Contents

[SVELTE 3](#_Toc72674694)

[WHY SVELTE 3](#_Toc72674695)

[CREATING A SVELTE SCAFFOLDING 3](#_Toc72674696)

[PACKAGE.JSON – ON A HIGH LEVEL 3](#_Toc72674697)

[GENERATED PROJECT STRUCTURE – ON A HIGH LEVEL 3](#_Toc72674698)

[STYLES 4](#_Toc72674699)

[DYNAMIC ATTRIBUTES 5](#_Toc72674700)

[HTML TAGS 5](#_Toc72674701)

[REACTIVITY 5](#_Toc72674702)

[REACTIVITE DECLARATIONS 5](#_Toc72674703)

[REACTIVITE - STATEMENTS 5](#_Toc72674704)

[REACTIVITY FOR ARRAYS AND OBJECTS 7](#_Toc72674705)

[PROPS 8](#_Toc72674706)

[DEFAULT VALUES 8](#_Toc72674707)

[SPREAD PROPS 8](#_Toc72674708)

[BLOCK STATEMENTS 8](#_Toc72674709)

[CONDITIONAL STATEMENTS 8](#_Toc72674710)

[LOOPS STATEMENTS 9](#_Toc72674711)

[KEYED EACH BLOCK 9](#_Toc72674712)

[AWAIT BLOCK 9](#_Toc72674713)

[DATA BINDING 11](#_Toc72674714)

[TWO WAY DATABINDING 12](#_Toc72674715)

[ADDING CSS CLASS DYNAMICALLY 12](#_Toc72674716)

[BINDINGS 12](#_Toc72674717)

[BINDING FORM ELEMENTS 12](#_Toc72674718)

[BINDING SELECT DROPDOWN 13](#_Toc72674719)

[BINDING TO ELEMENT REFERENCES 13](#_Toc72674720)

[TWO WAY BINDING WITH COMPONENT PROPS 13](#_Toc72674721)

[EVENT HANDLING 14](#_Toc72674722)

[EVENT MODIFIERS 14](#_Toc72674723)

[COMPONENTS 15](#_Toc72674724)

[TYPES OF COMPONENT 15](#_Toc72674725)

[COMPONENT COMMUNICATION OR DATA SHARING 15](#_Toc72674726)

[CREATING REUSABLE UI COMPONENTS 20](#_Toc72674727)

[CUSTOM UI COMPONENT 20](#_Toc72674728)

[SLOTS 20](#_Toc72674729)

[LIFE CYCLE HOOKS 21](#_Toc72674730)

[ON MOUNT 21](#_Toc72674731)

[STORES 21](#_Toc72674732)

[STORES TYPES 22](#_Toc72674733)

[WRITABLE STORE 22](#_Toc72674734)

[READABLE STORE 26](#_Toc72674735)

[UNSUBSCRIBING FROM STORE SUBSCRIPTION 27](#_Toc72674736)

[AUTO SUBSCRIPTION 27](#_Toc72674737)

[CUSTOM STORES 28](#_Toc72674738)

[STORE CONTRACT 28](#_Toc72674739)

[CONTEXT STORE 29](#_Toc72674740)

[HTTP INTERACTION 29](#_Toc72674741)

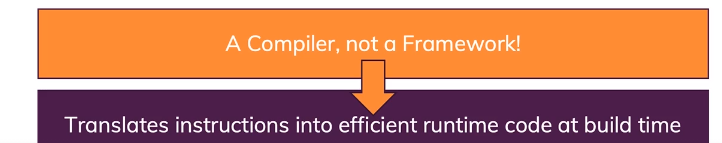
[FETCH API 29](#_Toc72674742)

[SPECIAL ELEMENTS 30](#_Toc72674743)

[DYNAMIC COMPONENT 30](#_Toc72674744)

[SAPPER 31](#_Toc72674745)

# SVELTE



* Svelte converts the app into ideal JavaScript at build time, rather than interpreting your application code at run time. This means you don't pay the performance cost of the framework's abstractions, and you don't incur a penalty when your app first loads.
* Svelte is an alternative to web frameworks like React, Vue, and Angular. Like those, Svelte can be used to build entire web apps. ***It can also be used to create custom elements that can be used in existing web apps implemented with other frameworks.***

## WHY SVELTE

* Svelte apps have smaller bundle sizes than equivalent apps created with other web frameworks. This is achieved by compiling the application code to a single, optimized JavaScript file that includes a very small amount of framework code.
* Svelte is a web application compiler implemented in TypeScript. It is not a runtime library.
* Some web frameworks, including React and Vue, use a virtual DOM to optimize rendering changes. When components re-render, the framework builds a new version of the DOM in memory and then compares it to the

previous version. Only the differences are applied to the actual DOM. While this is faster than updating everything in

the actual DOM, it does take time to build a virtual DOM and compare it to the previous one. Svelte provides reactivity without using a virtual DOM. It does this by tracking changes to top-level component variables that affect what each component renders and only re-rendering those parts of the DOM when changes are detected. This contributes to good performance.

* Svelte dramatically simplifies component and application state management. Contributing features include context, stores, and module context, each of which is covered in detail later.
* Svelte provides runtime warnings for accessibility issues. For example, <img> elements that have no alt attribute

are flagged.

## CREATING A SVELTE SCAFFOLDING

|  |  |
| --- | --- |
| **CREATES A BASIC PROJECT STRUCTURE**  npx degit sveltejs/template <app-name> | * The degit tool is useful for project scaffolding. * It clones a git repo, by default the master branch. * In this case "sveltejs" is the username and "template" is the repo. * The second argument is the name of the directory to create. |
| cd app-name | Switch to the App directory |
| npm install | Install the dependencies in package.json |
| npm run dev | This starts a local HTTP server and provides live reload, unlike npm run start which omits live reload. |
| Browse localhost:5000 | To Access the App |

### PACKAGE.JSON – ON A HIGH LEVEL

A peek at the package.json file reveals two things.

* The first is that Svelte uses Rollup by default for module bundling. If desired, it can be changed to use Webpack or Parcel.
* The second is that Svelte apps have no required runtime dependencies, only dev Dependencies

### GENERATED PROJECT STRUCTURE – ON A HIGH LEVEL

|  |  |
| --- | --- |
|  | * The most important starting files are public/index.html , src/main.js , and src/App.svelte . These files use tabs for indentation but, if preferred, the tabs can be replaced by spaces |
| The file public/index.html contains the following:  <!DOCTYPE html>  <html>  <head>  <meta charset="utf8" />  <meta name="viewport" content="width=device-width" />  <title>Svelte app</title>  <link rel="icon" type="image/png" href="favicon.png" />  <link rel="stylesheet" href="global.css" />  <link rel="stylesheet" href="bundle.css" />  </head>  <body>  <script src="bundle.js"></script>  </body>  </html>   * This pulls in two CSS files and one JavaScript file. **global.css holds CSS that can affect any component.** * bundle.css is generated from the CSS in each component. * bundle.js is generated from the JavaScript and HTML in each component and any other JavaScript in the app. |

|  |  |
| --- | --- |
| **src/main.js** | **src/App.sevlte** |
| import App from './App.svelte';  const app = new App({  target: document.body,  props: {  name: 'world'  }  });  export default app;   * This renders the App component. The target property specifies where the component should be rendered. For most apps this is the body of the document. This passes the name prop to the App component. | <script>  export let name;  </script>  <style>  h1 {  color: purple;  }  </style>  <h1>Hello {name}!</h1>   * Exported variables can be set as props in files that use the component. * Curly braces are used to output the value of a JavaScript expression. This is referred to as "interpolation". * Curly braces are also used for dynamic attribute values. |

## STYLES

|  |  |
| --- | --- |
| * The styles in component are scoped within the component itself. * But if we want to apply the CSS globally. We need to use special selector “global”. * This css will impact “p” of all components and element having “myclass” class. | <style>  :global(p){  color:orange  }  :global(.myclass){  color: red;  }  </style> |

## DYNAMIC ATTRIBUTES

|  |  |
| --- | --- |
| USUAL WAY | DYNAMIC ATTRIBUTE |
| <img src={src} alt="A man dances."> | <img **{src}** alt="A man dances.">  If the attribute name and props name are same- We can use above short hand |

## HTML TAGS

|  |  |
| --- | --- |
| * Ordinarily, strings are inserted as plain text, meaning that characters like < and > have no special meaning. * But sometimes you need to render HTML directly into a component. For example, the words you're reading right now exist in a markdown file that gets included on this page as a blob of HTML. * In Svelte, you do this with the special {@html ...} tag: | <script>  let string = `this string contains some <strong>HTML!!!</strong>`;  </script>  <p>{@html string}</p> |

## REACTIVITY

### REACTIVITE DECLARATIONS

* Svelte automatically updates the DOM when the component's state changes. Often, some parts of a component's state need to be computed from other parts (such as a fullname derived from a firstname and a lastname), and recomputed whenever they change.
* Reactive values become particularly valuable when you need to reference them multiple times, or you have values that depend on other reactive values.
* The change in the value of reactive variable is not instant. This is something happens asynchronously . So if we try to tap the vale immediate – we might see the old value.

|  |  |
| --- | --- |
| <script>  let firstName='';  let lastName='';  **$:fullName = firstName +' '+ lastName; 🡨 We don’t need to declaration of “fullName” reactive variable e.g. let fullName**  </script>  <main>  <input type="text" bind:value="{firstName}"/>  <input type="text" bind:value="{lastName}"/>  <p>${fullName}</p>  </main> | **EXAMPLE**  <script>  let count = 0;  $: doubled = count \* 2; 🡨 Reactive declaration  function handleClick() {  count += 1;  }  </script>  <button on:click={handleClick}>  Clicked {count} {count === 1 ? 'time' : 'times'}  </button>  <p>{count} doubled is {doubled}</p> |

### REACTIVITE - STATEMENTS

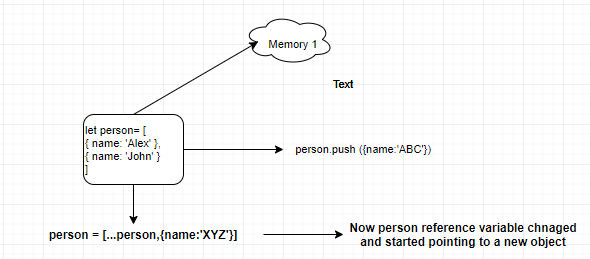
|  |  |
| --- | --- |
| * We're not limited to declaring reactive values — we can also run arbitrary statements reactively. For example, we can log the value of count whenever it changes:   **$: console.log(`the count is ${count}`);**   * We can easily group statements together with a block:   $: {  console.log(`the count is ${count}`);  alert(`I SAID THE COUNT IS ${count}`);  }   * We can even put the $: in front of things like if blocks:   $: if (count >= 10) {  alert(`count is dangerously high!`);  count = 9;  } | EXAMPLE  <script>  let count = 0;  $: if (count >= 10) {  alert(`count is dangerously high!`);  count = 9;  }  function handleClick() {  count += 1;  }  </script>  <button on:click={handleClick}>  Clicked {count} {count === 1 ? 'time' : 'times'}  </button> |

#### DEPENDENCIES OF REACTIVE VARIABLES AND STATEMENTS

|  |  |  |
| --- | --- | --- |
| DECLARATION:   * For declaration, the reactive variables depends on state variables to change its state. * Here fullName is reactive variable which depends upon the “firstName” & “lastName” state variables. | | **let firstName='';**  **let lastName='';**  **$:fullName = firstName +' '+ lastName;** |
| **REACTIVE STATEMENTS AND ITS DEPENDENCIES** | | |
|  | | |
| **ORDER OF EXECUTION OF REACTIVE STATEMENT** | | |
| let count = 0;  $: console.log("Double", double);  $: double = count+2; | * In a ideal scenario- the reactive varibales are evaluated from top to down * But for this exampe - “console” statements depends on the “double” statement. So to resolve such depencies – Svelte rearranges the dependency tree during compilation. | |

#### IMMUTABILITY OF ARRAYS AND OBJECT

* In JavaScript object and arrays are reference types.
* If are using an array / object to show the data in the DOM. Svelte will consider the reference types are changed only when reference start point to a new object



|  |
| --- |
| <script>  import ContactCard from './ContactCard.svelte';  let username;  let name;  let email;  let phone;  let imgUrl;  let contactCards=[  {  "id": 1,  "name": "Leanne Graham",  "username": "Bret",  "email": "Sincere@april.biz",  "phone": "1-770-736-8031 x56442",  "website": "hildegard.org",  }  ];  function addContactCard(){  const newContactCard = {  id:Math.random(),  name:name,  username:username,  email:email,  phone:phone  }  **contactCards = [...contactCards,newContactCard]; 🡨 UPDATING THE REFERENCE**  }  </script>  <main>  <h3>Contact Card List</h3>  {#each contactCards as contact , index (contact.id)} 🡨ITERATING THE CONTACT CARD ARRAY  <ContactCard  username="{contact.username}"  name="{contact.name}"  email="{contact.email}"  phone="{contact.phone}"  imgUrl ="https://randomuser.me/api/portraits/women/{index}.jpg"/>  {/each}  </main> |
|  |

### REACTIVITY FOR ARRAYS AND OBJECTS

|  |  |
| --- | --- |
| * Because Svelte's reactivity is triggered by assignments, using array methods like push and splice won't automatically cause updates. * WILL NOT WORK   function addNumber() {  numbers.push(numbers.length + 1);  numbers = numbers;  }   * WILL WORK:   function addNumber() {  numbers = [...numbers, numbers.length + 1];  }  We can use similar patterns to replace pop, shift, unshift and splice.  Assignments to properties of arrays and objects — e.g. obj.foo += 1 or array[i] = x — work the same way as assignments to the values themselves.  function addNumber() {  numbers[numbers.length] = numbers.length + 1;  }  In the the below -It won't update references to obj.foo.bar, unless you follow it up with obj = obj.  const foo = obj.foo;  foo.bar = 'baz’ | <script>  let numbers = [1, 2, 3, 4];  function addNumber() {  numbers = [...numbers, numbers.length + 1];  }  $: sum = numbers.reduce((t, n) => t + n, 0);  </script>  <p>{numbers.join(' + ')} = {sum}</p>  <button on:click={addNumber}> Add a number</button> |
| **A simple rule of thumb: the name of the updated variable must appear on the left-hand side of the assignment**. |

## PROPS

### DEFAULT VALUES

|  |  |  |
| --- | --- | --- |
| We can easily specify default values for props in  If we now add a second component without an answer prop, it will fall back to the default: | PARENT  <script>  import Nested from./Nested.svelte';  </script>  <Nested answer={42}/>  <Nested/> | **CHILD**  <script>  export let answer = 'a mystery';  </script>  <p>The answer is {answer}</p>  **OUTPUT**  The answer is 42  The answer is a mystery |

### SPREAD PROPS

|  |  |  |
| --- | --- | --- |
| * If we have an object of properties, we can 'spread' them onto a component instead of specifying each one: | **App.svelte** | **Info.svelte** |
| <script>  import Info from './Info.svelte';  const pkg = {  name: 'svelte',  version: 3,  speed: 'blazing',  website: 'https://svelte.dev'  };  </script>  <Info {...pkg}/> | <script>  export let name;  export let version;  export let speed;  export let website;  </script>  <p>The <code>{name}</code> package is {speed} fast.  Download version {version} from <a href="https://www.npmjs.com/package/{name}">npm</a>  and <a href={website}>learn more here</a>  </p> |

## BLOCK STATEMENTS

### CONDITIONAL STATEMENTS

|  |  |
| --- | --- |
|  | {#if color === 'yellow'}  <div>Nice color!</div>  {:else if color === 'orange'}  <div>That's okay too.</div>  {:else}  <div>Questionable choice.</div>  {/if} |

### LOOPS STATEMENTS

|  |  |
| --- | --- |
| **Suppose the variable colors is set to ['red', 'green', 'blue']**   1. **Outputs each color on a separate line using the color.** | EXAMPLE 1:  {#each colors as color}  <div style="color: {color}">{color}</div>  {/each} |
| **ACCESSING INDEX** | {#each colors as color, index}  <div>{index + 1}) {color}</div>  {/each} |
| **OBJECT ITERATION (USING OBJECT DESTRUCTURING)** | {#each people as {name, age}}  <div>{name} is {age} years old.</div>  {:else}  <div>There are no people.</div>  {/each} |

### KEYED EACH BLOCK

### AWAIT BLOCK

|  |  |
| --- | --- |
| Most web applications have to deal with asynchronous data at some point. Svelte makes it easy to await the value of directly in your markup:  {#await promise}  <p>...waiting</p>  {:then number}  <p>The number is {number}</p>  {:catch error}  <p style="color: red">{error.message}</p>  {/await}  Only the most recent promise is considered, meaning you don't need to worry about race conditions.   * If you know that your promise can't reject, you can omit the catch block. You can also omit the first block if you don't want to show anything until the promise resolves:   {#await promise then value}  <p>the value is {value}</p>  {/await} | <script>  async function getRandomNumber() {  const res = await fetch(`tutorial/random-number`);  const text = await res.text();  if (res.ok) {  return text;  } else {  throw new Error(text);  }  }  let promise = getRandomNumber();  function handleClick() {  promise = getRandomNumber();  }  </script>  <button on:click={handleClick}> generate random number</button>  {#await promise}  <p>...waiting</p>  {:then number}  <p>The number is {number}</p>  {:catch error}  <p style="color: red">{error.message}</p>  {/await} |

#### AWAIT EXAMPLE

* To understand the await – lets follow one example with simple fetch API and the finally migrate to await block

|  |
| --- |
| UI : Changing the dogs breed update its respective image in the UI.    <script>  import { onMount } from "svelte";  let alldogData =[];  let src ='';  let selectedDogBreed=''  let isDataLoading = false;  let hasErrors= false;  let showData = true;  $: getDogsData(selectedDogBreed)    function getDogsData(breed){  if(!breed)  return;  isDataLoading = true  fetch(`https://dog.ceo/api/breed/${breed}/images/random`).  then(response => response.json()).  then(dogImageData => {  src = dogImageData.message  isDataLoading = false;  showData =true  }).catch(error =>{  hasErrors = true;  }).finally(()=>{  isDataLoading = false;  });  }  onMount(()=>{  fetch('https://dog.ceo/api/breeds/list/all').  then(dogList => dogList.json()).  then(dogData => {  alldogData = Object.keys(dogData.message)  });  });  </script>  <div class="card">  <div class="card-body">  <select class="form-select" aria-label="Default select example" bind:value="{selectedDogBreed}">  <option selected>Open this select menu</option>  {#each alldogData as dogData}  <option value="{dogData}">{dogData}</option>  {/each}  </select>  </div>  </div>  <hr/>  {#if isDataLoading == true}  <div class="spinner-border text-primary" role="status">  <span class="visually-hidden">Loading...</span>  </div>  {:else if isDataLoading == false && showData }  <div class="card" style="width: 18rem;">  <img {src} class="card-img-top" alt="...">  <div class="card-body">  <h5 class="card-title">Breed Selected : {selectedDogBreed}</h5>  </div>  </div>  {:else if hasErrors}  <div class="alert alert-danger" role="alert">  Error!  </div>  {/if} |
|  |

NOTES :

* If we need to make an API call to fetch the data soon after the page is loaded – Do call it in onMount() lifecycle method.
* In this program we are using reactive statement to make the call whenever the drop down changes.

**PROGRAM FLOW:**

1. We are binding a “selectedDogBreed” variable with the dropdown so whenever the drop down changes it updates this variable . Update in this variable triggers the reactive statement $: getDogsData(selectedDogBreed)
2. The call the fetch API to make the HTTP call.

Now the getDogsData(breed) can also be implemented using async- await as below

|  |  |
| --- | --- |
| async function **getDogsDataAsync**(breed){  if(!breed)  return;  try{  isDataLoading = true;  let dogImageData = await getRandomDogImage(breed);  src = dogImageData.message;  isDataLoading = false;  showData = true;  }catch(error){  hasErrors= true;  }  }  async function **getRandomDogImage**(breed){  let response = await fetch(`https://dog.ceo/api/breed/${breed}/images/random`);  let dogImageData = await response.json();  return dogImageData  } | * The above example of **promise chaining** is a common pattern – but we always need to keep track of the response – specially when it comes to tuning the loader on /off * Async / Await – gives an easy way to do this. Await always wait for a promise and return the response. * The await block in svelte also follow the same pattern . It wait for a promise like “await” |
| THE AWAIT BLOCK : | |
| Step1 : Create a function which return a promise  async function **getRandomDogImage**(breed){  let response = await fetch(`https://dog.ceo/api/breed/${breed}/images/random`);  let dogImageData = await response.json();  return dogImageData  } | |
| Step 2: Embed Await Block  {#await getRandomDogImageForAwaitBlock(selectedDogBreed)}  <div class="spinner-border text-primary" role="status">  <span class="visually-hidden">Loading...</span>  </div>  {:then dogImageData}  <div class="card" style="width: 18rem;">  <img src="{dogImageData.message}" class="card-img-top" alt="...">  <div class="card-body">  <h5 class="card-title">Breed Selected : {selectedDogBreed}</h5>  </div>  </div>  {:catch error}  <div class="alert alert-danger" role="alert">  Error!  </div>  {/await} | |

## DATA BINDING

### TWO WAY DATABINDING

|  |  |
| --- | --- |
| This will two way data binding of value of textbox with name property | <script>  export let name;  </script>  <h1>Hello {name}</h1>  <input type="text" **bind:value="{name}"**/> |

### ADDING CSS CLASS DYNAMICALLY

|  |  |
| --- | --- |
| * class: 🡪 is a directive can be used to add class dynamically * This syntax will add “thumb-placeholder” to the div if the userName is value is truthy(not empty / not null/not false ). | <script>  export let userName;  </script>  <div class="thumb" class:thumb-placeholder="{userName}">  //Some more Elements  </div |

## BINDINGS

### BINDING FORM ELEMENTS

#### BINDING INPUT ELEMENTS

|  |
| --- |
| <script>  $: data = inputValue; 🡨 REACTIVE DECLARATION  let inputValue = '';  </script>  <div class="container">  <div class="row">  <h3>Input Bindings</h3>  <div class="mb-3">  <label for="formInput" class="form-label" >Input</label>  <input type="text" class="form-control" id="formInput" bind:value={inputValue}>  <p>Data Entered:{inputValue}</p>  <p>Data From Reactive variable :{data}</p>  </div>  </div></div> |
| **SHORTHAND OF BINDING**   * If the property and the variable name is same . We can use a shorthand operator.   <script>  let value = '';  </script>  <input type="text" class="form-control" id="formInput" bind:value> |

#### BINDING CHECKBOX ELEMENTS

|  |
| --- |
| <script>  let toggle = false;  </script>  <input type="checkbox" bind:checked={toggle}/>Toggle Content  {#if toggle}  <p>Show Hide content</p>  {/if} |
| CHECK BOX GROUP |

#### BINDING RADIO BUTTONS

|  |
| --- |
| <script>  let gender = 'male'; 🡨 MALE WILL BE SELECTED BY DEFAULT  </script>  <div class="row">  <h3>Radio </h3>  <div class="mb-3">  <p>Selected Gender: {gender}</p>  <div class="form-check">  <input class="form-check-input" type="radio" name="flexRadioDefault" id="male" value ="male" bind:group="{gender}">  <label class="form-check-label" for="male"> Male</label>  </div>  <div class="form-check">  <input class="form-check-input" type="radio" name="flexRadioDefault" id="female" value ="female" bind:group="{gender}">  <label class="form-check-label" for="female"> Female</label>  </div>  </div>  </div> |

### BINDING SELECT DROPDOWN

#### EXAMPLE

|  |  |
| --- | --- |
| **BINDING IN SINGLE SELECT SELECTBOX** | **BINDING IN MULTI SELECT SELECTBOX** |
| <script>  const colors =[ 'red', 'blue', 'green' ];  let selectedValue= colors[1];  </script>  <select bind:value="{selectedValue}">  {#each colors as color}  <option value={color}>{color}</option>  {/each}  </select>  <p style="color:{selectedValue}">Color</p> | <script>  let selectedHobbies = [];  </script>  <select **multiple** bind:value="{selectedHobbies}">  <option>Sports</option>  <option>Singing</option>  <option>Dancing</option>  </select>  <p>Selected Hobbies: </p>  {#each selectedHobbies as hobbies}  {hobbies}  {/each} |

### BINDING TO ELEMENT REFERENCES

#### BIND:THIS – TO THE DOM ELEMENT

|  |  |
| --- | --- |
| **BINDING WITH INPUT ELEMENT**  <script>  let element;  function reset() {  element.value = "";  }  function getData() {  console.log(element.value);  }  </script>  <input bind:this={element} />  <button on:click={getData}>Save</button>  <button on:click={reset}>Reset</button> | **BINDING WITH DIV ELEMENT**  <script>  let divElement;  function getDivElementData(){  console.dir(divElement);  console.dir(divElement.innerText);  }  </script>  <div bind:this="{divElement}"> Hello World!</div>  <button on:click={getDivElementData}>Get Div Data</button> |
| * **bind:this** is a way to get the reference of any element in the DOM * For example – In the click event handler we want to get hold of the reference of input element * Once we get hold of reference of the element – we can able to get and manipulate the value of the element. | |

### TWO WAY BINDING WITH COMPONENT PROPS

* Just as we can bind to properties of DOM elements, we can bind to component props.
* The below example shows the two way data binding .

|  |  |  |
| --- | --- | --- |
| App.svelte | Component.svelte |  |
| <p>Count From Parent : {parentCount}</p>  <button on:click="{()=>parentCount = parentCount+1}">Increment</button>  <CustomInput bind:count={parentCount}/> | <script>  export let count;  </script>  <p>{count}</p>  <input type="text" bind:value="{count}"/> |  |

## EVENT HANDLING

* Event handling is specified with the on: *event-name* attribute whose value is a function to invoke when the event is dispatched.
* The event name can be the name of a standard DOM event or a custom event. An event object is passed to the given function.

For example:

|  |  |
| --- | --- |
| The function "handleClick" must be defined in the <script> section above. | <button on:click={handleClick}>Press Me</button> |
| **INLINE FUNTION FOR EVENT HANDLING**  Inline event handling using an anonymous function. It just sets the variable "clicked" to the DOM element for the button | button on:click={event => clicked = event.target}>Press Me</button> |

* Multiple event handling functions can be specified for the same event and each will be invoked when the event is dispatched. For example:

**<button on:click={doOneThing} on:click={doAnother}>Press Me</button>**

### EVENT MODIFIERS

Event handlers can specify any number of event modifiers with vertical bars preceding modifier names. For example:

**<button on:click|once|preventDefault={handleClick}>Press Me</button>**

|  |  |
| --- | --- |
| **Event Modifiers** | **Description** |
| **capture** | This causes the handler function to only be invoked in the capture phase, not the default bubbling phase. |
| **Once** | This removes the handler after the first occurrence of the event |
| **passive** | This can improve scrolling performance. Read about it at https://developer.mozilla.org/en-  US/docs/Web/API/EventTarget/addEventListener#Improving\_scrolling\_performance\_with\_passive\_listeners. |
| **preventDefault** | This prevents the default action for the event from occurring. For example, it can stop a form submission. |
| **stopPropagation** | This prevents subsequent handlers in the capture/bubbling flow from being invoked. |

#### EXAMPLE OF PREVENTDEFAULT EVENT MODIFIER

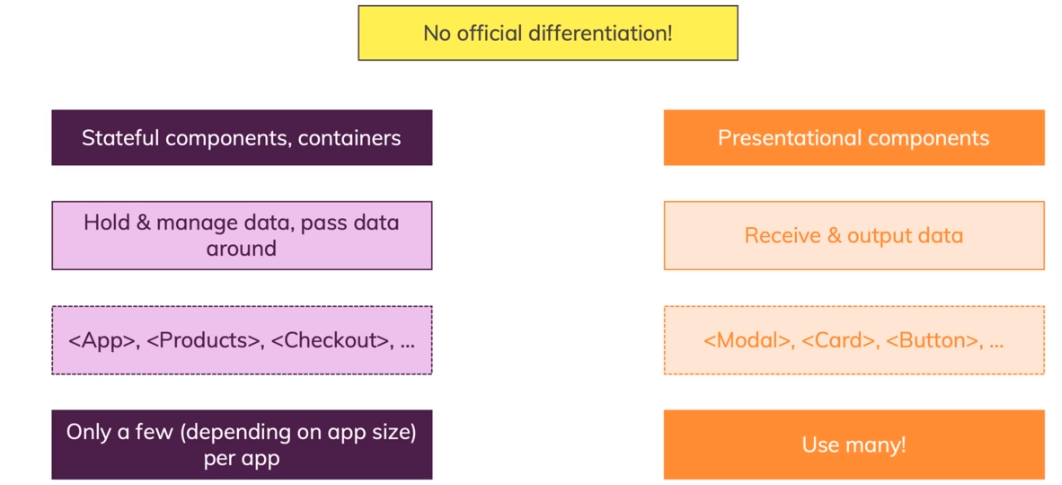
Sds

## COMPONENTS

### TYPES OF COMPONENT

Logically componets can be characterized in two different ways

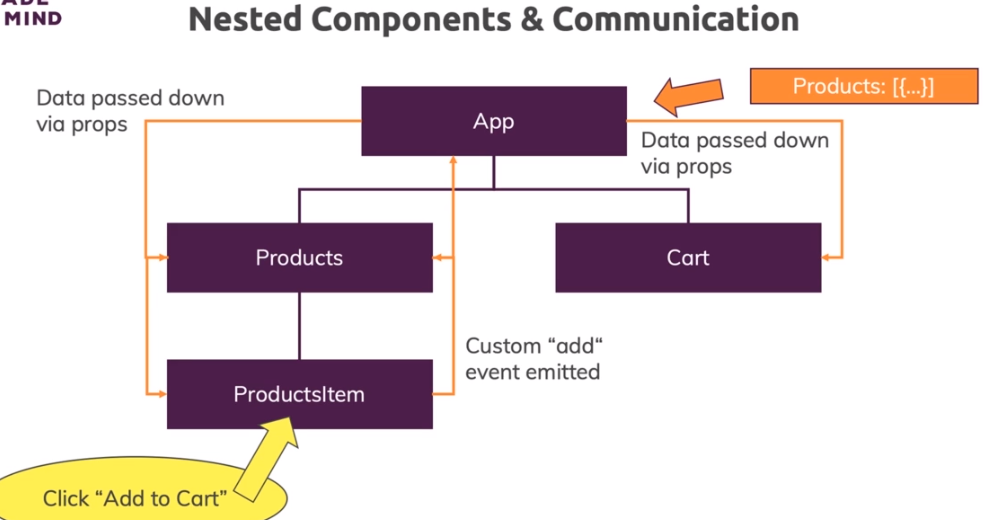
* Stateful Components : This componet has a resposibility to manage the data flow or pass data to presentational components.
* Presentational Component : These component are used only for the view. They receive data from stateful components to render it.



|  |  |
| --- | --- |
| **ContactCard.svelte** | **App.svelte** |
| <script>  // JS Script  </script>  <div>      <header>          <h1>User Name</h1>          <h2>Job Title</h2>      </header>      <div>          <p>Description</p>      </div>  </div> | <script>  import ContactCard from './ContactCard.svelte' #1  </script>  <main>  <ContactCard/> #2  </main>   * Importing svelte file * Including the component (like an HTML tag) |

* The CSS written on component file is limited to that file’s HTML only.
* Svelte also remove the unused CSS before shipping it to the browser.

### COMPONENT COMMUNICATION OR DATA SHARING

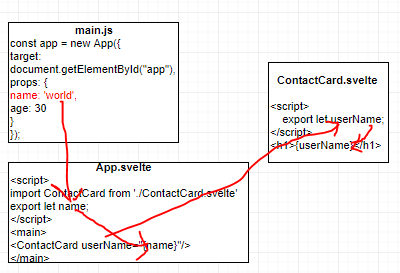


#### WAYS OF DATA SHARING AMONG COMPONENTS

* **PROPS** : - These pass data from parent components to child components.
* **CONTEXTS** : These allow ancestor components to make data available to descendant components.
* **STORES** : These store data outside any component and make it available to all of them.
* **MODULE** **SCOPE** :These store data in component modules and make it available to all instances of the component.

#### PARENT TO CHILD COMMUNICATION - PROPS

* The value of ”name” property in main.js get set in the “export” variable App.svelte
* If the component is included as an markup . The data from parent to child compoent can be sent child using attribute.



NOTE : If the property name and value are same the we can use a shorthand

|  |  |
| --- | --- |
| <ContactCard  userName="{name}"  jobTitle="{jobTitle}"  description="{description}"  imgUrl ="https://randomuser.me/api/portraits/women/2.jpg"/> | <ContactCard  userName="{name}"  jobTitle="{jobTitle}"  {description}  imgUrl ="https://randomuser.me/api/portraits/women/2.jpg"/> |

##### SENDING OBJECT IN PROPS

* In parent to child communication - all the above scenarios we were sending each value explicitly as an individual property. But what if the we have huge object – having large number of properties. - It will be hard to send that property -by-property.
* We use Spread operator to send all properties in one go. The only condition is that the property name should same in child too.

|  |  |
| --- | --- |
| **App.svelte** | **Products.svelte** |
| <script>  let products=[{  "id": 1,  "title": "Fjallraven",  "price": 109.95,  "description": "Your perfect pack",  "category": "men clothing",  "image": "https://fakestoreapi.com/img/81fPKd-2AYL.\_AC\_SL1500\_.jpg"  }];  </script>  {#each products as product , index (product.id)}  <Products {...product} />  {/each} | script>  export let title;  export let description;  export let price;  export let category;  export let image;  export let id;  </script>  <div class="card mb-2 mr-2" style="width: 18rem;">  <img src="{image}" class="card-img-top" alt="...">  <div class="card-body">  <h5 class="card-title">{title}</h5>  <h6 class="card-subtitle mb-2 text-muted">{category}</h6>  <p class="card-text">#{id}{description}</p>  <p class="card-text"><strong>${price}</strong></p>  </div>  </div> |

##### PARENT TO CHILD COMMUNCATION - USING CONTEXT

* In the parent child communication – we send the data to the using props . This becomes more difficult to manage when we have deep nesting of components.
* In this scenario – **We use context to pass on the data**.
* The Parent set the data in context using a key . The Child retrieve the value from the context using that key itself.

|  |  |
| --- | --- |
| PARENT.SVELTE  <script>  import {setContext} from 'svelte'  import Child from "./Child.svelte";  setContext("parentData",{parentName:'Alex'})  </script>  <Child/> | **CHILD.SVELTE**  <script>  import GrandChild1 from "./GrandChild1.svelte";  import GrandChild2 from "./GrandChild2.svelte";  import {getContext} from 'svelte'  let parentData = getContext("parentData")  </script>  <p> Data from Parent in Child : {parentData.parentName}</p>  <GrandChild1/>  <GrandChild2/>  **GRANDCHILD.SVELTE**  <script>  import {getContext} from 'svelte'  let parentData = getContext("parentData")  </script>  <p> Data from Parent in GrandChild1 : {parentData.parentName}</p> |

#### CHILD TO PARENT COMMUNICATION

##### EVENT FORWARDING

* Omitting the event handling function from an on: attribute is a shorthand to forward events up to the parent

component.

###### EVENT FORWARDING – DOM CLICK EVENT

* In the below example – Products.svelte is the child component of App.svelte. The “Product.svelte” delegates the event handling to its parent component by just saying on on:click.
* Now the parent must implement the onClick event handler.

|  |  |
| --- | --- |
| **App.svelte** | **Products.svelte** |
| <script>  import Products from './Products.svelte'  </script>  <div class="container">  <div class="row">  <Products productTitle="Book" on:click="{() => console.log("Button Clicked in Child")}"/>  </div>  </div> | <script>  export let productTitle;  </script>  <div class="card" style="width: 18rem;">  <div class="card-body">  <h5 class="card-title">{productTitle}</h5>  <h6 class="card-subtitle mb-2 text-muted">Card subtitle</h6>  <p class="card-text">Some quick example text to build on the card title and make up the bulk of the card's content.</p>  <button class="btn btn-primary" on:click>Add to Cart</button>  <button class="btn btn-primary">Delete</button>  </div>  </div> |

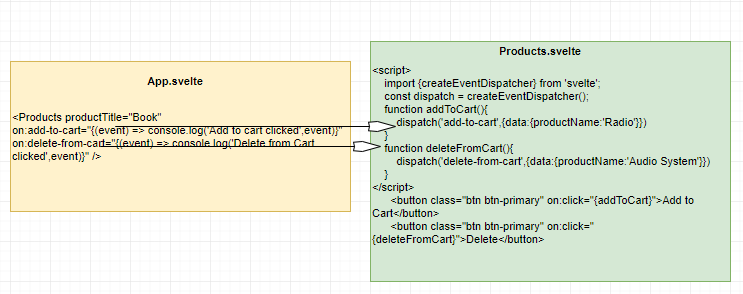
* **In this approach – We can have one issue – what if both button forward the event to parent. So, in both buttons click – it executes the same handler on the parent**
* These events only go to the parent component. They do not automatically bubble farther up the component in the hierarchy.

##### CHILD TO PARENT COMMUNCATION - USING CONTEXT

###### EVENT FORWARDING – CUSTOM EVENT

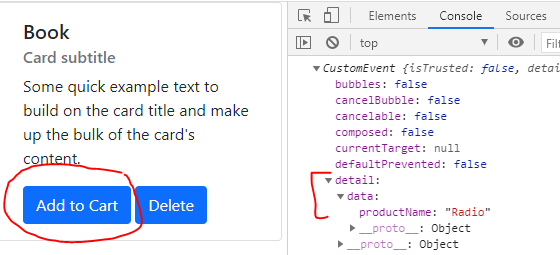
* The below example shows the way to forward the event to the parent
* This helps the child component to send the data to Parent component using dispatch() method

|  |
| --- |
| **App.svelte (Parent)** |
| <script>  import Products from './Products.svelte'  </script>  <div class="container">  <div class="row">  <Product on:add-to-cart={(event) => console.log(event.detail)}  on:delete-from-cart={(event) => console.log(event.detail)}/>  </div>  </div> |
| **Product.svelte (Child)** |
| <script>  import {createEventDispatcher} from 'svelte';  const dispatch = createEventDispatcher();  const products = [  { id: 'P0001', productName: "Audio System" },  { id: 'P0002', productName: "Video System" },  ];  function addToCart(productId){  const product = getProduct(productId);  dispatch('add-to-cart',product);  }  function deleteFromCart(productId){  const product = getProduct(productId);  dispatch('delete-from-cart',product);  }  function getProduct(productId){  return products.find((product) => product.id == productId);  }  </script>  {#each products as product, index}  <h3>{product.productName}</h3>  <p>#{product.id}</p>  <button on:click="{()=> addToCart(product.id)}">Add to Cart</button>  <button on:click={()=> deleteFromCart(product.id) }>Delete From Cart</button>  {/each} |

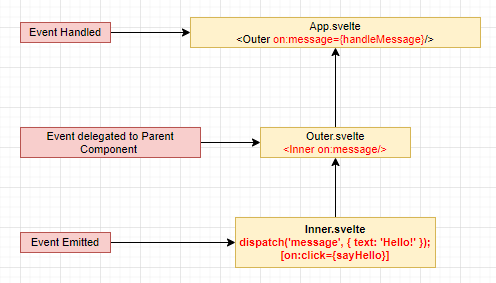


###### EXTRACTING VALUE

* The data sent by the child during event forwarding can be extracted the Parent in the “event” object (Screen shot below) i.e. event .detail property.



###### EVENT FORWARDING – EXAMPLE -2



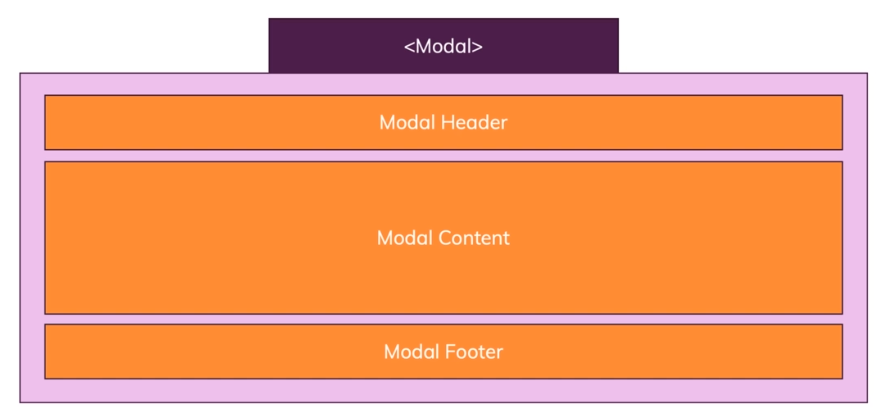
|  |  |  |
| --- | --- | --- |
| **App.svelte** | **Outer.svelte** | **Inner.svelte** |
| <script>  import Outer from './Outer.svelte';  function handleMessage(event) {  alert(event.detail.text);  }  </script>  <Outer on:message={handleMessage}/> | <script>  import Inner from './Inner.svelte';  </script>  <Inner on:message/> | <script>  import { createEventDispatcher } from 'svelte';  const dispatch = createEventDispatcher();  function sayHello() {  dispatch('message', {  text: 'Hello!'  });  }  </script>  <button on:click={sayHello}>  Click to say hello  </button> |

## CREATING REUSABLE UI COMPONENTS

### CUSTOM UI COMPONENT

### SLOTS

* The slots help us in creating reusable UI component.
* Slots help is creating UI component with same structure – but different content. It gives the placeholder for different content
* Below example – shows creating reusable modals



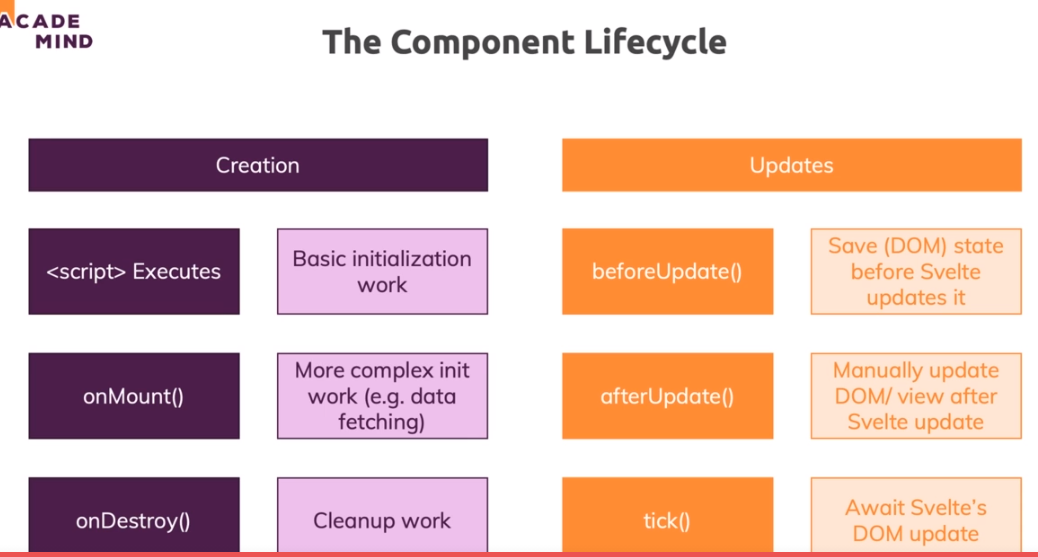
#### SIMPLE SLOTS

|  |  |
| --- | --- |
| App.svlete  <script>  import Box from './Box.svelte'  </script>  <Box>  <h1>Box Title</h1>  <p>Box Body</p>  </Box> | Box.svelte  <div>  **<slot>**  **<em>no content was provided</em>**  **</slot>**  </div>  \*Note – The red color content is the default content – which shows up when no content is provided to the Slot |

#### NAMED SLOTS

|  |  |
| --- | --- |
| **App.svelte**  <script>  import NamedSlot from './NamedSlot.svelte';  </script>  <NamedSlot>  <h1 slot="heading">This is Heading </h1>  <p slot="description">This is description</p>  </NamedSlot> | **NamedSlot.svelte**  <div>  <slot name="heading">  <em>No Heading Addeed yet !</em>  </slot>  <slot name="description">  <em>No description Addeed yet !</em>  </slot>  </div> |

## LIFE CYCLE HOOKS



### ON MOUNT

* The onMount() function schedules a callback to run as soon as the component has been mounted to the DOM.
* It must be called during the component's initialization (i.e not wrapping it around setTimeout function etc..)

#### ON MOUNT SYNTAX

|  |  |
| --- | --- |
| <script>  import { onMount } from 'svelte';  onMount(() => {  console.log('the component has mounted');  });  </script> | * If a function is returned from onMount, it will be called when the component is unmounted. * In this example the timeout will keep running even if the component is destroyed * Although this can be done in onDestroy() as well. * There are couple of more scenarios when the return – comes out to be very handy like   + Unsubscribing from a subscription for a store(check the store section for example),   + Aborting Http call when the component is unmounted. |
| <script>  import { onMount } from 'svelte';  onMount(() => {  const interval = setInterval(() => {  console.log('beep');  }, 1000);  return () => clearInterval(interval);  });  </script> |

## STORES

* Svelte stores help in data management or sharing between loosely coupled or disconnected components – specially between the component - which don’t have a parent-child relationship.
* Even if there is a parent- child relationship between components – event and pops way of sending data becomes cumbersome to manage. Svelte has “Stores” to manage and ease the communication of data between the components.
* Stores hold application state outside any component. They are an alternative to using props or context to make data available in components.
* **For stores that should be available to any component, define and export them in a file like src/stores.js and import them from that file wherever needed.**

### STORES TYPES

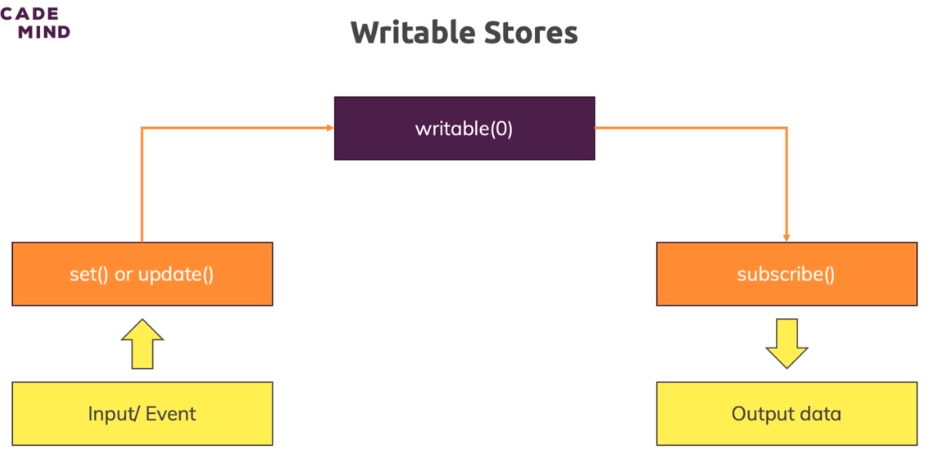
Svelte provides three kinds of stores.

|  |  |
| --- | --- |
| WRITABLE STORES | These are the only kind that can be modified by components. |
| READABLE STORES | These handles computing their own data |
| DERIVED STORES | These derive data from the current values of other stores. |

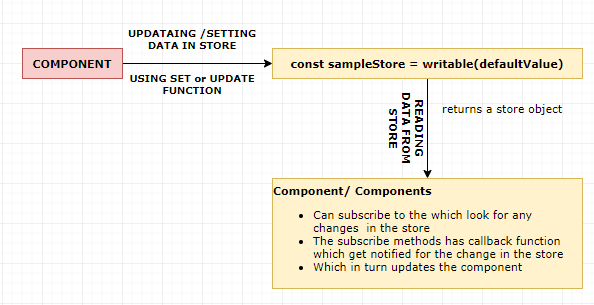
* All of these have a subscribe method that returns a function to call to unsubscribe.
* Custom stores can also be created. Their only requirement is to be an object with a properly implemented **Subscribe**() method.

### WRITABLE STORE

* Svelte provides build-in function to create to create stores – which help in maintain the state and communication between disconnected components.



* For a writable store - The component must subscribe for the store. So that it gets notified when the data is updated by another component.
* The component who all are subscribing for the store – The share the same copy of Store Object.
* The writable store has “set” and “update” function to update the store data.

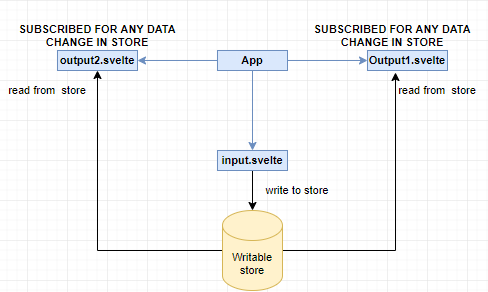


#### SYNTAX

|  |  |
| --- | --- |
| export const dataStore = writable(initialValue, async set => {  // Called when subscribe count goes from 0 to 1.  // Compute initial value and pass to set function.  return () => {  // Called when subscriber count goes to 0.  };  }); | * The writable store function takes 2 values * One is an initial value of the store and another is call back function, which get called when at least one subscription happens * The return gets called when the last subscribed component – unsubscribe for the subscription. * Writable stores has set() and update() methods to change the store. |

#### EXAMPLE 1- SET IN WRITABLE STORE

* In the diagram below we have 3 main disconnected component which has no parent – child relationship, to share the data between then we are using writable store.
* The input component writes data to the writable store
* All the output components are subscribed for any data change in the store and gets notified when data is updated on the store
* Once the product is added to cart – the Cart Components get notifies which in turn add items in the Cart.
* The “subscribe” method returns a unsubscribe which in-turn can be used to unsubscribe from the subscription in onDestroy() life cycle method.



|  |  |
| --- | --- |
| **APP.svelte** | **Input.svelte** |
| <script>  import Input from "./Input.svelte";  import Output1 from "./Output1.svelte";  import Output2 from "./Output2.svelte";  </script>  <main>  <Input/>  <Output1/>  <Output2/>  </main> | <script>  import valueStore from './value.store';  let name='';  function updateValue(){  valueStore.set(name);  }  </script>  INPUT VALUE : <input type="text" bind:value="{name}" on:input="{updateValue}">  <p>INPUT COMPONENT - {name}</p> |
| **Output1.svelte** | **Output2.svelte** |
| <script>  import { onMount } from 'svelte';  import valueStore from './value.store';  let \_value=''  onMount(() =>{  return valueStore.subscribe(value =>{  \_value = value;  });  })  </script>  <p>OUTPUT 1: {\_value} </p> | <script>  import { onMount } from 'svelte';  import valueStore from './value.store';  let \_value=''  onMount(() =>{  return valueStore.subscribe(value =>{  \_value = value;  });  })  </script>  <p>OUTPUT 2: {\_value} </p> |

#### EXAMPLE 2- HTTP CALL IN WRITABLE STORE

|  |
| --- |
| **hobbies.store.js**  import {writable} from 'svelte/store';  const hobbiesStore = writable([],async set =>{  const response = await fetch('https://svelte-app-d56b0-default-rtdb.firebaseio.com/hobbies.json');  const hobbies = await response.json();  set(hobbies); 🡨 SETTING VALUE TO STORE  return()=>{  console.log("Some Clean up task");  }  });  export default hobbiesStore; |
| **COMPONENT** : Hobbies.svelte [Subscriber]  <script>  **let hobbies= [];**  **import hobbiesStore from './hobbies.store.js';**  **onMount(()=>{**  **return hobbiesStore.subscribe(data=>{ hobbies = Object.values(data);});**  **})**  </script>  <h3>Hobbies List</h3>  <ul>  {#each hobbies as hobby}  <li>{hobby}</li>  {/each}  </ul> |

#### EXAMPLE 3- UPDATE IN WRITABLE STORE

|  |  |
| --- | --- |
| **App.svelte**  <script>  import Admission from "./Admission.svelte";  import NewStudent from "./NewStudent.svelte";  const studentList = [{  id: 1,  name: "Leanne Graham",  username: "Bret",  email: "Sincere@april.biz",  },  {  id: 2,  name: "Ervin Howell",  username: "Antonette",  email: "Shanna@melissa.tv",  },  {  id: 3,  name: "Clementine Bauch",  username: "Samantha",  email: "Nathan@yesenia.net",  },  ];  </script>  <div class="container">  <div class="row">  <div class="col-md-6">  <NewStudent studentList= {studentList} />  </div>  <div class="col-md-6">  <Admission />  </div>  </div>  </div | **NewStudent.svelte**  <script>  import schoolStore from './school.store.js';  export let studentList;  function admitStudent(student){  schoolStore.**update**(students => {  return [...students, { ...student }];  });  }  </script>  <div class="row">  <h3>New Student List</h3>  <table class="table table-striped">  <thead>  <tr>  <th scope="col">#</th>  <th scope="col">Name</th>  <th scope="col">Email</th>  <th scope="col">UserName</th>  <th scope="col">Action</th>  </tr>  </thead>  <tbody>  {#each studentList as student}  <tr>  <td>{student.id}</td>  <td>{student.name}</td>  <td>{student.email}</td>  <td>@{student.username}</td>  <td><button class="bt btn-primary" on:click="{()=>admitStudent(student)}">Add</button></td>  </tr>  {/each}  </tbody>  </table>  </div> |
| **Admission.svelte**  <script>  import { onDestroy } from "svelte";  import schoolStore from "./school.store.js";  onDestroy(() => {  unsubscribe();  });  function removeStudent(id){  schoolStore.update(students => {  return students.filter((student) => student.id != id);  })  }  </script>  {#if $schoolStore.length}  <h3>Admission</h3>  <table class="table table-striped">  <thead>  <tr>  <th scope="col">#</th>  <th scope="col">Name</th>  <th scope="col">Email</th>  <th scope="col">UserName</th>  <th scope="col">Action</th>  </tr>  </thead>  <tbody>  {#each $schoolStore as student}  <tr>  <td>{student.id}</td>  <td>{student.name}</td>  <td>{student.email}</td>  <td>@{student.username}</td>  <td><button on:click="{() => removeStudent(student.id)}">Remove</button></td>  </tr>  {/each}  </tbody>  </table>  {/if} | **school.store.js**  **import {writable} from 'svelte/store';**  **const schoolStore= writable([]);**  **export default schoolStore;** |
| **SCREEN SHOT** |
| **NOTES**   * All the components subscribed for the store should “unsubscribe” for it in onDestroy() method * We can change the store using “set” or “update” method |
| **AUTO SUBSCRIPTION**  **Svelte provides a shortcut to subscribe and unsubscribe for store. In the above example the subscriber can also be written as** | |
| <script>  import schoolStore from "./school.store.js";  </script>  {#if $schoolStore.length}  <h3>Admission</h3>  <table class="table table-striped">  <thead>  <tr>  <th scope="col">#</th>  <th scope="col">Name</th>  <th scope="col">Email</th>  <th scope="col">UserName</th>  </tr>  </thead>  <tbody>  {#each $schoolStore as student}  <tr>  <td>{student.id}</td>  <td>{student.name}</td>  <td>{student.email}</td>  <td>@{student.username}</td>  </tr>  {/each}  </tbody>  </table>  {/if} | * “$schoolStore” subscribe for the store and unsusbscribe as well when the component is unmounted from the DOM * There is no need of explicit un-subscription |

### READABLE STORE

* The readable stores are only read-only store. For which other components can subscribe.
* The question here is – if the data cannot be updated, then why components will subscribe to even?) .
* If the store is read-only, how the value in the store changes

|  |  |
| --- | --- |
| <script>  import sampleReadableStore from './sampleReadable.store';  let sportsList =[];  onMount(()=>{ return sampleReadableStore.subscribe((sports)=> sportsList = sports); })  </script>  <ul>  {#each sportsList as sport}  <li>{sport}</li>  {/each}  </ul> | |
| import {readable} from 'svelte/store';  const sampleReadableStore = readable([],(set) =>{  set(['Cricket','Swimming','Weight Lifting']);  return()=>{  console.log("Some Clean up task");  }  });  export default sampleReadableStore; | * The readable store function takes 2 values.One is an initial value of the store and another is call back function, which get called when at least one subscription happens * As it is a readable store – We cannot set the value from outside. The value can be set only from inside. * The readable store return function which can be used to do the clean up task . e,g clearing the interval (in below example) when the store is not in use. **This avoids the memory leaks** |
| **EXAMPLE 2: READABLE STORE – TICKING CLOCK** | |
| App.svelte  <script>  import { onMount } from "svelte";  import timeStore from './time.store.js'  let tickingTime;  onMount(() =>{  const unsubscribe = timeStore.subscribe((timeFromStore)=>{  tickingTime =timeFromStore;  });  return () =>{  unsubscribe();  }  }); </script>  <p>{tickingTime}</p> | Time.store.js  import {readable} from 'svelte/store';  const timeStore = readable([], set =>{  const timeInterval = setInterval(()=>{  const date = new Date();  set(`${date.getHours()}:${date.getMinutes()}:${date.getSeconds()}`);  },1000);  return ()=>{  clearInterval(timeInterval);  }  });  export default timeStore; |

### UNSUBSCRIBING FROM STORE SUBSCRIPTION

* When a component is subscribed for a store change. It should unsubscribe as well – when the component is detached from the DOM. Un-subscription is mandatory to avoid memory leaks.Un-Subscription can be done using :

|  |  |
| --- | --- |
| 1. ONDESTORY() LIFE CYCLE METHOD | **EXAMPLE**  let hobbies= [];  import hobbiesStore from './hobbies.store.js';  const unsubscribe = hobbiesStore.subscribe(data=>{  hobbies = Object.values(data);  })  onDestroy(() => {  if(!unsubscribe)  unsubscribe();  }); |
| 1. RETURN STATEMENT OF ONMOUNT() LIFE CYCLE METHOD. | **EXAMPLE**  onMount(()=>{  const unsubscribe = hobbiesStore.subscribe(data=>{  hobbies = Object.values(data);  });  return ()=>{  unsubscribe();  }  }); |

### AUTO SUBSCRIPTION

|  |  |
| --- | --- |
| **STORE**  import {writable} from 'svelte/store';  const hobbiesStore = writable([],async set =>{  const response = await fetch('https://svelte-app-d56b0-default-rtdb.firebaseio.com/hobbies.json');  const hobbies = await response.json();  let hub = Object.values(hobbies);  set(hub);  return()=>{  console.log("Some Clean up task");  }  });  export default hobbiesStore; | |
| <script>  import hobbiesStore from './hobbies.store'  </script>  <h3>Hobbies List</h3>  <ul>  {#each $hobbiesStore as hobby}  <li>{hobby}</li>  {/each}  </ul> | * $hobbiesStore – Any variable prefixed with “$“ – Svelte consider it as a Store variable. * For such variable svelte do the Auto subscription for that store and unsubscribe it too when the component is unmounted(removed from DOM) |

### CUSTOM STORES

|  |  |
| --- | --- |
| **App.svelte**  <script>  import customStore from './custom.store.js';  import Hobby from './Hobby.svelte';  let hobby;  function addHobbies(){  customStore.add(hobby);  hobby ='';  }  function resetHobbies(){  customStore.reset();  }  </script>  <input type="text" bind:value="{hobby}"/>  <button on:click="{addHobbies}">Add Hobby</button>  <button on:click="{resetHobbies}"> Reset</button>  <Hobby/> | **custom.store.js**  import {writable} from 'svelte/store';  const hobbies = writable([]);  const customStore= {  subscribe: hobbies.subscribe,  add:function(hobby){  hobbies.update(hobbies => {  return [...hobbies,hobby];  });  },  reset: function(){  hobbies.set([]);  }  }  export default customStore; |
| **Hobby.svlete**  <script>  import customStore from './custom.store.js';  </script>  {#if $customStore.length}  <p>Hobbies List </p>  <ul>  {#each $customStore as hobby}  <li>{hobby}</li>  {/each}  </ul>  {/if} | * In the above examples of stores- The components are **updating and setting** the stores * Ideally – rather than components calling the store functions (set & update) directly – Stores can in turn provide functions on abstract level to call store functions * This can be achieved using Custom Stores |

### STORE CONTRACT

You can create your own stores without relying on svelte/store, by implementing the store contract:

#### WHAT IS STORE CONTRACT?

* A store must contain a subscribe method, which must accept as its argument a subscription function. This subscription function must be immediately and synchronously called with the store's current value upon calling. subscribe. All a store's active subscription functions must later be synchronously called whenever the store's value changes.
* The subscribe method must return an unsubscribe function. Calling an unsubscribe function must stop its subscription, and its corresponding subscription function must not be called again by the store.
* A store may optionally contain a .set method, which must accept as its argument a new value for the store, and which synchronously calls all of the store's active subscription functions. Such a store is called a writable store.
* For interoperability with RxJS Observables, the. subscribe method is also allowed to return an object with an .unsubscribe method, rather than return the unsubscription function directly. Note however that unless .subscribe synchronously calls the subscription (which is not required by the Observable spec), Svelte will see the value of the store as undefined until it does.

#### CREATING CUSTOM STORE USING STORE CONTRACT

### CONTEXT STORE

|  |  |
| --- | --- |
| For stores that should only be available to descendants of a given component, define them in that component and  pass them to descendants using props or context. | import {setContext} from 'svelte';  import B from './B.svelte';  setContext('favorites', {color: 'yellow', number: 19});  import C from './C.svelte';  import {getContext} from 'svelte';  const {color, number} = getContext('favorites'); |

## HTTP INTERACTION

* Svelte has no inbuild HTTP client. We can use native JavaScript API or third part clients like “**axios**”

### FETCH API

* We will be setting Firebase for storing the data and fetch the data using firebase Api.
* If we want to load the initial data right after the component is loaded . It is recommended to call it in “onMount()” life cycle method.
* The HTTP calls can also be implemented using async and await

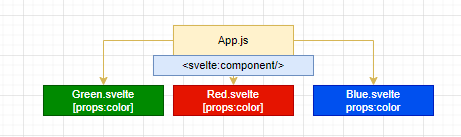
|  |
| --- |
| <script>  import { onMount } from "svelte";  export let hobbies =[];  let isDataLoading = false;  let hobbyInput;  function addHobby(){ 🡨 POST REQUEST  isDataLoading = true;  fetch('https://svelte-app-d56b0-default-rtdb.firebaseio.com/hobbies.json',{  method:'POST',  body:JSON.stringify(hobbyInput.value),  header: {  'Content-type':'application/json'  }  }).then((data)=>{  if(!data.ok){  return new Error("Network Error");  }  isDataLoading = false;  console.log(data);  hobbies = [...hobbies, hobbyInput.value];  hobbyInput.value = '';  }).catch(()=>{  isDataLoading = false;  console.log("Failed")  });    }  onMount(()=>{getHobbies()}); 🡨 Calling the GET in “onMount()” life cycle method.  function getHobbies(){🡨 GET REQUEST  fetch('https://svelte-app-d56b0-default-rtdb.firebaseio.com/hobbies.json').then((data)=>{  if(!data.ok){  return new Error("Network Error");  }  return data.json();  }).then((hobbiesRes)=>{  hobbies = Object.values(hobbiesRes);  console.log(hobbies);  }).catch((error)=>{  console.log("Failed",error)  });    }  </script>  <div class="container">  <div class="row mb-3">  <form>  <div class="mb-3">  <label for="hobby" class="form-label">Hobby</label>  <input type="text" class="form-control" id="hobby" bind:this ="{hobbyInput}" />  </div>  <button type="button" on:click="{addHobby}" class="btn btn-primary">Add Hobby</button>  </form>  </div>  <div class="row">  {#if hobbies.length >0}  <ul class="list-group">  {#each hobbies as hobby }  <li class="list-group-item">{hobby}</li>  {/each}  </ul>  {:else if hobbies.length == 0}  <div class="alert alert-danger" role="alert">  No hobbies Data Available  </div>  {:else if isDataLoading == true}  <div class="spinner-border text-primary" role="status">  <span class="visually-hidden">Loading...</span>  </div>  {/if}  </div>  </div> |

## SPECIAL ELEMENTS

### DYNAMIC COMPONENT

* Dynamic components are used when we want to render different components dynamically.
* Those components should have common props.
* Syntax : <svelte:component>

#### EXAMPLE



|  |  |
| --- | --- |
| **App.svelte**  <script>  import Blue from "./Blue.svelte";  import Green from "./Green.svelte";  import Red from "./Red.svelte";  const options =[  {component : Green,color:'green'},  {component : Red,color:'red'},  {component : Blue,color:'blue'},  ];  let selectedComponent =options[0];  $: console.log(selectedComponent)  </script>  <main>  <h1> Dynamic Component </h1>  <select bind:value="{selectedComponent}">  {#each options as option}  <option value="{option}">{option.color}</option>  {/each}  </select>  **<svelte:component**  **this={selectedComponent.component} color="{selectedComponent.color}"/>**  </main> | **Blue.svelte**  <script>  export let color;  </script>  <h1 style="color:{color}">Blue Component</h1> |
| **Green.svelte**  <script>  export let color;  </script>  <h1 style="color:{color}">Green Component</h1> |
| **Red.svelte**  <script>  export let color;  </script>  <h1 style="color:{color}">Red Component</h1> |
|  |

# SAPPER