스벨트 정리

## 선언적 렌더링

중괄호를 이용하여, 해당 값들이 변경되면 렌더링이 발생한다 .

<script>

let hello = "world"

</script>

<h1 on:click={()=> hello = "hokeys"}>hello {hello}</h1>

## 조건문과 반복문

스벨트에서는 HTML부분에서 조건문과 반복문을 통해서 컴포넌트를 조건 렌더 및 반복 렌더링 할 수 있다.

조건문 예시

<script>

let toggle = false;

</script>

<button on:click={() => toggle = !toggle}>Click</button>

{#if toggle}

True

{:else}

False

{/if}

중괄호를 작성 할 때 시작 중괄호는 #으로 시작하고, 중간에 들어가는 구문은 :을 사용한다. 구문을 종료 하기 위해 /를 사용해서 구문의 범위를 설정한다.

반복문 예시

<script>

let fruits = ["바나나","사과","딸기"]

</script>

{#each fruits as fruit}

<div>{fruit}</div>

{/each}

반복문을 사용하기 위해서는 each구문을 사용한다. Each구문의 사용 방식은 다음과 같다.

#each [iterable Object] as Element, index (component id)

/each

## 이벤트 핸들링

스벨트의 컴포넌트에서 이벤트에 연결하는 문법은 기존 이벤트 발생대상에서 on: 키워드를 붙여 주면 된다.

핸들링 예시

<script>

let value;

let toggle;

</script>

<button on:click={()=> toggle = !toggle}>Toggle</button>

<input type="text"

{value}

on:input={(e) => value = e.target.value}>

만약 양방향 바인딩 문법을 사용한다면 다음과 같이 작성할 수 있다 .

<script>

let value;

let toggle;

</script>

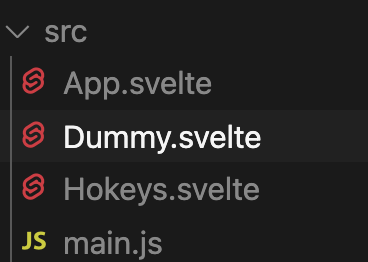
<button on:click={()=> toggle = !toggle}>Toggle</button>

<input type="text"

bind:value={value}>

## 컴포넌트

스벨트에서 컴포넌트는 파일명으로 export된다.



<script>

import Hokeys from './Hokeys.svelte'

import Dummy from './Dummy.svelte'

</script>

<Hokeys></Hokeys>

<Dummy></Dummy>

## Props 사용하기

컴포넌트에서 props를 사용하는 방법은 하위 컴포넌트에서 export 를 붙인 변수를 작성하는 것이다.

./Dummy.svelte 에서 props준비하기 예시

<script>

export let name;

</script>

<h1>{name}</h1>

./App.svelte 에서 name props전달하기

<script>

import Hokeys from './Hokeys.svelte'

import Dummy from './Dummy.svelte'

</script>

<Hokeys></Hokeys>

<Dummy name="hokeys!"></Dummy>

## Store 사용하기

스토어는 전역적으로 사용 할 수 있는 객체이다. 이 파트에서는 readable,derived 메서드는 다루지 않고, writable만 소개한다.

스토어는 js나 ts파일로 관리하는 것이 좋다. Svelte로 export 하려면 script type=”module” 이 필요하다.

읽기 쓰기 가능한 스토어는 svelte/store에서 import 할 수 있으며, import할 함수는 ‘writable’ 이다.

/store.js

import { writable } from 'svelte/store'

const \_address = "대구광역시"

export const addressStore = writable(\_address)

/App.js subscribe 및 자동구독 예제

<script>

import Hokeys from './Hokeys.svelte'

import Dummy from './Dummy.svelte'

import { addressStore } from './store';

let address;

addressStore.subscribe((add)=>{

address = add;

})

</script>

<Hokeys></Hokeys>

<Dummy name="hokeys!"></Dummy>

{$addressStore}

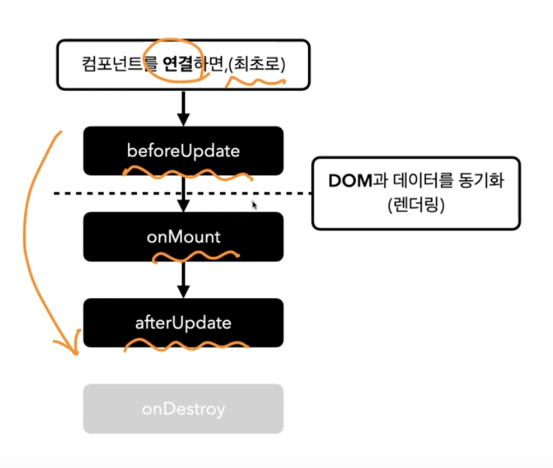
{address}

Store객체는 subscribe, set, update 메서드가 존재한다. 이중 subscribe는 해당 컴포넌트에서 store객체를 구독하여, 콜백의 첫번 째 인자로 store객체의 값이 전달 된다 . 이를 수동 구독이라고 하며, 이와 반대로 자동 구독은 $[store]를 사용하는 것으로 일반적인 변수처럼 할당, 읽기가 가능한 문법이다.

## Svelte Life Cycle

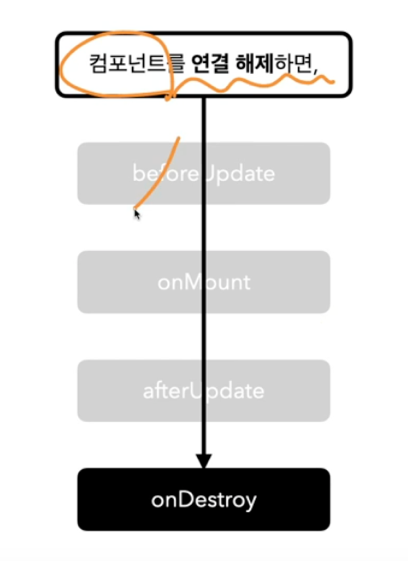
스벨트 라이프 사이클에는 “onMount, onDestroy, beforeUpdate, afterUpdate” 가 존재한다. 별개의 개념으로 Tick이 존재 한다.

컴포넌트 최초 마운트 시



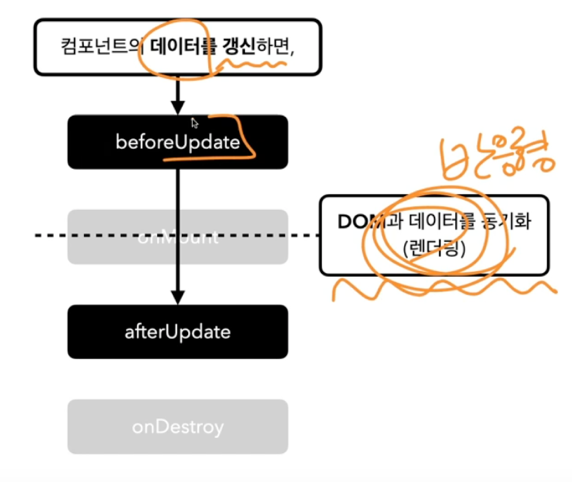
onMount 메서드는 componentDidMount와 그 뜻이 같아, 최초 렌더 직후 필요한 데이터를 받는 용도로 사용가능하다.

컴포넌트 연결 해제시



onDestroy 함수는 컴포넌트가 부모 요소와 연결 해제될 때 호출되는 함수로, 이는 componentWillUnMount 함수와 기능이 같아, 로그를 찍는 용도나 이전까지의 데이터를 서버로 전달하는 용도로 사용하기 적합하다.

컴포넌트 업데이트 시



beforeUpdate와 afterUpdate는 컴포넌트의 데이터가 변경될 때 호출된다. 그러므로, 이 함수 호출 내에서 컴포넌트의 업데이트를 발생시키는 행위는 금지되어야 한다.(이 두 함수는 컴포넌트의 데이터가 업데이트가 발생되고 호출 된다. )

beforeUpdate(()=>{

console.log(count) //동일

console.log(count) // 동일

})

afterUpdate(()=>{

console.log(count)//동일

})

## onMount, onDestroy

onMount와 onDestroy 함수는 각각 컴포넌트가 마운트되고, 연결 해제 되었을 때 호출되는 함수이다.

onMount, onDestroy는 첫번째 매개변수로 마운트 되고 실행시킬 콜백을 인자로 받는다. onMount의 첫번째 콜백 함수의 경우, callback내부에서 return하는 함수를 작성할 수 있는데, 이는 onDestroy 와 같이 컴포넌트가 연결 해제된 경우에 호출되는 함수이다. 이는 스벨트의 구 버전에서 onDestroy가 없었기 때문이다.

<script>

import { addressStore } from './store';

import { beforeUpdate,afterUpdate,onMount,onDestroy } from 'svelte'

onMount(()=>{

console.log("onMount !")

return ()=>{

console.log("Destroy")

}

})

onDestroy(()=>{

console.log("Destroy")

})

</script>

## Beforeupdate, afterUpdate

위의 라이프 사이클 업데이트 내용과 동일

## 라이프 사이클 모듈화

라이프 사이클 또한 js파일에서 모듈처럼 정의 가능하다.

/delayrender.js

import { onMount } from 'svelte';

import { writable } from 'svelte/store'

export const delayRender = (time = 3000) => {

let render = writable(false);

onMount(()=>{

setTimeout(()=>{

render.set(true);

},time);

})

return render;

}

/App.js

<script>

import { delayRender } from './delayRender';

let mounted = delayRender();

</script>

{#if $mounted}

<h1>Mounted !!</h1>

{:else}

<h1>Not Mounted</h1>

{/if}

주의해야 하는 점은 delayRender 내부에 스토어 객체를 사용했다는 점이다. 일반 불리언 값을 반환하게 된다면, 반응성을 가지지 않아 데이터가 변경되지 않는다. 그렇기에 스토어 자체를 반환하고, 이 변경되는 값을 통하여 반응성을 가지게 끔 한다.

## Tick

Tick은 화면이 렌더링이 끝난 다음까지 기다리다, Promise를 반환하는 함수이다. 이를 통해서, 데이터의 변화가 일어나기 전과 일어난 후의 데이터를 한 스코프 내에서 비교 할 수 있다.

/app.svelte

<script>

import { tick } from "svelte";

let count = 0;

async function handler() {

console.log(count); // 0

count += 1

await tick();

console.log(count) //1

}

</script>

<button on:click={handler}>Click</button>

{count}

Tick함수는 promise를 사용하기 때문에, async/await 패턴을 사용하는 것이 도움이 된다.

## 내용/속성/표현식 보간

내용 보간법

Html에 데이터를 보간하려면 {변수} 를 작성하면 된다.

<div>{count}</div>

속성 보간법

Html요소에 속성을 데이터로 변경할 경우, 데이터명과 속성이 동일하다면 {변수}, 라고 작성하고 아닌 경우에는 속성={변수} 라고 작성한다 .

<script>

let value = "hi"

let holder = "작성해주세요"

</script>

<input type="text" placeholder={holder} {value}>

표현식 보간법

{#if}{/if}구문을 사용해서 콘텐츠를 작성할 수 도 있지만, 삼항연산자로도 가능하다.

<script>

let count = 1;

</script>

{count > 0 ? "0보다 큽니다." : "0보다 작아요"}

## 원시 HTML / 데이터 디버깅

요소에 string으로 이루어진 html요소를 렌더링 하고 싶은 경우 {@html 스트링 변수}를 사용한다.

<script>

const h1 = "<h1>hello world</h1>"

</script>

{@html h1}

데이터를 디버깅 하고 싶은 경우, {@debug 변수, 변수}를 사용하면 된다. 다만 개발자 도구가 켜져있는 경우 계속해서 멈추기 때문에, 주의해서 사용하자.

{@debug count}

<h1 on:click={()=>count+=1}>{count}</h1>

## 데이터의 불변성

자바스크립트 원시타입 (bool, number,string,symbol,undefine,null)은 다른 변수에 변수를 할당한 경우 얕은 복사가 일어나, 원본 데이터가 변경되어도 반응을 가지지 않는다.

자바스크립트 객체타입 (object,set,map,array,function)은 다른 변수에 변수를 할당한 경우 깊은 복사가 일어나, 원본데이터가 변경되면 복사된 변수에도 똑같이 적용된다.

## 할당(Assignment)

스벨트는 특정 변수가 값이 변경될 때, 반응성을 가진다. 그러나, push , pop , split같은 메서드나 delete같은 키워드를 사용하게 되면, 반응성을 가지지 않는다.

<script>

let name = "hokeys";

let numbers = [1,2,3];

</script>

<!-- 반응성을 가진다. -->

<h1 on:click={()=>name = "lee jongho"}>{name}</h1>

<!-- 반응성을 가지지 않는다. -->

<button on:click={()=>numbers.push(numbers.length)}>PUSH !</button>

<ul>

{#each numbers as numb}

<li>{numb}</li>

{/each}

</ul>

그래서 이를 해결하기 위해서는 다음과 같이 할당이 발생해야 한다.

<!-- 반응성을 가진다. -->

<button

on:click={()=>{

numbers.push(numbers.length + 1)

numbers = numbers

}}

>PUSH !</button>

<ul>

{#each numbers as numb}

<li>{numb}</li>

{/each}

</ul>

그럼 다음의 경우는 어떠한지 확인해보자.

<script>

let name = "hokeys";

let numbers = [1,2,3];

let user = {

name,

numbers

}

</script>

<!-- 반응성을 가진다. -->

<h1 on:click={()=>user.name = "lee jongho"}>{user.name}</h1>

<!-- 반응성을 가진다. -->

<button

on:click={()=>{

user.name = "Hokeys"

user.numbers.push(user.numbers.length + 1)

//user또한 갱신 됨으로 본다.

}}

>PUSH !</button>

<ul>

{#each user.numbers as numb}

<li>{numb}</li>

{/each}

</ul>

조금 독특하게도, 변경이 발생한다. 이는 user의 객체가 변경되는 것으로 인지 한다. 이처럼 스벨트의 반응성을 가지는 행위에 대해서 깊은 이해가 있어야 잘 다룰 수 있다.

## 자바스크립트 Label구문의 이해

자바스크립트의 반복문에는 label이라는 것을 붙여, 이를 break로 멈출 수 있다. Ecma 2015에서는 함수에도 label을 붙여 사용가능하며, 이 사용 방법은 다음과 같다. [label Name] :

<script>

loop1:

for(let i=0; i < 10; i++){

console.log(i);

loop2 :

for(let u =0; u < 10; u++){

console.log(u);

break loop1

}

}

</script>

## 반응성 구문의 이해

스벨트에서는 반응성 구문이라는 것이 존재한다. 구문 이름으로 $를 표시하는 것이 특징이며, 내부에 반응성 데이터가 변경되면, 해당 구문이 실행된다.

<script>

let count = 0;

let double = 0;

const handler = () => count += 1;

$ : {

double = count \* 2;

console.log(double)

//count 변수가 변경될 때 마다 실행.

}

</script>

<button on:click={handler}></button>

이를 이용하여 반응성 변수라는 것을 만들 수 있다. 반응성 변수란. 어떤 변수가 반응성을 가질 때 마다, 그 값을 사용하여, 새로운 값을 만들어 내는 변수이다.

<script>

let count = 0;

let double = 0;

const handler = () => count += 1;

$ : {

double = count \* 2;

console.log(double)

//count 변수가 변경될 때 마다 실행.

}

$ : triple = count \* 3;

</script>

<button on:click={handler}></button>

{triple}

## 반응성 구문 패턴 정리

<script>

let count = 0;

let double = 0;

const handler = () => (count += 1);

//변수

$: triple = count \* 3;

//블록

$: {

double = count \* 2;

}

//함수 실행 !

$: count, console.log("hi");

//즉시 실행 함수

$: count,

(() => {

console.log("IIFE");

})();

//조건문

$: if (count > 10) console.log("if !");

//반복문

$: for (let i = 0; i < count; i++) console.log(i);

//조건문(Switch)

$: switch (count) {

case 0:

console.log("0");

default:

console.log("update");

}

//중첩 함수

$: {

function scope1() {

function scope2() {

function scope3() {

console.log(count);

}

scope3()

}

scope2()

}

scope1()

}

</script>

<button on:click={handler} />

{triple}

## 클래스와 스타일 속성 바인딩

기존에 Html 요소의 클래스를 변경하기 위해서는 삼항연산자를 사용하여, 클래스를 스위치하고는 했다. 그러나 스벨트에는 class: 접두어가 존재하여, 반응성 변수에 따라 특정 클래스를 ON/OFF 할 수 있다.

<script>

let active = false;

let letterSpacing = "letter-spacing : 5px;"

let color = {

t : "tomato",

w : "#fff"

}

</script>

<h1 on:click={()=>active = !active}>

active !

</h1>

<div class={active ? "active" : ""}>

기존 방식

</div>

<div class:active={active}>

새로운 방식

</div>

<div style="color : {color.w}; background-color : {color.t} {letterSpacing}">

스타일 속성 바인딩

</div>

<style>

div{

width: 50px;

height : 50px;

background-color: red;

}

.active{

width: 100px;

height : 100px;

background-color: antiquewhite;

}

</style>

## 클래스 바인딩 패턴 정리

클래스의 class 접두어를 사용하는 패턴을 이용하여, 양방향 바인딩처럼 적용하고자 하는 클래스명과 데이터명이 같다면 단축할 수 있다. 또한 함수를 통해서 클래스를 정의 할 수 있다.

<script>

let active = false;

function multi(){

return "active"

}

</script>

<h1 on:click={()=>active = !active}>

active !

</h1>

<div class:active>

단축 !

</div>

<div class={multi()}>

함수를 이용한 클래스 정의

</div>

<style>

div{

width: 50px;

height : 50px;

background-color: red;

}

.active{

width: 100px;

height : 100px;

background-color: antiquewhite;

}

</style>

## 스타일의 유효범위와 전역화

스벨트의 컴포넌트에서 정의한 스타일은 해당 컴포넌트에서만 유효하다. 만약 이를 전역화 하고자 한다면, 다음과 같이 :global() 옵션으로 선택자를 감싸야 한다.

<style>

div{

width: 100px;

height: 100px;

background-color: aqua;

}

:global(.active){

width: 500px;

height: 500px;

background-color: red;

}

</style>

## Key frame의 전역화

키프레임은 @keyframe [name]{} 으로 정의 내린다. 그렇기 때문에, 선택자를 인수로 받는 :global은 사용할 수 없다. 이를 위해 keyframe을 전역화 하기 위해서는, -global- 이라는 예약어를 사용하는데, 예시는 다음과 같다.

<style>

div{

width: 100px;

height: 100px;

background-color: aqua;

}

:global(.active){

width: 500px;

height: 500px;

background-color: red;

animation : zoom .4s infinite alternate;

}

@keyframes -global-zoom{

from{

transfrom : scale(1);

}

to{

transfrom : scale(1.5);

}

}

</style>

## 일반 요소 바인딩

특정 HTML 요소를 포커스하거나, 조작하기 위해서 리액트에서는 useRef라는 기능이 존재한다. 스벨트에서는 이를 bind:this 키워드로 html요소를 데이터에 바인딩 가능하다.

<script>

let bindInput;

let handler = () => {

bindInput.focus();

}

</script>

<button on:click={handler}>

Click !

</button>

<input type="text" bind:this={bindInput}>

## 입력 요소 바인딩 패턴 정리

기존에 봤던 패턴처럼 bind:value 형태를 작성하게 되면, 해당하는 html 프로퍼티와 데이터가 양방향 바인딩 된다. 이를 응용하여, 다양한 형태의 input에 응용 가능하다.

<script>

let text = "";

let numb = 3;

let checked = false;

let fruits = ["사과", "바나나", "체리"];

let selectedFruits = [];

let selectedFruit = "";

let select = "";

let selectMulti = []

</script>

<div>

<h1>text</h1>

<input type="text" bind:value={text} />

</div>

<div>

<h1>Number/Range</h1>

<input type="number" bind:value={numb} min="0" max="10" />

<br />

<input type="range" bind:value={numb} min="0" max="10" />

</div>

<div>

<h1>checkbox</h1>

<input type="checkbox" bind:checked />Agree

<input type="checkbox" bind:checked />AgreeAll

</div>

<div>

<h2>다중 선택</h2>

<strong>selected :{selectedFruits}</strong>

{#each fruits as fruit}

<label>

<input type="checkbox" value={fruit} bind:group={selectedFruits} />

{fruit}

</label>

{/each}

</div>

<div>

<h1>Radio</h1>

<strong>Selected : {selectedFruit}</strong>

{#each fruits as fruit}

<label>

<input type="radio" value={fruit} bind:group={selectedFruit} />

{fruit}

</label>

{/each}

</div>

<div>

<h1>Textarea</h1>

<pre>{text}</pre>

<textarea bind:value={text} />

</div>

<div>

<h1>Select</h1>

<strong>Select : {select}</strong>

<select bind:value={select}>

<option disabled >Please select one</option>

{#each fruits as fruit}

<option>{fruit}</option>

{/each}

</select>

</div>

<div>

<h1>Select Multi</h1>

<strong>Select : {selectMulti}</strong>

<select multiple bind:value={selectMulti}>

<option disabled >Please select multi</option>

{#each fruits as fruit}

<option>{fruit}</option>

{/each}

</select>

</div>

## 편집 가능 요소 바인딩

Div 요소에 contenteditable 속성을 true로 활성화 하면 클릭시 해당 콘텐트를 편집 가능하게 끔 변경 된다.

<script>

let innerHtml = "Hello world"

let textContent = ""

$ : console.log(innerHtml)

</script>

<div

contenteditable

bind:textContent={textContent}

bind:innerHTML={innerHtml}

>

hello world

</div>

<div>

{innerHtml}

</div>

<div>

{textContent}

</div>

<style>

div{

height: 100px ;

border : 1px solid red;

}

</style>

## 조건 블록 패턴 정리

조건 블록은 if , else, else if 다중 if문을 사용 할 수 있다.

<script>

let toggle\_1 = false;

let toggle\_2 = true;

</script>

{#if toggle\_1}

toggle\_1 = true;

{:else}

toggle\_1 = false;

{/if}

{#if toggle\_1}

toggle\_1 = true;

{:else if toggle\_2}

toggle\_2 = true;

{:else}

toggle\_1 = false;

{/if}

{#if toggle\_1}

toggle\_1 = true;

{:else if toggle\_2}

toggle\_2 = true;

{/if}

## 반복 블럭의 key사용

리액트에서도 반복문을 사용하여, 동적으로 컴포넌트를 생성시, key라는 것을 입력해준다. 스벨트에서도 key의 개념이 존재하고, 이 key를 반복 블럭 내에서 명시 하지 않으면, 데이터가 변경 시, 새롭게 모두 렌더링된다. 사용 방법은 {#each [iterable] as element (key)}형태로 작성하면 된다.

<script>

let fruits = ["사과","바나나","체리","오랜지"]

</script>

<button on:click={()=>{

fruits = fruits.slice(1);

}}>

삭제

</button>

<ul>

{#each fruits as fruit (fruit)}

<li>{fruit}</li>

{/each}

</ul>

## 반복 블럭 패턴 정리

<script>

let fruits = [

{id : 1, name : "orange"},

{id : 2, name : "cherry"},

{id : 3, name : "apple"},

{id : 4, name : "mango"},

]

let matrix = [

[1,"orange"],

[2,"cherry"],

[3,"apple"],

[4,"mango"],

]

let user = {

id : 1,

name : "lee",

email : "substantiation"

}

</script>

<h1>기본 형태</h1>

{#each fruits as fruit (fruit.id)}

<div>{fruit.name}</div>

{/each}

<h1>구조 분해</h1>

{#each fruits as {id,name} (id)}

<div>{name}</div>

{/each}

<h1>2차원 배열</h1>

{#each matrix as [id,name] (id)}

<div>{name}</div>

{/each}

<h1>나머지 연산자</h1>

{#each fruits as {id,...rest} (id)}

<div>{rest.name}</div>

{/each}

<h1>객체 데이터</h1>

{#each Object.entries(user) as [key,value] (key)}

<div>{key} : {value}</div>

{/each}

## Key 블럭

Key 블럭은 연결된 데이터의 값이 변경될 때 마다 내용을 파괴하고 다시 생성하는 블럭이다.

<script>

import Count from "./Count.svelte"

let reset = false;

</script>

{#key reset}

<Count></Count>

{/key}

<button on:click={()=>{

reset = !reset

}}>

reset

</button>

## 자바스크립트 비동기 처리의 이해와 사용 패턴 정리 – 1

Promise와 Callback의 사용법

## 자바스크립트 비동기 처리의 이해와 사용 패턴 정리 -2

Async와 try catch final 사용법

## Await 블럭

Javascript의 Promise 는 Pending(대기중), Success(성공), Reject(에러) 3가지 상태를 가질 수 있다.

<script>

let isError = false;

function getName(){

// 펜딩 !

return new Promise((resolve,reject)=>{

if(!isError){

//성공

resolve("Hokeys")

}else{

//에러

reject(new Error("에러 발생 !"))

}

})

}

</script>

이를 응용하여, 스벨트에서는 프라미스의 3가지 상태에 맞게, 렌더링을 다르게 해줄 수 있다.

<script>

let isError = true;

// 초기 값

let promise = Promise.resolve("");

function getName(){

// 펜딩 !

return new Promise((resolve,reject)=>{

setTimeout(()=>{

if(!isError){

//성공

resolve("Hokeys")

}else{

//에러

reject(new Error("에러 발생 !"))

}

},3000)

})

}

</script>

<button on:click={()=>{

promise = getName()

}}>

Click !

</button>

{#await promise}

<p>로딩중 !</p>

{:then name}

{name}

{:catch error}

{error.message}

{/await}

다만 async/await 패턴으로 작성하게 되면 reolve혹은 reject의 값이 할당 되기 까지 진행이 되지 않기 때문에, pendding이 존재하지 않게 된다.

## 인라인 이벤트 핸들러

인라인 이벤트 핸들러란. 정의되어 있는 함수를 이벤트에 연결하는 것이 아니라, 이벤트 함수에서 콜백 함수를 정의하는 것을 의미한다. 다른 프레임워크나 라이브러리에서는 권장하지 않지만, 스벨트에서는 권장하는 문법이다.

<script>

let isError = true;

// 초기 값

</script>

<h1>is Erorr ? : {isError}</h1>

<button on:click={()=>{

isError = !isError;

}}>

Click !

</button>

## 다중 이벤트 핸들러

다중 이벤트 핸들러란 ? 동일한 이벤트에 여러 이벤트를 발생 시키는 방법을 의미한다.

<script>

let isError = true;

// 초기 값

</script>

<h1>is Erorr ? : {isError}</h1>

<button

on:click={() => {

isError = !isError;

console.log("무한 !!")

}}

on:click={()=>{console.log("무야호 !")}}

>

Click !

</button>

이 경우 가장 먼저 정의되어 있는 이벤트부터 실행된다.

## 이벤트 수식어

이벤트를 제어하는 대표적인 요소로는 stopPropagation과 preventDefault등이 존재한다. 이는 콜백에서 이벤트 인자를 받아, 동작을 막아야하지만, 이벤트 함수가 event객체로 오염될 수 있다. 이를 방지 하기 위해서 스벨트에서는 다양한 이벤트 수식어를 제공하고 있다 .

preventDefault

설명 : 없음

<script>

let naver = "https://naver.com";

</script>

<div>

<a href={naver}

on:click|preventDefault={()=>{

console.log("기존 이벤트가 더 이상 진행되지 않음")

}}

>

네이버

</a>

</div>

once

설명 : 한번만 동작하고, 해당 이벤트가 삭제되는 옵션이다.

<div>

<button on:click|once={()=>{

console.log("한번만 출력 !")

}}>

Once

</button>

</div>

StopPrapogation

설명 : 이벤트 버블링을 방지 하는 옵션이다. Ex : 하위 컴포넌트가 클릭되었는데 상위 컴포넌트의 이벤트가 같이 발생 .

<script>

let naver = "https://naver.com";

</script>

<div

class="big"

on:click={() => {

console.log("big ! ");

}}

>

<div

class="small"

on:click|stopPropagation={() => {

console.log("small ! ");

}}

/>

</div>

<style>

.big {

width: 100px;

height: 100px;

background-color: red;

}

.small {

width: 50px;

height: 50px;

background-color: blue;

}

</style>

Capture

이벤트 버블링과 반대로 상위 컴포넌트에서 하위컴포넌트로 이벤트가 전달 되는 방식이다.

<script>

let naver = "https://naver.com";

</script>

<div

class="big"

on:click|capture={() => {

console.log("big ! ");

}}

>

<div

class="small"

on:click={() => {

console.log("small ! ");

}}

/>

</div>

<style>

.big {

width: 100px;

height: 100px;

background-color: red;

}

.small {

width: 50px;

height: 50px;

background-color: blue;

}

</style>

Self

설명 : event의 target과 currentTarget이 일치해야만 발생하는 이벤트 옵션이다.

<script>

let naver = "https://naver.com";

</script>

<div

class="big"

on:click|self={() => {

console.log("big ! ");

}}

>

<div

class="small"

on:click={() => {

console.log("small ! ");

}}

/>

</div>

<style>

.big {

width: 100px;

height: 100px;

background-color: red;

}

.small {

width: 50px;

height: 50px;

background-color: blue;

}

</style>

Passive

설명 : 스크롤 시 이벤트가 계속 발생한다면, 오버헤드가 많이 발생하여 성능에 영향을 준다. 이를 위해 Passive 옵션이 존재하는데, 이 옵션을 true로 설정한다면 이벤트가 끝나지 않아도, 기본 속도로 스크롤을 실행한다.

<script>

let naver = "https://naver.com";

</script>

<div

class="big"

on:wheel|passive={(e)=>console.log(e)}

>

<div

class="small"

on:click={() => {

console.log("small ! ");

}}

/>

</div>

<style>

.big {

width: 100px;

height: 100px;

background-color: red;

overflow-y: auto;

}

.small {

width: 50px;

height: 2050px;

background-color: blue;

}

</style>

## 컴포넌트 개요 (컴포넌트 바인딩)

기존에 HTML 요소를 바인딩 하기 위해서는 bind:this={변수} 로 변수에 할당하였다. 스벨트 컴포넌트 또한 동일하게, bind:this로 부모요소에서 하위 컴포넌트를 변수에 할당 할 수 있다. 그러나, 단순히 하위 컴포넌트를 바인드 하였다고 하여도, 특별한 방법 없이는 하위 컴포넌트의 프롭스에 접근 할 수 없다.

<script>

import Count from './Count.svelte';

import { onMount } from 'svelte'

let countRef;

onMount(()=>{

console.log(countRef)

})

</script>

<Count bind:this={countRef}></Count>

countRef의 내용

텍스트이(가) 표시된 사진

자동 생성된 설명

## 부모에서 자식으로

Props를 전달하는 방법에는 여러 방법이 존재한다.

상수 값으로 직접 전달하기 .

<script>

import Profile from './Count.svelte';

import { onMount } from 'svelte'

let user = {

name : "hokey",

age : 28,

mail : "substantiation@naver.com"

}

</script>

<Profile

name={user.name}

age={user.age}

mail={user.mail}

/>

단축 문법

Props명과 전달하고자 하는 변수의 이름이 같은 경우 사용 가능

<script>

import Profile from './Count.svelte';

import { onMount } from 'svelte'

let user = {

name : "hokey",

age : 28,

mail : "substantiation@naver.com"

}

const {name ,age, mail} = user;

</script>

<Profile

{name}

{age}

{mail}

/>

전개 연산자

<script>

import Profile from './Count.svelte';

let user = {

name : "hokey",

age : 28,

mail : "substantiation@naver.com"

}

</script>

<Profile

{...user}

/>

## Props 양방향 바인딩

React에서는 단방향 바인딩이기 때문에, 하위 컴포넌트로 전달해준 콜백 함수를 이용하여, 반응성을 만들었다. 그러나 스벨트는 양방향 바인딩을 지원하기 때문에, 하위 컴포넌트에서 양방향 바인딩된 변수를 직접 변경 할 수 있다.

Todo.svelte

<script>

export let index;

export let title;

export let done;

export let todos;

function buttonHandler(){

todos.splice(index,1);

todos = todos

}

</script>

<div>

<h1>

<input type="checkbox"

bind:checked={done} />

{index} : {title}

<button on:click={buttonHandler}>X</button>

</h1>

</div>

App.svelte

<script>

import Todo from "./Todos.svelte";

let todos = [

{ id: 1, title: "아침", done: false },

{ id: 2, title: "점심", done: false },

{ id: 3, title: "저녁", done: false },

];

</script>

{#each todos as { id, title, done }, index (id)}

<Todo bind:todos {index} {title} bind:done />

{/each}

{#each todos as { id, title, done }, index (id)}

<Todo bind:todos {index} {title} {done} />

{/each}

## Event dispatcher

## 스벨트에서는 하위컴포넌트에서 이벤트를 발생시켜, 상위 컴포넌트에서 연결된 콜백을 실행 시킬 수 있다. customEvent는 createEventDispatcher로 발생시킬 수 있으며, createEventDispatcher함수를 호출 시켜 리턴 받은 dispatch에는 (EventName : string, detail : any) 인자가 존재한다.

./Dummy.js

<script>

import { createEventDispatcher } from 'svelte';

const dispatch = createEventDispatcher();

function handler(){

dispatch('eventOccured',"hello")

}

</script>

<button on:click={handler}>

Click !

</button>

./App.js

<script>

import Dummy from './Dummy.svelte';

function awsomeEventHandler(event){

console.log(event.detail)

}

</script>

<Dummy on:eventOccured={awsomeEventHandler}/>

## 이벤트 포워딩

이벤트 포워딩이란, 어떤 컴포넌트가 부모 – 자식 – 손자의 형태를 하고 있을 때, 손자 컴포넌트에서 발생한 이벤트를 자식 컴보넌트에서 처리하지 않은 경우, 이 이벤트를 처리하고자 하는 컴포넌트에서 이벤트를 받아 처리하는 것을 의미한다.

/App.svelte

<script>

import Dummy from './Dummy.svelte';

function awsomeEventHandler(event){

console.log(event.detail)

}

</script>

<Dummy on:eventOccured={awsomeEventHandler}

on:clickOccured={awsomeEventHandler}

/>

./Dummy.svelte

<script>

import { createEventDispatcher } from 'svelte';

import Child from './Child.svelte';

const dispatch = createEventDispatcher();

function handler(){

dispatch('eventOccured',"hello")

}

</script>

<button on:click={handler}>

Click !

</button>

<Child on:clickOccured></Child>

./Child.svelte

<script>

import { createEventDispatcher } from 'svelte';

const dispatch = createEventDispatcher();

function handler(){

dispatch("clickOccured","i'm Child component");

}

</script>

<button on:click={handler}>

Child Click

</button>

## GetContext, setContetxt

React의 contextAPI와 달리 svelte의 contextAPI는 전역적으로 쓰이는 것이 아닌, 하위 컴포넌트로만 전파 해줄 수 있는 특성을 가지고 있다. contextAPI를 호출한 부모컴포넌트나 형제컴포넌트는 context를 공유할 수 없다.

/app.js

<script>

import Dummy from './Dummy.svelte';

import { setContext } from 'svelte';

function awsomeEventHandler(event){

console.log(event.detail)

}

setContext("pm",10000);

</script>

<Dummy on:eventOccured={awsomeEventHandler}

on:clickOccured={awsomeEventHandler}

/>

./Dummy 에서는 호출하지 않음

./Child.svelte

<script>

import { createEventDispatcher,getContext } from 'svelte';

const dispatch = createEventDispatcher();

const pm = getContext("pm");

console.log(pm);

function handler(){

dispatch("clickOccured","i'm Child component");

}

</script>

<button on:click={handler}>

Child Click

</button>

## 모듈 상황에서 데이터 정의

스벨트에서는 독특하게도, 같은 컴포넌트 종류에서 사용 할 수 있는 컨텍스트가 존재한다. 이를 Module Context라고 한다. 대신 이 모듈 컨텍스트에서 사용된 값은 반응성을 가질 수 없다.

현재 사용중인 script부분은 인스턴스 내부에서만 유효한 값이다.

일반 script 예제

./Dummy.svelte

<script>

export let name;

let count = 0;

$ : console.log(count)

</script>

<h1 on:click={()=>{

count += 1

}}>{name}</h1>

Count는 공유되는 값이 아니기 때문에 각각이 컴포넌트 클릭 수에 따라 다르게 동작한다.

모듈 스크립트 예제

<script context="module">

let count = 0;

</script>

<script>

export let name;

// $ : console.log(count) 안됨

</script>

<h1 on:click={()=>{

count += 1

console.log(count)

}}>{name}</h1>

해당 카운트는 공유되기 때문에, 어떤 이름을 눌러도 count가 1씩 증가한다.

## 오디오 플레이어 예제

스킵

## $$props와 $$restProps

$$props란 컴포넌트에 내장된 객체이며, 부모로부터 전달되는 props를 모두 관리하고 있는 객체이다.

./App.js

<script>

import Child from './Child.svelte';

</script>

<Child

value="hi"

name="hokeys"

mail="substantiation@naver.com"

/>

./Child.svelte

<script>

console.log($$props)

</script>

텍스트이(가) 표시된 사진

자동 생성된 설명

$$restProps란 자식 컴포넌트에서 export 하여, 명시적으로 props를 정의하지 않은 프롭스에 한하여, 관리하는 객체이다.

./App.svlete

동일

./Child.svelte

<script>

export let name;

export let value;

console.log($$props)

console.log($$restProps)

</script>

텍스트이(가) 표시된 사진

자동 생성된 설명

## 단일 슬롯과 Fallback content

슬롯이란 react.children처럼 컴포넌트 태그 빈공간에 컴포넌트나, 엘리먼트들을 삽입하여, 해당 컴포넌트에서 전달 받은 컴포넌트들을 사용 할 수 있는 것을 말한다.

Fallback content란 슬롯이 존재하기는 하나, 컴포넌트나 요소들이 전달 되지 않은 경우, 대체할 수 있는 컨텐츠를 말한다.

./App.svelte

<script>

import Child from './Child.svelte';

</script>

<Child>

Click Me

</Child>

<Child />

<!-- 슬롯 없음 -->

./Child.svelte

<script>

</script>

<button>

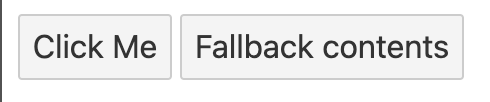
<slot>

Fallback contents

</slot>

</button>

결과



## 이름을 가지는 슬롯

위의 예제처럼 슬롯이 한개이며, 엘리먼트 요소가 한개의 부모요소를 가진다면, 단일 슬롯을 사용하는 경우, 관리하기 매우 편하지만 여러 요소를 슬롯으로 전달하려면 불편함이 존재할 것이다. 이를 위해 슬롯에는 고유한 이름이 존재하고, 전달하는 요소에 슬롯의 이름을 명시하면 관리가 용이하다.

./Child.svelte

<script>

</script>

<div class="card">

<slot name="name">

<h1>"이름 없음"</h1>

</slot>

<slot name="mail">

<h2>"메일 없음"</h2>

</slot>

</div>

<style>

.card{

box-sizing: border-box;

padding: 5px;

margin-top: 20px;

border: 1px solid black;

border-radius: 10px;

min-height: 10px;

box-shadow: rgba(0, 0, 0, 0.482) 5px 5px 5px 5px;

}

</style>

./App.svelte

<script>

import Child from './Child.svelte';

</script>

<Child>

<h1 slot="name">

이종호

</h1>

<h2 slot="mail">

substantiation@naver.com

</h2>

</Child>

<Child />

<!-- 슬롯 없음 -->

하위 컴포넌트에 있는 slot의 name은 상위 컴포넌트에서 전달하려는 엘리먼트의 slot 프로퍼티와 일치한다. (만약 전달 하려는 엘리먼트의 순서를 임의로 변경하더라도, 슬롯에서 순서는 동일하다.)

## 범위를 가지는 슬롯

슬롯에는 독특한 기능이 존재하는데, 슬롯으로 전달되는 엘리먼트에서만 사용가능 값을 전달 해줄 수 있다. 이를 범위를 가지는 슬롯이라고 한다 .

./App.js

<script>

import Child from './Child.svelte';

</script>

<Child>

<h1 slot="name">

이종호

</h1>

<h2 slot="mail" let:domain>

substantiation{domain}

</h2>

</Child>

<Child />

<!-- 슬롯 없음 -->

./Child.svelte

<script>

const domain = "@naver.com"

</script>

<div class="card">

<slot name="name">

<h1>"이름 없음"</h1>

</slot>

<slot name="mail" {domain}>

<h2>"메일 없음"</h2>

</slot>

</div>

<style>

.card{

box-sizing: border-box;

padding: 5px;

margin-top: 20px;

border: 1px solid black;

border-radius: 10px;

min-height: 10px;

box-shadow: rgba(0, 0, 0, 0.482) 5px 5px 5px 5px;

}

</style>

이를 응용하면, 클로져를 만들어 내어, 이벤트 콜백을 만들어 낼 수 있다.

./App.svelte

<script>

import Child from './Child.svelte';

let dummy = "";

$ : console.log(dummy)

</script>

<h1>

{dummy}

</h1>

<Child>

<h1 slot="name">

이종호

</h1>

<h2 slot="mail" let:domain

on:click={()=>{

dummy = domain

console.log(dummy)

}}

>

substantiation{domain}

</h2>

</Child>

<Child />

<!-- 슬롯 없음 -->

## 슬롯 포워딩

슬롯 포워딩이란 이벤트 포워딩과 마찬가지로, 부모 – 자식 – 손자 구조를 가지는 컴포넌트에서, 중간 컴포넌트인 자식 컴포넌트가 슬롯을 처리하지 않은 경우, 명시적으 손자 컴포넌트로 부모로부터 전달 받은 슬롯을 전달하는 개념이다.

./App.svelte

<script>

import Parent from './Parent.svelte'

</script>

<Parent>

<h1 slot="name">hokeys</h1>

<h1 slot="age">28</h1>

<h1 slot="mail">substantiation@naver.com</h1>

</Parent>

./Parent.svelte

<script>

import Child from './Child.svelte'

</script>

<Child>

<slot name="name" slot="name"/>

<slot name="age" slot="age"/>

<slot name="mail" slot="mail"/>

</Child>

./Child.svelte

<script>

</script>

<slot name="name"></slot>

<slot name="age"></slot>

<slot name="mail"></slot>

## $$slot

$$slot 객체는 부모 컴포넌트로부터 전달 받은 슬롯의 유무를 관리하는 객체이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

./Parent.svelte에서 console.log($$svlete)를 실행한 결과

부모 컴포넌트에서 slot을 제거하면 아예 해당 슬롯이름을 가진 키가 삭제된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이를 응용하면, 슬롯의 유무에 따라, 특정 엘리먼트를 조정 할 수 있다.

## Store 수동 구독과 자동 구독

Store는 전역적으로 컴포넌트 간 데이터를 전송하는 역할을 하기 때문에, .svlete 포맷파일이 아닌 .js나 .ts에서 관리해야 편리하다. Store객체는 “$” 를 사용하여, .svelte 파일에서 편리하게 사용 가능하지만, .js나 .ts에서는 $사인을 사용 할 수 없다. 이러한 특성 때문에, store객체는 수동 구독 방법과 자동 구독 방법을 지원한다.

수동 구독 기본 예제

./store.js

import { writable } from 'svelte/store'

export const countStore = writable(0,()=>{

console.log("count 구독자가 1명 이상일 때 ")

console.log("누군가가 구독을 시작 하였습니다.")

return ()=>{

console.log("1명이상의 구독자에서 0명의 구독자가 되었을 때 실행")

}

})

writable함수를 호출하면, 스토어 객체가 생성된다. 이 스토어 객체의 2번째 인자로는 0개의 구독자에서 구독자가 1명이상 생겼을 때 실행되는 함수를 전달 받는다. 구독하는 컴포넌트가 증가하여도, 이 함수는 다시 실행되지 않고 오로지 0개의 구독자에서 1개의 구독자가 생겼을 때 실행된다. 이 콜백은 함수를 return 하는데, 이 함수는 구독자가 1명이상에서 0명이 되었을 때 발생하는 함수이다.

./App.svelte

<script>

import { countStore } from "./store";

let count;

const unSubscribe = countStore.subscribe((c) => {

console.log(c);

count = c;

});

const increase = () => {

countStore.update((c) => {

return c + 1;

});

};

const reset = () => countStore.set(0);

</script>

<button on:click={reset}></button>

<h1 on:click={increase}>

{count}

</h1>

Subscribe메서드

스토어 객체를 import하여, 이 객체의 subscribe메서드를 호출 하게 된다면, return 값으로 구독을 취소 할 수 있는 함수가 전달된다. Subscribe메서드는 첫번째 인자로, 구독이 시작될 때와 값이 변경 될 때 첫번째 인자로 값을 받는 콜백을 전달 받는다.

Update 메서드

스토어 객체의 또 다른 메서드로 update메서드가 존재한다. 이 update메서드를 호출 시 , 첫번째 인자로 받은 콜백이 실행되어, 이 콜백이 return 되는 값이 새로운 store객체의 값이 된다.

Set 메서드

Set메서드는 update메서드와 달리 콜백이 아니라 상수 값을 전달 인자로 받아 스토어 객체를 update 한다.

자동 구독 예제

자동 구독은 .svlete포맷에서 스토어 객체에 $ 를 붙여, 일반 변수처럼 사용 하는 것을 의미한다.이를 이용하면, 꽤나 코드량을 단축할 수 있다.

./App.svelte

<script>

import { countStore } from "./store";

import Parent from './Parent.svelte'

const reset = () => $countStore = 0;

</script>

<button on:click={reset}></button>

<h1 on:click={()=> $countStore += 1}>

{$countStore}

</h1>

<Parent></Parent>

## 읽기 전용 스토어

읽기 전용 스토어는 일반 스토어와 달리 읽기만 가능하다. 이 스토어 객체로 값을 변경 할 수는 없으며, 오로지 읽기만 가능하다. 이 객체는 서버로부터 받은 상수값을 사용하기 적합해 보인다.

./store.js

import { readable } from 'svelte/store';

const user = {

name : "lee",

age : 28,

mail : "substantiation@naver.com"

}

export const userStore = readable(user);

./App.svelte

<script>

import { userStore } from './store'

</script>

{#each Object.entries($userStore) as [key,value]}

<h1>{key} : {value}</h1>

{/each}

Readable함수 또한 writable 처럼 2번째 인수로 콜백을 전달 받는다. 이 callback은 구독자가 0명에서 1명이 될 때만 호출되며, 이 콜백의 첫번째 인자로 readable 값을 set할 수 있는 함수가 전달 된다.

./store.js

import { readable } from 'svelte/store';

const user = {

name : "lee",

age : 28,

mail : "substantiation@naver.com"

}

export const userStore = readable(user,(set)=>{

console.log("구독자가 1명이상이다. ")

delete user.mail;

set(user)

return ()=>{

console.log("구독자가 0명일 때")

}

});