand in hand with a well-structured app. If it doesn't, it might be that the code is just hard to test

because it's very high up the component tree or because the logic you want to test inherently needs many dependencies. In this case, it's often easier to use E2E testing tools, which we'll look at next.

## Writing your First E2E Test

We'll write our E2E tests using [Cypress](#). Alternatives would be [Selenium](#) or [Playwright](#), the most recent addition by Microsoft.

Fortunately, the Cypress setup is rather easy. Just run `npm install cypress -D` and wait a few minutes until everything is set up. This will create a new `cypress` folder at the root of your project, which contains a configuration file we don't need to look at for now, as well as an example test suite. Next, run `npx cypress open` and wait for Cypress to start up. You'll be greeted with the screen pictured below, and you can look around and run the example tests if you like.



If you want, you can delete the example tests; we'll be creating our own test now. Cypress expects the application to run at a specific location, so we'll start up our dev server in a separate terminal and then access the app—for example, at `http://localhost:3000`. We now generate a new `.spec.js` file within `cypress/integration` and proceed to write the same test we just wrote using Jest, this time using Cypress:

```
/// <reference types="cypress" />

describe("Books App", () => {
  beforeEach(() => {
    // Open app
    cy.visit("http://localhost:3000");
  });
});
```

```
it("can add book", () => {
  // Check that there are no list items at first
  cy.get("li").should("have.length", 0);
  // Simulate user input
  cy.get("input").type("Writing Svelte Tests{enter}");
  // Check that there is one list item now
  cy.get("li")
    .should("have.length", 1)
    .last()
    .should("contain.text", "Writing Svelte Tests");
});
});
```

As you can see, the test reads quite similarly to the Jest test we wrote earlier. We open the app first, check if there are no items in the list, then add one through user input simulation, and then check that there's the expected item.

Writing the test is more focussed around the HTML tags right now, which you might find better or worse in comparison to the Jest tests above, depending on your preference. If you want to use the same style of retrieving the HTML elements, the Testing Library has you covered. Just install `@testing-library/cypress` and you'll be able to use the same queries you already know!

Regardless of how you decide to proceed, you can see that the abstraction level is much higher when writing E2E tests. You don't mock anything within your app. In fact, you run it like you normally would during development. If you want, you can mock backend calls. Cypress has handy utilities for making this possible—or you just set up your own simple mock server. Because this frees you from many mocking headaches, some people prefer to write more tests this way. This is especially helpful if you plan on rewriting parts of your app and want to ensure the behavior stays the same.

## Wrapping Up

That's all for testing Svelte apps right now! We saw how to create a proper test setup using Jest for unit and integration tests and Cypress for E2E tests, and we wrote simple tests using each of them. We learned about the testing pyramid, which showcases the different levels of abstraction when testing, and we saw the difference firsthand using the two testing tools. We also

learned that good tests and good code often influence each other, and that tests that are brittle and/or hard to write may hint at your code needing some refactoring.

This is the end of the book! We've come quite a long way. We learned the ins and outs of template syntax and how to add dynamic behavior to our HTML. We learned about reactive statements that help us compute derived values or react to other variables changes, and how easy it is to write them. We saw the same when using Svelte stores, how easy it is to subscribe to them, and how to create a robust state management solution built upon them. We also learned about the different component interaction concepts that help you organize and scale your app. Finally, in this chapter, we saw how to test Svelte apps.

With all this new information, you're now ready to build Svelte apps that scale. Good luck, and have fun!