SolidJS
The Complete Guide

A Comphensive Guide to Reactive Web Development With SolidJS and Typescript

Sinan Polat

SolidJS: The Complete Guide

A comprehensive guide to reactive web development with SolidJS and TypeScript

Sinan Polat

SolidJS: The Complete Guide by Sinan Polat

Copyright © 2025 Sinan Polat. All rights reserved.

October 2025: First Edition, V8

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 by the author, except in the case of brief quotations embedded in critical articles or reviews.

The effort has been made to ensure the accuracy of the information and instructions presented. However, the information contained in this work is sold without warranty, either express or implied. Neither the author, nor its dealers or distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this work. Use of the information and instructions contained in this work is at your own risk.

If any code samples or other technology this work contains or describes is subject to the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Table of Contents

CHAPTER 1

Introduction

This is a comprehensive book that aims to teach you the ins and outs of Solid, covering its core principles, the inner workings, and the API. By the end of this book, you will have a thorough understanding of Solid to write efficient applications.

Solid is a lightweight JavaScript library for building applications that can run on both the client and server side. It can be used as a standalone library or alongside other libraries, as it is designed to be small and efficient.

Solid does not introduce any novel approaches to frontend development, but instead borrows the best ideas from other battle-tested libraries, including KnockoutJS, React, Vue, and Marko. It is built on proven concepts and ideas, making it a pleasure to use.

Solid has a relatively small API surface; its core library exposes only a handful of items. However, the intricacies of reactivity and its implementation involve complex interactions between its parts, which makes it really hard to explain some of the concepts without a lengthy discussion. While I've avoided repetitions as much as possible, occasional reminders were necessary to present the topic in a complete and coherent way, eliminating the need to go back and forth between chapters to understand the concept at hand.

When there are multiple ways of doing something, for instance, accessing underlying DOM elements, we discuss the pros and cons of each approach, and provide tips on the best practices when appropriate.

Like any other library, Solid has its own quirks that might leave you puzzled. I have tried to shed light on the root causes of those quirks, rather than merely mentioning them, and included callouts to help you steer clear of probable pitfalls around them, if there are any.

This book is based on Solid v1.8, but rest assured, the concepts and principles we explore aren't closely tied to any single version. Even as Solid continues to evolve, the core ideas and foundational logic will stay consistent, so you'll find lasting value in these pages regardless of version updates. I hope what you learn here will keep serving you well.

Code Examples

You can download supplemental materials—including code examples and exercises—from the official GitHub repository:

https://github.com/solid-courses/solidjs-the-complete-guide

If you run into technical issues or have questions about the examples, please start a discussion or open an issue on the repository.

In general, you are free to use the example code provided with this book in your own programs and documentation. Permission is not required unless you intend to reproduce a substantial portion of the code. For example, you may use several snippets in your programs without restriction, and you may also cite the book or quote short examples when answering a question. However, selling or redistributing the examples requires prior permission, as does incorporating a significant amount of code into your product's documentation.

Contact and Feedback

As we journey together through the contents of this book, your insights and experiences will be invaluable. Even with our best efforts, we understand that there may be areas that could be improved, clarified, or corrected. Whether it's a typo, a conceptual error, or a suggestion for improvement, we welcome your feedback.

To make the process as smooth as possible, we've created a dedicated repository for you to share your thoughts, criticisms, and suggestions. Please visit https://github.com/solid-courses/solidjs-the-complete-guide to submit your feedback. You can open a new issue to detail your findings or suggestions.

Your feedback is crucial not only to improve this book but also to enhance the learning experience of future readers. By contributing your insights, you'll help create a more accurate, comprehensive, and user-friendly resource for anyone interested in Solid. We greatly appreciate your time and effort in helping us achieve this goal.

Requirements

This book requires basic knowledge of JavaScript, HTML, and CSS. We won't delve into language-related concepts, maybe briefly touch upon a few of them when there is a need for them.

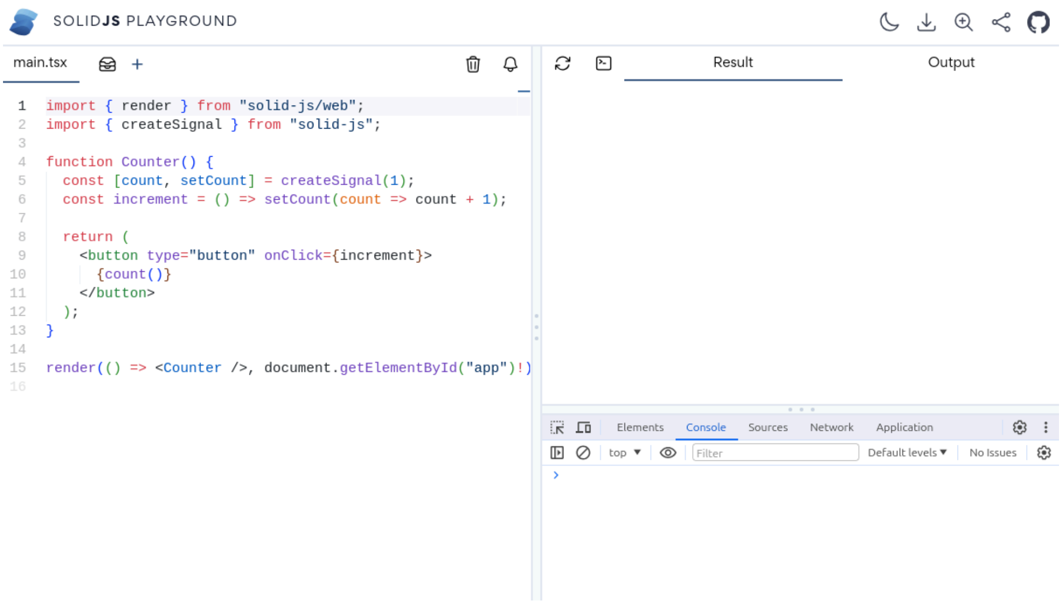
Examples are written in TypeScript. However, even if you have never used TypeScript before, you should be able to understand them, as an explanation will accompany any code snippet that is too complex or requires a certain TypeScript feature to be turned on.

If you don't want to use TypeScript at all, you need to set up your development environment accordingly. There is a pre-built template from the core library for using JavaScript only. Ignoring types will be enough to make the code examples work.

We will need Node.js for both building and running the code examples. Node.js is a JavaScript runtime environment. There are plenty of resources on the Internet on how to install and run Node.js. We won't use the Node.js binary directly but through pnpm commands. Pnpm is a package manager that is an alternative to npm, the officially supported package manager of Node.js. Pnpm offers some valuable improvements over npm, which reduces the installation time and the space taken up by packages.

Trying Solid via Online Playground

The easiest way to get started with Solid is by using the online Playground. Head to the Solid website and navigate to the playground page: https://playground.solidjs.com.



```
import { render } from "solid-js/web";
import { createSignal } from "solid-js";

function Counter() {
  const [count, setCount] = createSignal(1);
  const increment = () => setCount(count => count + 1);

  return (
    <button type="button" onClick={increment}>
      {count()}
    </button>
  );
}

render(() => <Counter />, document.getElementById("app")!)
```

Figure 1.1 *Solid Playground*

The Playground allows you to experiment with Solid in a safe and interactive environment. You can run code and check for errors. The playground automatically executes any code you write and displays the output in the result tab.

NOTE

Solid Playground supports TypeScript, but you need to import types explicitly.

Explicit type-only imports were introduced in TypeScript 3.8, allowing types to be imported from a module while being fully erased during compilation, so they never generate any runtime side effects in the compiled output.

Here's how you can import a type:

```
import { type Component } from "solid-js";
import type { Component } from "solid-js";
```

At times, automatic execution might result in issues if the code is incomplete. In those situations, you can use the refresh button to refresh the output window manually.

Solid is a compiled library. You can inspect the compiled output on the output tab. You can select different compilation targets using the radio buttons at the bottom of the output window.

The playground allows you to work with multiple files. You can share your code using the share button. This feature comes in very handy when asking for help. You can recreate the issue and share the link.

You can reset the editor window by clicking on the trash icon.

The bell icon next to the reset button toggles the error reporting.

Creating a Project From a Template

You can create a local Solid project using one of the project templates offered by the core team: https://github.com/solidjs/templates.

```
$ npx degit solidjs/templates/js my-app
$ cd my-app
$ pnpm i
$ pnpm run dev
```

You can use typescript if you like:

```
$ npx degit solidjs/templates/ts my-app
$ cd my-app
$ pnpm i
$ pnpm run dev
```

In the next section, we will create our own development environment from scratch using Vite. Until then, you can use the online playground to get your feet wet.

Note For React Developers

Solid and React have almost nothing in common other than using JSX for their UI layer. Solid has its own compiler, which produces a different output than React's. It has its own rendering paradigm and its own way of keeping state. If you are familiar with React, it is best to leave your assumptions and expectations aside and try to adapt to Solid's way of doing things to save your time and effort.

CHAPTER 2

Setting Up a Development Environment

In this chapter, we will create a basic development environment for writing single-page Solid applications using Vite.

The Solid documentation recommends using Vite for building apps, so we will use Vite for the majority of our examples. However, occasionally we will need additional tools, for instance, when working with server-side rendering, which we will introduce in their respective chapters.

It is important to note that Vite is not a hard dependency for Solid. Solid's only dependency is its Babel plugin which is `babel-preset-solid`. We have chosen Vite for its features, such as fast refresh, which enables us to swap the application's code while maintaining its state, built-in TypeScript support, and the ability to use CSS modules for styling components.

If you prefer alternative bundlers, please refer to Appendix 1, *Setting Up the Development Environment with Webpack* for guidance on using Solid with Webpack. These instructions can easily be adapted for other bundlers as well.

We are going to use `pnpm` to run npm commands for it offers a better developer experience over `npm`. However, you can keep using `npm` if you like.

With that being said, let's begin our development journey.

First, create an empty folder:

```
mkdir my-app
```

And navigate into it:

```
cd my-app/
```

Once you are inside, initialize a new npm project by running:

```
pnpm init
```

Open the folder in your preferred code editor. Here, I use Visual Studio Code to open the current directory:

```
code .
```

Then install the required dependencies:

```
pnpm i -D vite solid-js vite-plugin-solid typescript
```

As you see, Vite reduces the number of dependencies to a few modules.

Next, we create a few files at the root of our project:

```
touch index.html tsconfig.json vite.config.ts
```

`index.html` will be served by Vite's development server and it contains the entry point to our Solid application:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>My App</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="/src/index.tsx" type="module"></script>
  </body>
</html>
```

The `index.html` file loads `index.tsx` which we are going to create in a moment.

`tsconfig.json` provides the TypeScript configuration:

```
{
  "compilerOptions": {
    "strict": true,
    "target": "ESNext",
    "module": "ESNext",
    "moduleResolution": "node",
    "allowSyntheticDefaultImports": true,
    "esModuleInterop": true,
    "isolatedModules": true,
    "jsx": "preserve",
    "jsxImportSource": "solid-js",
    "types": ["vite/client"],
    "noEmit": true
  }
}
```

`"jsx": "preserve"` instructs TypeScript not to compile JSX files but to pass them as-is so that we can use Babel-js for that task.

`"jsxImportSource": "solid-js"` sets `solid-js` as the JSX runtime. The `jsxImportSource` setting is required to provide static types to JSX elements.

NOTE

`jsxImportSource` declares the module from which to import the jsx and jsxs factory functions when running files with tsx and jsx extensions.

https://www.typescriptlang.org/tsconfig#jsxImportSource

CAUTION

If you are using VSCode and did not set TypeScript properly, the editor will emit an error like the one below:

```
Cannot find name 'React'
```

If you fail to set `jsxImportSource` option, you will have an error like this one:

```
JSX element implicitly has type 'any' because no interface
'JSX.IntrinsicElements' exists.
```

These are the only Solid-specific options we need to set.

`"types": ["vite/client"],` option imports types from the `vite/client` module. You can learn more about this feature in the Vite documentation:

https://vitejs.dev/guide/features.html#client-types

You can consult the TypeScript docs for the remaining settings:

https://www.typescriptlang.org/tsconfig

Now, open the `vite.config.js` file and add the following content:

```
import { defineConfig } from 'vite';
import solid from 'vite-plugin-solid';

export default defineConfig({
  plugins: [solid()],
});
```

Here we modified the default Vite configuration with an additional plugin. `vite-plugin-solid` internally uses Babel-JS to transform jsx/tsx files and provides hot module replacement with fast-refresh support.

Now, let's open the `package.json` file and add a few scripts:

```
"scripts": {
  "dev": "vite --port 3000",
  "build": "vite build",
  "serve": "vite preview --port 3000"
}
```

The `dev` script runs Vite's development server. The `port` parameter sets the port number for the server.

Although the port value is not required, we set it to provide clear instructions. If the specified port is busy, Vite will continuously increment the port number and connect to the first available port.

The `build` script builds the application while the `serve` script serves the application we built.

Now, it's time to create a simple Solid application. First, add a new folder called `src` on the root directory, and then create the `index.tsx` file inside. Make sure the file path matches the `src` attribute we used in the `index.html` file.

```
mkdir src
touch src/index.tsx
```

Add the following script to `index.tsx`:

```
import { render } from 'solid-js/web';

export const App = () => {
  return (
    <div>Hello World!</div>
  );
}

render(() => <App />,  document.getElementById('root')!);
```

This SolidJS code imports the `render` function from the `solid-js/web` package and then defines an `App` component that returns a single div element with the text `"Hello World!"`. Finally, the `render` function is called to mount the `App` component to the DOM element with the ID `root`. The HTML-like syntax used here is called JSX, which is employed to define the component's structure in a manner that resembles HTML. Solid utilizes JSX to construct the view layer.

The `render` function requires us to provide a component that returns a JSX element.

Although we wrap the `App` component in an anonymous function in our example, both of these usages are valid because the `App` function itself is a component that returns a JSX element:

```
render(App,  document.getElementById('root'));

render(() => <App />,  document.getElementById('root'));
```

In this book, we prefer the second method due to difficulties with Vite's HMR using the first one; it fails to replace the imported modules. In other words, you may have issues with HMR if you pass the `App` component directly to the `render` function:

```
render(App,  document.getElementById('root'));
```

In those situations, try changing `App` to `() => <App />` to resolve the issue:

```
render(() => <App />,  document.getElementById('root'));
```

If the problem persists, you can try to force a full-page refresh by using the `@refresh reload` pragma. Open the module file that you wish to refresh the page for when changes are made, and insert the following comment at the top of the file:

```
// @refresh reload
```

This pragma instructs the HMR system to fully reload the page when this module is updated.

NOTE

In the context of Hot Module Replacement (HMR), a pragma is a special comment or annotation in the source code that directs the HMR system on how to handle module updates. The `@refresh reload` pragma, in particular, triggers a full page reload, ensuring all changes are applied cleanly.

The `!` symbol that follows the mount point in the `render` call is known as the non-null assertion operator. It tells the TypeScript compiler to treat the element as if it is guaranteed to not be `null` or `undefined`:

```
document.getElementById('root')!
```

To run the development server, we need to execute the `dev` command in the terminal:

```
pnpm run dev
```

Now, visit `http://localhost:3000/` in your browser to see the result.

Now, let's update the application by adding a few exclamation marks, just to see if fast-refresh works:

```
<div>Hello World!!</div>
```

You should see the exclamation marks appear on the screen without the need to restart the server or refresh the page. If not, it means your development server does not work as expected. You need to check if you've made any mistakes while setting it up.

With the setup complete, we are now ready to dive into how Solid works.

CHAPTER 3

On SolidJS

In this chapter, we will have a high-level overview of Solid, covering the fundamental principles that Solid is built upon. If you find any of these concepts difficult to grasp, don't worry as we will delve into each one of them in greater detail later in the book.

Before getting into how Solid works, let's talk about the problems Solid aims to solve.

The Problem Solid Solves

We create applications to manipulate data. Data may come from various sources, such as databases or files, or it may be hard-coded into the program.

The data that an application operates on lives inside the computer's memory. What we see on the screen is its visual representation. Data is the source of truth, but the pixel on the screen is its manifestation.

In an actual application, data changes over time. We use the term *state* to describe the state of data at a specific point in time. When data changes, we need to change its visual representation too, to avoid inconsistencies.

For example, consider a character's position in a 2D video game:

```
Position { x: 0, y: 0 }
```

When the player presses an arrow key for moving or jumping, the data that controls the character's position changes. The program needs to update the character's visual representation on the screen too. It is essential to keep track of data and ensure that all its representations reflect its actual state.

There are different approaches to solving this synchronization problem.

In non-reactive systems, when we update the state we update its visual representation too. This is a tedious and error-prone task, so we structure our application in a certain way to minimize errors, which is commonly referred to as an architectural pattern. MVC, MVP, and Flux are some well-known examples of architectural patterns.

For example, in the *MVC* pattern, the *Model* represents the state of data, the *View* is the visual representation of the model, and the *Controller* is the glue code that runs the synchronization logic between the Model and the View. After updating the state, the Model calls the controller and the controller updates the UI layer. Usually, there is an explicit invocation of the method that notifies the view layer of a change.

However, adhering to a specific architectural pattern can be limiting, especially for large applications, and they bring their own set of problems.

In reactive systems, data is reactive and the UI derives its state directly from the data.

Updating data triggers a synchronization logic that updates the UI. This practically makes the UI a side effect, reducing potential errors.

```
View = fn(State)
```

In this approach, `fn` is a transform function that takes the state as input and produces a corresponding view. Since `fn` is pure, it always outputs the same result for a given state, making it easy to reason about the behavior of the system.

Synchronizing the state and the UI is important, but it is not the only problem we face when working with browsers. Modern browsers have a very complex rendering pipeline. A piece of data goes through several successive stages before being displayed on the page.

Let's take a look at these stages briefly and then talk about how Solid helps us with them:

1. Construction of the DOM Tree
    This is the initial step where data is converted into a tree-like structure called the DOM. Depending on the source of the data, DOM nodes can be created programmatically using the browser API or it could be generated automatically by the browser using the HTML code that is returned from the server.

2. Construction of the CSSOM Tree
    The CSS is parsed into a CSS Object Model (CSSOM) tree. The browser takes the CSS code and constructs another tree-like structure called the CSSOM, which represents the styles and visual rules for the web page. This step is crucial because it determines the visual appearance of the elements on the page.

3. Render Tree Construction

The browser takes the DOM and CSSOM trees and combines them into a single tree, called the Render Tree. This tree represents the visual structure of the web page, including the styles and positions of each element, determining which elements should be displayed and how they should be laid out.

4. Layout

In this stage, the browser figures out where the elements go on the page. It calculates the position and size of each element in the Render Tree, based on factors like the viewport size and element dimensions.

5. Painting

The browser takes the Render Tree and paints the elements onto the screen, applying styles, colors, and textures. At this stage, the end user can see the element on the page, however, elements that are part of the composited layers may not be fully interactive until the compositing stage is complete.

6. Compositing

The painted elements are composited together to form the final image displayed on the screen called a frame. Elements are stacked together or put into layers based on their properties like `z-index`. Once the layers are created, they are organized into a Layer Tree. This tree structure maintains the hierarchy of the visual elements and their relationships, allowing the browser to understand which elements are in front of or behind others.

This stage is especially important for complex animations and scrolling. The composited frame is then sent to the screen for display. High refresh rate displays require this process to occur at a minimum of 60 times per second (60 Hz) to provide a smooth and fluid user experience.

It’s important to note that this pipeline is not a strictly linear process. Modern browsers employ various optimization techniques, such as speculative parsing and incremental rendering, to improve the perceived performance and responsiveness of web pages. However, improper actions on our part can disrupt this process, leading to significant performance degradation and user experience issues.

For example, if you modify the element’s size after it is painted, you will force the browser to recalculate the layout and repaint the element. Applying certain CSS properties like opacity, transform, or filter can cause the browser to create a new compositing layer for the affected element.

Layout calculations and repaints are expensive operations that can significantly impact performance. If you manipulate DOM nodes while they go through these stages, you will break the pipeline and make the process fall back to the previous stage, or worse it will make the browser cancel all the work it did and start over.

Understanding the rendering pipeline is vital for performance optimization. By minimizing unnecessary reflows and repaints, we can enhance the efficiency of our web applications.

Solid, like many modern frameworks, observes the browser's rendering pipeline when building and modifying the DOM tree. It helps us to minimize DOM modifications, allowing us to batch them and apply them together, as this can minimize the number of reflows and repaints.

Long-running JavaScript tasks can block the main thread, causing the browser to freeze and preventing updates to the rendering pipeline. Solid employs a scheduler to gain more control over the rendering pipeline, allowing it to optimize performance, prioritize updates, and maintain a smooth and responsive user experience. Because of this, even if we run a state update in a tight loop, Solid's UI never becomes unresponsive.

Solid also helps us to take advantage of various optimization techniques like code splitting and lazy loading which can improve the application's initial load time and perceived performance.

In conclusion, Solid helps us to write more efficient JavaScript code, ensuring a smoother user experience.

How Solid Works?

At its core, Solid has two guiding principles and everything else revolves around them:

- Reactive data
- Composable UI

Reactive data means the changes to the data are propagated to all interested parties, including the UI layer. This ensures that the UI is always up-to-date and reflects the current state of the data.

Composable UI means the UI layer is constructed from small, reusable pieces called components. Components can be combined to build highly complex applications. This approach makes it easier to develop, test, and maintain the UI code.

Reactive Data

Solid uses signals for reactive data. The concept of a signal is quite simple: It is a wrapper around a regular JavaScript value that keeps a list of subscribers which are notified whenever the value changes.

To create a signal, we import the `createSignal` function into the current module and call it with an initial value:

```
import { createSignal } from "solid-js";
```

```
const [getCount, setCount] = createSignal(0);
           ↑           ↑
//     accessor    setter
```

`createSignal` returns an array with two items: a getter and a setter. The getter function, commonly referred to as the signal accessor, retrieves the stored value, while the setter function updates it.

We use array destructuring to extract them onto local variables.

Array destructuring is a deliberate design choice because:

- We can use any variable name we prefer.
- It provides separation of concerns by enforcing read-and-write segregation.

Signals provide unidirectional data flow by design. Data is always updated at its source, only then the new value trickles down to its consumers through effects.

Although having an isolated setter and a getter function is not a requirement for unidirectional data flow, it prioritizes readability over the co-location of the said functions.

Signals can store any type of data, including complex objects.

Signals are tracked within a tracking scope. A tracking scope, also known as a reactive context, is a function scope created by `createEffect` or similar functions from the core module that can create subscribers to monitor reactive values. These subscribers are designed to perform specific tasks, such as running side effects, and are deeply integrated into the reactive system.

```
import { createEffect } from "solid-js";

createEffect(() => {
  console.log(getCount());
});
```

`createEffect` creates an effect that is called whenever the signal updates. The callback function we pass to `createEffect` needs to read the signal in order to subscribe to it. Subscription takes place automatically upon reading the signal.

Since `getCount` is a function, we had to call it to retrieve the value.

Solid uses a naming convention for the getter functions that do not include the get prefix, for example, `count` instead of `getCount`.

While this convention may be confusing at first, it is an intentional design choice that promotes clear and concise variable names. The benefit becomes more apparent when we introduce the concept of derived signals which are pure functions that transform reactive values.

```
const double = () => count() * 2;
```

Note that `double` is not a reactive value itself but recalculates its value each time it is called. When used within a reactive scope, like in `createEffect`, it re-runs its transformation whenever the effect re-executes:

```
createEffect(() => {
  console.log(double());
});
```

We will revisit this topic later in Chapter 5, *Tracking State With Signals*.

Now that we have created our effect, we can update the signal. We will use the `setInterval` function:

```
setInterval(() => {
  setCount(count() + 1);
}, 1000);
```

This code sets a timer that increments the `count` value by `1` every second.

Each signal keeps its own subscribers. When an effect reads a signal, it will be added to the signal's subscribers queue.

There is a dynamic dependency between an effect and a signal.

There is no manual process for subscribing or unsubscribing; it all occurs automatically. When the code is executed, the effect reads the signal and gets added to the signal's subscribers and called back when the signal's value is updated. Once it is called back, the effect will be removed from the subscribers. If the effect re-reads the signal while it is being called back, it will be added to the subscribers again to be called upon the next update. We will explain the reason for this later when learning about Solid's reactive core.

If an effect fails to read a signal for some reason, such as the signal being wrapped in a conditional statement, the effect will not be able to subscribe to the signal.

```
let x = 0;
createEffect(() => {
  if(x > 5) {
    console.log(count());
  }
});
```

There are many-to-many relationships between signals and effects. An effect can subscribe to multiple signals. A signal can be observed by multiple effects.

Composable UI

We already mentioned that being composable means the UI layer is made up of small, independent, reusable pieces called components that can be put together to form complex structures.

A Solid component is a JavaScript function that returns a JSX element. JSX is a special syntax that resembles HTML or XML but is more powerful, as it allows the execution of JavaScript expressions within it. We use JSX elements to describe what we want to see on the page.

Components serve as building blocks for applications.

Here is a simple Solid component that returns `H1` element with `"Hello World!"` as its content:

```
const Greeting = ()  => {
  return <h1>Hello World!</h1>
}
```

Solid uses a compiler to convert JSX elements into native DOM nodes. The component above will be converted into the following code:

```
import { template as _$template } from "solid-js/web";
var _tmpl$ = /*#__PURE__*/_$template(`<h1>Hello World!`);
const Greeting = () => {
  return _tmpl$();
}
```

Solid does not use `document.createElement` calls in the compiled output but instead uses HTML strings that produce HTML fragments. That is for achieving a better performance while producing a smaller bundle size.

The `template` function creates DOM nodes from the provided string value.

The `/*#__PURE__*/` comment is a directive for the compiler indicating that the function call it precedes is pure. This means the function always returns the same result if given the same arguments.

We will revisit this topic later in *Chapter 09, Composing User Interfaces*.

Now that we have a component, it is time to display it on the screen. We use the `render` function for it.

The entry point for any Solid application is the `render` function. It takes two arguments: a component to render (commonly called the root component) and a DOM element where the root component's output will be mounted.

Listing 3.1 *The render function mounts the root component to the DOM (ch03/example-01)*

```
import { render } from 'solid-js/web';

const Greeting = ()  => {
  return <h1>Hello World!</h1>
}

render(Greeting, document.body);
```

NOTE Although mounting applications directly onto the `body` element is strongly discouraged, we will do it for clarity and simplicity. It is discouraged because `body` is a common ancestor for all page elements, and it could be mutated by other libraries as well, causing inconsistencies and hard-to-catch bugs.

The `render` function is not exported by the core module but through `solid-js/web`. The aim is to isolate the core library from runtime-dependent logic, allowing Solid to be used inside different runtime environments without conflict.

Solid owes its composable traits to JSX, and JSX supports a variety of features that make it very practical and easy to work with.

Listing 3.2 *JSX allows components to be nested like HTML elements (ch03/example-02)*

```
const Heading = ()  => <h1>Hello World!</h1>;
const Message = ()  => <p>Here is a message for you.</p>;

const App = () => {
  return (
    <div>
      <Heading />
      <Message />
    </div>
  );
}
```

We can use expressions inside a JSX element via `{}`. An expression is a unit of code that evaluates to a value.

Listing 3.3 *Use curly braces to embed JavaScript expressions inside a JSX element (ch03/example-03)*

```
const Counter = () => {
  const [count, setCount] = createSignal(0);

  setInterval(() => {
    setCount(count() + 1);
  }, 1000);

  return (
    <div>Count: {count()}</div>
  );
}
```

Listing 3.4 *Use a ternary expression for conditional rendering inside a JSX element (ch03/example-04)*

```
const Counter = () => {
  const [count, setCount] = createSignal(0);

  setInterval(() => {
    setCount(count() + 1);
  }, 1000);

  return (
    <div>Count: {count()} {count() % 2 === 0 ? 'Even' : 'Odd'}</div>
  );
}
```

Components receive data via props — special attributes passed in JSX:

```
<Greeting name="John Doe" age={25} />
```

JSX attributes are collected into a single `props` object, which is passed as the first argument to the component function.

Listing 3.5 *Attributes are passed to the component definition as the props object (ch03/example-05)*

```
function Greeting(props) {
  return <h1>Hello, {props.name}! You are {props.age} years old!</h1>;
}

const App = () => {
  return <Greeting name="John Doe" age={25} />
};
```

Composition and reactivity give us all the flexibility we need to write efficient and developer-friendly applications.

Please note that these examples fall short of displaying the true power of JSX. Since we are going to have a dedicated chapter on JSX, we will move on now.

Solid uses a compiler to convert JSX into DOM elements. Using a compiler brings certain benefits like producing smaller bundles with better-optimized code that we cannot write by hand, at least not consistently.

If you look at the frontend frameworks benchmarks, you will see that Solid applications are as performant as their vanilla JavaScript counterpart:

https://krausest.github.io/js-framework-benchmark/

TIP Rendering benchmarks gives us an idea of how fast a library is, but we should approach these benchmarks with caution because the browser's rendering pipeline will be the bottleneck for all competing projects. That is why there is a tiny difference between libraries unless the library does something very costly.

In Solid, state is completely decoupled from the UI, enabling a more maintainable and scalable architecture.

In Listing 3.6, the `Counter` component defines its state using a signal scoped to the `Counter` function itself. However, as demonstrated in Listing 3.7, we can move the state outside the component and into a broader scope — in this case, the global scope — and the component continues to function as before.

Listing 3.6 *State is declared locally inside the component using a signal (ch03/example-06)*

```
import { createSignal } from "solid-js";
import { render } from "solid-js/web";

const Counter = () => {
  const [count, setCount] = createSignal(0);

  const handleClick = () => {
    setCount(count() + 1);
  };

  return (
    <div onClick={handleClick}>Count: {count()}</div>
  )
}

render(() => <Counter />, document.querySelector('#root')!);
```

Listing 3.7 *State can reside outside a component (ch03/example-07)*

```
import { createSignal } from "solid-js";
import { render } from "solid-js/web";

const [count, setCount] = createSignal(0);

const Counter = () => {
  const handleClick = () => {
    setCount(count() + 1);
  };

  return (
    <div onClick={handleClick}>Count: {count()}</div>
  )
}

render(() => <Counter />, document.querySelector('#root')!);
```

The Advantages of Solid Over Its Alternatives

Now, let's talk about what makes Solid a valuable tool for developing front-end applications.

Performance

First and foremost, Solid is a performant library and its performance can be attributed to its fine-grained reactive updates.

If you inspect the following code inside the browser's developer tools, you will see that only the `innerText` of the div element gets updated:

Listing 3.8 *Solid performs fine-grained updates (ch03/example-08)*

```
import { createSignal } from "solid-js";
import { render } from "solid-js/web";

const Counter = () => {
  const [count, setCount] = createSignal(0);

  setInterval(() => {
    setCount(count() + 1);
  }, 1000);

  return (
    <div>Count: {count()}</div>
  );
}

render(() => <Counter />, document.body);
```

Solid achieves this by compiling components into dynamic and static parts which we will discuss later in the book. Static parts remain unchanged, whereas dynamic parts are wrapped in an effect so that their content can be updated surgically.

Unlike many popular frontend frameworks, Solid does not use a virtual DOM. Instead, it compiles components to native DOM nodes. This approach, combined with fine-grained reactive updates, makes Solid applications more resistant to performance degradation.

The virtual DOM is a programming abstraction that keeps a copy of the UI in memory and synchronizes it with the actual DOM only when the application state changes. This strategy exists because direct DOM manipulation is traditionally slow and expensive. However, the virtual DOM itself can introduce performance overhead — especially as it re-evaluates and re-renders entire sections of the DOM tree, even when only part of it has changed. To mitigate this, frameworks often implement techniques like diffing and batching, which reduce unnecessary updates but add complexity and can still incur costs.

One consequence of Solid's use of native DOM nodes is that a DOM node cannot be inserted in multiple places without moving it. This is a natural limitation of the DOM: a node can only exist in one place at a time.

Listing 3.9 *A DOM node moves when re-inserted (ch03/example-09)*

```
import { render } from "solid-js/web";

function App() {
  let button = <button>Click</button>;
  return (
    <div>
      {button}
      <h1>Hello World</h1>
      {button}
    </div>
  );
}

render(() => <App />, document.body);
```

When rendered in the browser, the `App` component produces the following output:

```
<div>
  <h1>Hello World</h1>
  <button>Click</button>
</div>
```

This behavior can lead to subtle bugs in your application if not handled properly. Always ensure that nodes are cloned or recreated when they need to appear in multiple parts of the DOM to avoid unintended side effects.

A simple solution is to wrap the element in a function:

```
function App() {
  let button = () => <button>Click</button>;
  return (
    <div>
      {button()}
      <h1>Hello World</h1>
      {button()}
    </div>
  );
}
```

Each call to `button()` returns a fresh DOM node, which prevents problems caused by reusing the same node in multiple places.

An alternative approach is to define the button as a component:

```
import { render } from "solid-js/web";

function App() {
  let Button = () => <button>Click</button>;
  return (
    <div>
      <Button />
      <h1>Hello World</h1>
      <Button />
    </div>
  );
}
```

While this method introduces a bit more complexity — since each `<Button />` triggers a `createComponent` call under the hood — it improves modularity and maintainability by encapsulating the button in its own component.

Synchronous Execution

In Solid, both state updates and effects are executed synchronously, making it straightforward to work with.

Some libraries use asynchronous logic, utilizing either the microtasks or the event loop to execute effects. This is done to ensure consistency as an update may trigger subsequent updates that can put the UI layer in an inconsistent state or cause unnecessary re-renders. However, asynchronous rendering can make code difficult to reason about, limiting our ability to manage it effectively since we have less room for maneuvering.

Vanishing Components

In Solid, components serve the purpose of organizing code. They are executed only once when the application is loaded. Subsequent updates operate independently of the component structure. This approach gives way to better-optimized code with better developer experience as updates are not affected by the component structure.

Compiled Code

Solid employs a compiler to transform JSX into native DOM elements, resulting in smaller bundles. The use of a compiler offers several advantages, such as producing more optimized code, more consistent and predictable performance, and the ability to modify the output without altering the API or the semantics.

Unidirectional Data Flow

Signals offer unidirectional data flow by design which reduces code complexity and makes it easier to manage the flow of information within the application. In other words, it is easier to track data through the component tree.

Declarative data and declarative UI

Declarative programming is a style where the consuming code describes what it needs, and the component handles how it gets done. The imperative logic is abstracted away — hidden from the outside world.

For example, imagine we have a list of images and want to display them as a slideshow. Instead of manually writing code to create slides, manage timers, and switch images, we can simply write:

```jsx
<Slider
  images={['1.png', '2.png', '3.png']}
  duration={5}
  autoplay={true}
/>
```

This tells the `Slider` component everything it needs: a list of images, how long to show each one, and whether it should autoplay. Our code — the consuming side — doesn't need to worry about how the slideshow actually works. The component takes care of transitions, timing, and state management behind the scenes.

This approach keeps our code clean and focused. We can build powerful interfaces just by describing our intent, without getting lost in implementation details.

Composable UI

Solid components can be nested and they can be passed around like any other variable which makes them highly flexible and powerful, leading to more efficient and maintainable code.

Modular and Portable

Solid components offer clear boundaries and well-established relationships between data and UI, making them highly modular.

With Solid, we can extract parts of our application into separate modules and use them anywhere we like, aligning well with React's "write once, use everywhere" motto. However, unlike React components, Solid components do not trigger unexpected re-renders, thereby reducing potential bugs and performance issues.

Furthermore, Solid's custom directives allow us to abstract not only UI elements but also event listeners into reusable pieces. This means that you can write fully-functioning services in one project and import and use them in another one.

SSR with streams

Solid allows us to write isomorphic applications with server-side rendering. It provides a reactive API for fetching and rendering asynchronous data, which also supports streaming.

Alternative rendering methods

If you are unable to use a compiler for some reason, you can use Tagged Template Literals or HyperScript to skip the compilation step. While the output may not be as performant or as small as the compiled code, these alternative methods offer a viable option.

Solid has a small API surface, produces small bundles, and offers good results in performance benchmarks.

Thanks to these qualities, re-implementing a library in Solid is often easier than in frameworks like React. In many cases, you can achieve comparable performance without needing special optimizations or workarounds. Porting existing libraries into Solid also tends to be faster and more straightforward than with other libraries.

CHAPTER 4

How Solid's Reactive System Works

In the previous chapter, we covered reactive data just enough to get started. In this chapter, we will learn how Solid implements reactivity by writing our own reactive core that works the same way and has the same API as Solid's.

Solid's reactivity is built on three core components: signals, computations, and memos.

Signals serve as the data sources for other components in the system. They are observable values that other components can subscribe to, enabling them to receive notifications when the signal's value changes.

```
const [count, setCount] = createSignal(0);
```

A signal stores a single value of any data type. Each signal keeps its own subscribers and notifies them when its value changes.

It is the computation object that subscribes to signals and gets notified when a signal's value changes.

```
createEffect(() => {
  console.log(`Count is ${count()}`);
});
```

Understanding computations is crucial for developing the correct mental model of the Solid runtime. Many challenges in Solid stem from misunderstandings about how computations work and interact within the reactivity system. The most frequently asked questions in the community often relate to how computations are created or executed and how dependencies are tracked. Without a solid grasp of these fundamentals, it's easy to encounter unexpected behavior or performance issues. That's why we've dedicated this chapter to an in-depth discussion of the internal APIs and mechanics of computations, providing the foundational knowledge needed to work effectively with Solid's reactivity model.

A computation is not an abstract concept but a concrete object with methods and properties. They are used within the core module and are not directly exposed to the outside world.

There are various ways to create a computation. The `createEffect` function we have been using so far is just one of them. Other methods include `createRenderEffect`, `createComputed`, `createMemo`, `createDeferred`, and `createReaction`.

Although each function creates a computation internally, they either execute at different times to meet specific requirements for the tasks they perform or offer a different API suited to a particular need. For instance, the effects and the render effects are almost identical, except that the render effects are used specifically for rendering DOM nodes and have priority over the effects during their initial execution. Once the rendering phase is completed, they behave like regular effects. We will discuss each of these functions in their respective topics.

Solid doesn't execute computations immediately; instead, it relies on a scheduler. When a signal triggers a computation, the computation is placed on the scheduler's queue. The scheduler is an interruptible task queue that pauses briefly to run more critical code while updating the UI. Due to this feature, even if you update a signal within an effect in a way that could cause an infinite loop, the browser will not become unresponsive.

Although computations don't inherently have priority, the scheduler's execution logic achieves a similar outcome by effectively prioritizing their execution order.

Observer Pattern

Solid employs the observer pattern internally but doesn't rely on event streams or message passing, which might confuse those familiar with other reactive libraries.

There are numerous implementations of the observer pattern, but they all boil down to three styles: those that push the state to observers, those that let observers pull the state as needed, and those that use both methods.

Libraries such as *RxJS* and *CycleJS* use the push style: An observable pushes the updated state to a subscriber in the form of an event or a message. Subscribers always receive the new state through an event.

A typical callback function for executing side-effects in such libraries is as follows:

```
function callback(event) {}
```

SolidJS, *Mobx*, and *Knockout* use a combination of both the pull and push styles: An observable notifies its subscribers, which constitutes the push phase, and each subscriber reaches for the new state as part of the pull phase. The most noteworthy aspect of this approach is that the new data is received from the source directly through re-execution, without any intermediate values or events. Consequently, their callback functions for executing side effects do not rely on events:

```
function callback() {}
```

Re-execution provides a simpler mental model with better performance characteristics, while the event streams can quickly get unwieldy with multiple observables because now we need to use conditionals to handle different events in the subscriber's logic, or we have to transform them before they are passed to their consumers so that they can be consumed directly. This is why libraries like *RxJS* include numerous operators for creating new observables from existing ones by merging, splitting, and transforming them.

Although Solid uses the observer pattern, its documentation avoids explicitly using terms associated with it. The reasons for this avoidance are not clearly stated, but it may be an intentional effort to minimize technical jargon and reduce the cognitive load that comes with the observer pattern.

It's time to get our hands dirty. We will start with a very crude implementation to understand the relationship between the underlying components and then refactor to a more accurate one. You can explore the full code examples in the repository under `ch04`.

The Essence of Reactive Core

Let's start with the requirements for a signal.

A signal should:

- Store a value that changes over time.
- Maintain a list of subscribers.
- Notify subscribers whenever the value changes.
- Add and remove subscribers, as a subscriber's dependency may change over time.
- Provide read-write segregation through getter and setter functions.

With these requirements in mind, we can define the `Signal` type. As outlined, a signal stores a value and maintains a list of subscribers.

```
interface Signal<T> {
  value: T;
  observers: Set<Computation<any>>;
}
```