MLH Fellowship

Svelte

GitHub Repo

Rose Kamal Love

What is Svelte?

Another JS Framework / Library ?

Svelte is a radical new approach to build user interfaces. Whereas traditional frameworks like React and Vue do the bulk of their work in the *browser*, Svelte shifts that work into a *compile step* that happens when you build your app.

Instead of using techniques like virtual DOM diffing, Svelte writes code that surgically updates the DOM when the state of your app changes.

```
{
  "name": "svelte-app",
  "version": "1.0.0",
  "private": true,
  ▷ Debug
  "scripts": {
    "build": "rollup -c",
    "dev": "rollup -c -w",
    "start": "sirv public --no-clear"
  },
  "devDependencies": {
    "@rollup/plugin-commonjs": "^17.0.0",
    "@rollup/plugin-node-resolve": "^11.0.0",
    "rollup": "^2.3.4",
    "rollup-plugin-css-only": "^3.1.0",
    "rollup-plugin-livereload": "^2.0.0",
    "rollup-plugin-svelte": "^7.0.0",
    "rollup-plugin-terser": "^7.0.0",
    "svelte": "^3.0.0"
  },
  "dependencies": {
    "sirv-cli": "^1.0.0"
  }
}
```

Why is Svelte?

Make our life easier (I hope so)?

Write Less Code

Build boilerplate-free components using languages you already know — HTML, CSS and JavaScript

No Virtual DOM

Svelte compiles your code to tiny, framework-less vanilla JS — your app starts fast and stays fast

Truly Reactive

No more complex state management libraries — Svelte brings reactivity to JavaScript itself

Basic *Demo* of the Component Pattern

```
<script>
  let name = "Alex";
  let count = 0;

  const handleClick = () => (count += 1);
  const handleInput = (e) => (name = e.target.value);
</script>

<main>
  <h1>Hello {name}!</h1>
  <input type="text" value={name} on:input={handleInput} />
  <button on:click={handleClick}>Click: {count}</button>
</main>

<style>
  main {
    text-align: center;
    padding: 1em;
    max-width: 240px;
    margin: 0 auto;
  }
</style>
```

Using Components and Passing Props

Parent

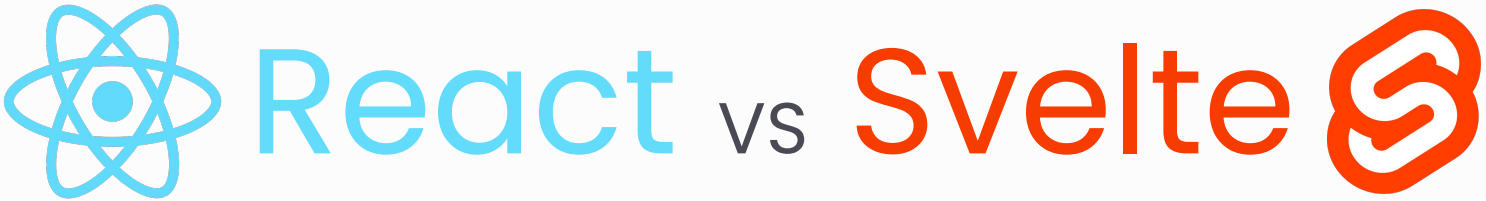
```svelte
<script>
  import Rando from "./Rando.svelte";
  import Todo from "./Todo.svelte";

  let name = "Alex";
  let count = 0;

  const handleClick = () => (count += 1);
  const handleInput = (e) => (name = e.target.value);
</script>

<main>
  <h1>Hello {name}!</h1>
  <input type="text" value={name} on:input={handleInput} />
  <button on:click={handleClick}>Click: {count}</button>

  <hr />
  <Rando {name} rando={count} />
  <hr />
  <h1>Todo in Svelte</h1>
  <Todo />
</main>

<style>…
```

Child

```svelte
<script>
  export let name;
  export let rando;

  const setRando = () => (rando = Math.random());

  /* Yes this is valid JavaScript */
  $: result = Math.round(rando) ? "Winner" : "Loser";
</script>

<main>
  <h1>Hello {name}</h1>
  <p>The random Number is: <code> {rando} </code></p>
  <pre> {Math.round(rando) ?  "Winner" : "Loser"} </pre>
  <pre>Better {result}</pre>
  <button on:click={setRando}>Randomize</button>

  <!— Isn't this way simpler than creating a onChange()⇒{}
  <input type="text" bind:value={rando} />
</main>

<style>…
```

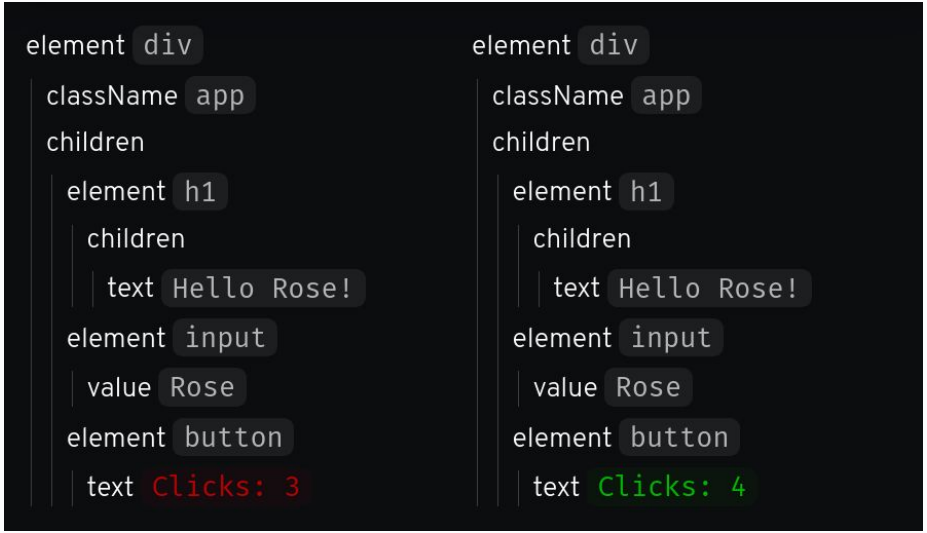
Let's talk about this in more detail

```
<script>
  export let name;
  export let rando;
  const setRando = () => (rando = Math.random());

  /* Yes this is valid JavaScript */
  $: result = Math.round(rando) ? "Winner" : "Loser";
  /* If we wanted to do this in say React,
     We would have to create a state and use
        this.setState({result: Math.round(rando)...})
     or
        setState({...state, result: Math.round()...})
     with hooks.
  */
</script>

<main>
  <h1>Hello {name}</h1>
  <p>The random Number is: <code> {rando} </code></p>

  <!-- A bad way to do this -->
  <pre> {Math.round(rando) ?  "Winner" : "Loser"} </pre>
  <pre> {Math.round(rando) ?  "Winner" : "Loser"} </pre>
  <!-- The better way -->
  <pre>Better {result}</pre>
  <button on:click={setRando}>Randomize</button>

  <!-- Isn't this way simpler than creating a onChange()=>{}  -->
  <input type="text" bind:value={rando} />
</main>
```


React
vs
Svelte

How React handles State Change

```jsx
import { useState } from "react";
import "./App.css";

export default function App() {
  /*
   * When any of these values changes, React goes through
   * each and every element and checks if it has been changed
   * and if it has beem then it re renders it.
   *
   * All that work to change a 0 to a 1
   *
   * To solve this problem react has following:
   *  → shouldCompnentUpdate
   *  → React.PureComponent
   *  → useMemo
   *  → useCallback
   *
   *  Basically, `You're` doing the computer's job
   */

  const [count, setCount] = useState(0);
  const [name, setName] = useState("Alex");

  const handleInput = (e) => setName(e.target.value);
  const handleClick = () => setCount(count + 1);

  return (
    <>
      <div class="center">
        <h1>Hello, {name}!</h1>
        <input type="text" value={name} onInput={handleInput} />
        <button onClick={handleClick}>Click: {count}</button>
      </div>
    </>
  );
}
```



How not to tell the computer “something changed”?

React

```
state = { count: 0 };

// later...
const { count } = this.state;
this.setState({
  count: count + 1
});

/* or... */

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

// later...
setCount(count + 1);
```

Old Svelte

```
data: () => ({ count: 0 })

// later...
const { count } = this.get();
this.set({
  count: count + 1
});
```

THERE ALREADY EXIST A WAY WHICH YOU ALL ARE FAMILIAR WITH

Vanilla JavaScript

```
let count = 0;

// later...
count += 1;
```

Anyone here from Vue.js

```
data: () => ({ count: 0 })

// later...
this.count += 1;
```

```
<script>
  let name = 'world';
  let count = 0;

  function handleClick() {
    count += 1;
  }

  function handleInput(event) {
    name = event.target.value;
  }
</script>
```

FRAMEWORKS
ARE NOT TOOLS FOR
ORGANISING YOUR CODE, THEY
ARE TOOLS FOR
ORGANISING
YOUR MIND

40% Less Code

- Rich Harris (Creator Svelte)

Todo.js

reactDemo > src > Todo.js > …

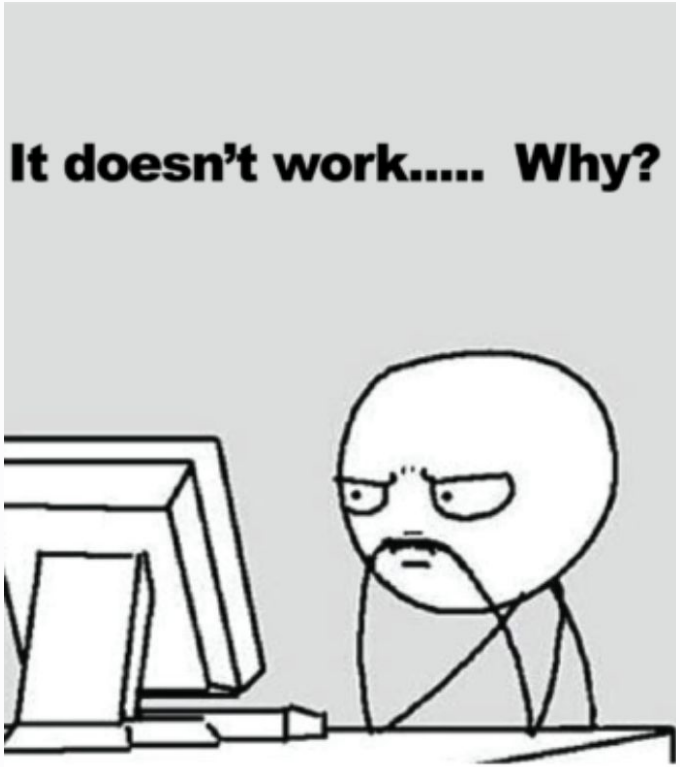
```jsx
import React, { useMemo, useState } from "react";

export default function TodoList() {
  const [todos, setTodos] = useState([
    { done: false, text: "eat" },
    { done: false, text: "sleep" },
    { done: false, text: "code" },
    { done: false, text: "repeat" },
  ]);

  function toggleDone(t) {
    setTodos(
      todos.map((todo) => {
        if (todo === t) return { done: !t.done, text: t.text };
        return todo;
      })
    );
  }

  const [hideDone, setHideDone] = useState(false);

  function toggleHideDone() {
    setHideDone(!hideDone);
  }

  const filtered = useMemo(
    () => (hideDone ? todos.filter((todo) => !todo.done) : todos),
    [todos, hideDone]
  );

  return (
    <div>
      <label class="hide-done">
        <input
          type="checkbox"
          checked={hideDone}
          onChange={toggleHideDone}
        ></input>
        Hide Done
      </label>

      <ul>
        {filtered.map((todo) => (
          <li onClick={() => toggleDone(todo)}>
            {todo.done ? "👍" : ""} {todo.text}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

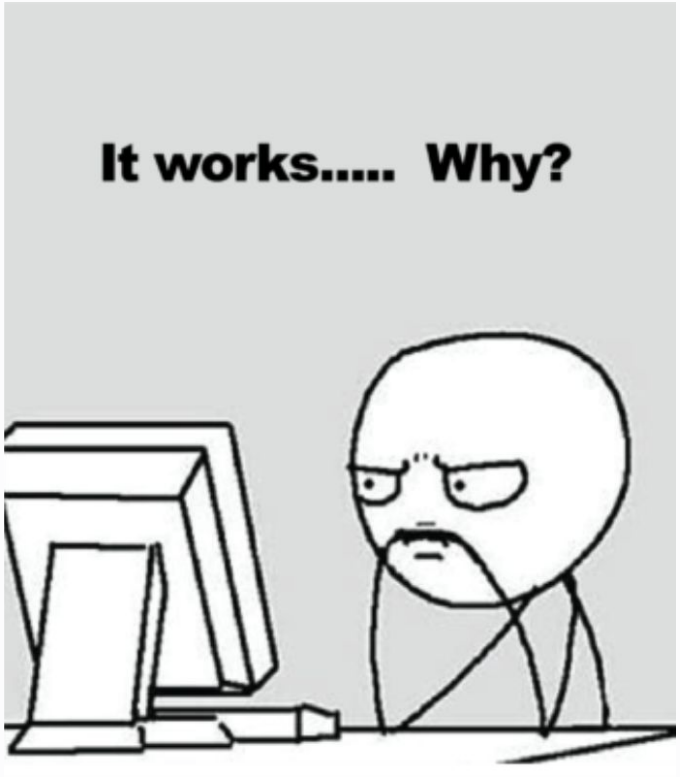
Todo.svelte

svelteDemo > src > Todo.svelte > script

```svelte
<script>
  let todos = [
    { done: false, text: "eat" },
    { done: false, text: "sleep" },
    { done: false, text: "code" },
    { done: false, text: "repeat" },
  ];

  function toggleDone(t) {
    todos = todos.map((todo) => {
      if (todo === t) return { done: !t.done, text: t.text };
      return todo;
    });
  }
  let hideDone = false;

  const filtered = hideDone ? todos.filter((todo) => !todo.done) : todos;
</script>

<label class="hide-done">
  <input type="checkbox" bind:checked={hideDone} />
  Hide Done
</label>

<ul>
  {#each filtered as todo}
    <li on:click={() => toggleDone(todo)}>
      {todo.done ? "👍" : ""}
      {todo.text}
    </li>
  {/each}
</ul>

<style>…
```

Todo Demo


It doesn’t work..... Why?


It works..... Why?

We can relate to this as if it were the code version of Cats in the Cradle. First there is **a**, and **b** who needs **a**. But by the third line, **b** has grown up, and has no time for **a**. It is only when **a** calls that the two are momentarily re-connected, but even that is only fleeting.

Thanks to these cave paintings, we can draw the conclusion that the primitive compilers weren't smart enough to figure out that the destinies of these two proud brothers were intertwined. It seems that ancient programmers had to continually re-establish the relationship, or risk data being out of sync. As modern day code anthropologists, it's hard to imagine how it might have felt to write code like this.

In the year 2051, reactive programming is the norm. Language creators discovered the **destiny operator** decades ago, and the old ways were quickly forgotten. For example, in P#, we can write:

```
var a = 10;
var b <= a + 1;
a = 20;
Assert.AreEqual(21, b);
```

As you can see, the statement establishes **b** and **a** as having intertwined destinies, which are unbroken and forever. They are **bound**. The relationship between them isn't implicit, an idea that only exists in the mind of the programmers; it's explicit, a part of the language, and it exists for all time.

Although the **destiny operator** is wide spread, the way it works is a closely guarded secret. Some say that when the compiler encounters code that changes **a**, it inserts the corresponding change for **b**, such that they are always in sync. Others say that **a**, instead of being a lowly 4-byte integer, is ascended into a higher plane of existence. It becomes an **observable**, an object whose changes reverberate throughout the software at runtime, with the aid of event handlers created by the compiler. Old wives tales even tell of a great timer that constantly ticks, re-aligning all the variables after every change.

We can relate to this as if it were the code version of Cats in the Cradle. First there is **a**, and **b** who needs **a**. But by the third line, **b** has grown up, and has no time for **a**. It is only when **a** calls that the two are momentarily re-connected, but even that is only fleeting.

Thanks to these cave paintings, we can draw the conclusion that the primitive compilers weren't smart enough to figure out that the destinies of these two proud brothers were intertwined. It seems that ancient programmers had to continually re-establish the relationship, or risk data being out of sync. As modern day code anthropologists, it's hard to imagine how it might have felt to write code like this.

In the year 2051, reactive programming is th **rator** decades ago, and the old ways were quickly forgotten. For example, in P#, we ca

```
let a = 10;
$: b = a + 1;
```

```
var a = 10;
var b <= a + 1;
a = 20;
Assert.AreEqual(21, b);
```

As you can see, the statement establishes **b** and **a** as having intertwined destinies, which are unbroken and forever. They are **bound**. The relationship between them isn't implicit, an idea that only exists in the mind of the programmers; it's explicit, a part of the language, and it exists for all time.

Although the **destiny operator** is wide spread, the way it works is a closely guarded secret. Some say that when the compiler encounters code that changes **a**, it inserts the corresponding change for **b**, such that they are always in sync. Others say that **a**, instead of being a lowly 4-byte integer, is ascended into a higher plane of existence. It becomes an **observable**, an object whose changes reverberate throughout the software at runtime, with the aid of event handlers created by the compiler. Old wives tales even tell of a great timer that constantly ticks, re-aligning all the variables after every change.

Hands on **Performance** Demonstration

Concurrent React

Plain Svelte

Wasn't that mind blowing?

Things I **Like About** Svelte

CSS Scoping

In Svelte, you can write CSS in a stylesheet like you normally would on a typical project. You can also use CSS-in-JS solutions, like styled-components and Emotion, if you'd like. It's become increasingly common to divide code into components, rather than by file type.

React, for example, allows for the collocation of a components markup and JavaScript. In Svelte, this is taken one logical step further: the Javascript, markup and styling for a component can all exist together in a single `.svelte` file. If you've ever used single file components in Vue, then Svelte will look familiar.

Styles are scoped by default

By default, styles defined within a Svelte file are *scoped*. Like CSS-in-JS libraries or CSS Modules, Svelte generates unique class names when it compiles to make sure the styles for one element never conflict with styles from another.

This happened recently to me when working on our project.

It lets us create global styles

As we've just seen, you can use a regular stylesheet to define global styles. Should you need to define any global styles from within a Svelte component, you can do that too by using :global. This is essentially a way to opt out of scoping when and where you need to.

Also tells us about unused styles and removes it

:bind vs onChange()

```svelte
<script>
  let crap;
</script>

<main>
  <input type="password" bind:value={crap} />
</main>
```

```jsx
import React, { useState } from "react";

const App = () => {
  const [state, setState] = useState();
  const handleChange = (e) => {
    setState({
      state = e.target.value,
    })
  };
  return (
    <div>
      <input type="text" name="state" onChange={handleChange} />
      <pre>{state}</pre>
    </div>
  );
};

export default App;
```

DOM Access and JavaScript Debugging

HTML Templating vs JSX

```
<ul>
  {#each filtered as todo}
    <li on:click={() => toggleDone(todo)}>
      {todo.done ? "👍" : ""}
      {todo.text}
    </li>
  {/each}
</ul>
```

```
    <ul>
      {filtered.map((todo) => (
        <li onClick={() => toggleDone(todo)}>
          {todo.done ? "👍" : ""} {todo.text}
        </li>
      ))}
    </ul>
  </div>
```

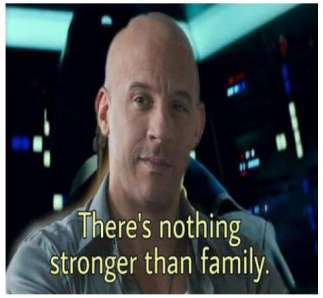
Reactivity
+
Built in State Management

Thanks You So Much

Lépod

is

Family



BYE!!