(v3)

A magically disappearing framework

A compiler, not a runtime

```
<!-- App.html -->
<h1>Hello {name}!</h1>
```

Turns into Vanilla JS in the browser

The old days

```
document.querySelector('input').addEventListener('input', function(event) {
  span.textContent = `Hello ${this.value}`;
});
```

Directly attaching to DOM events. Gets messy fast.

Now

```
<span>Hello, {{ name }}</span>
<input @input="name = $event.target.value">
```

We write state-driven code, and it's the framework's job to tell the browser what to do.

DX vs UX

1. On event
2. Do this thing

Better for users

1. on state change
2. re-render app
3. reconcile new VDOM with the current view, to figure out what we need to do
4. do the things

Better for developers

Frameworks help us organize mentally

- Lets us think about our code as a series of interconnected states
- Abstracts away fiddly DOM management bits
- But that creates extra work (VDOM diffing, change detection) we didn't have to do originally

Most frameworks have runtimes

- Runtimes that live in your browser
- When you ship your app you ship runtime too
- Examples: Angular, React, Vue
- Interpret your app code when browser loads

What if you could have it both ways?

- State management
- Direct DOM manipulation

Enter compilers!

Compilers live in your dev env

- Turn your apps and components into plain JS
- Works as part of your build flow (webpack, rollup, parcel)
- Bundle overhead is just what's needed to modify DOM

Why compile?

Runtimes are big

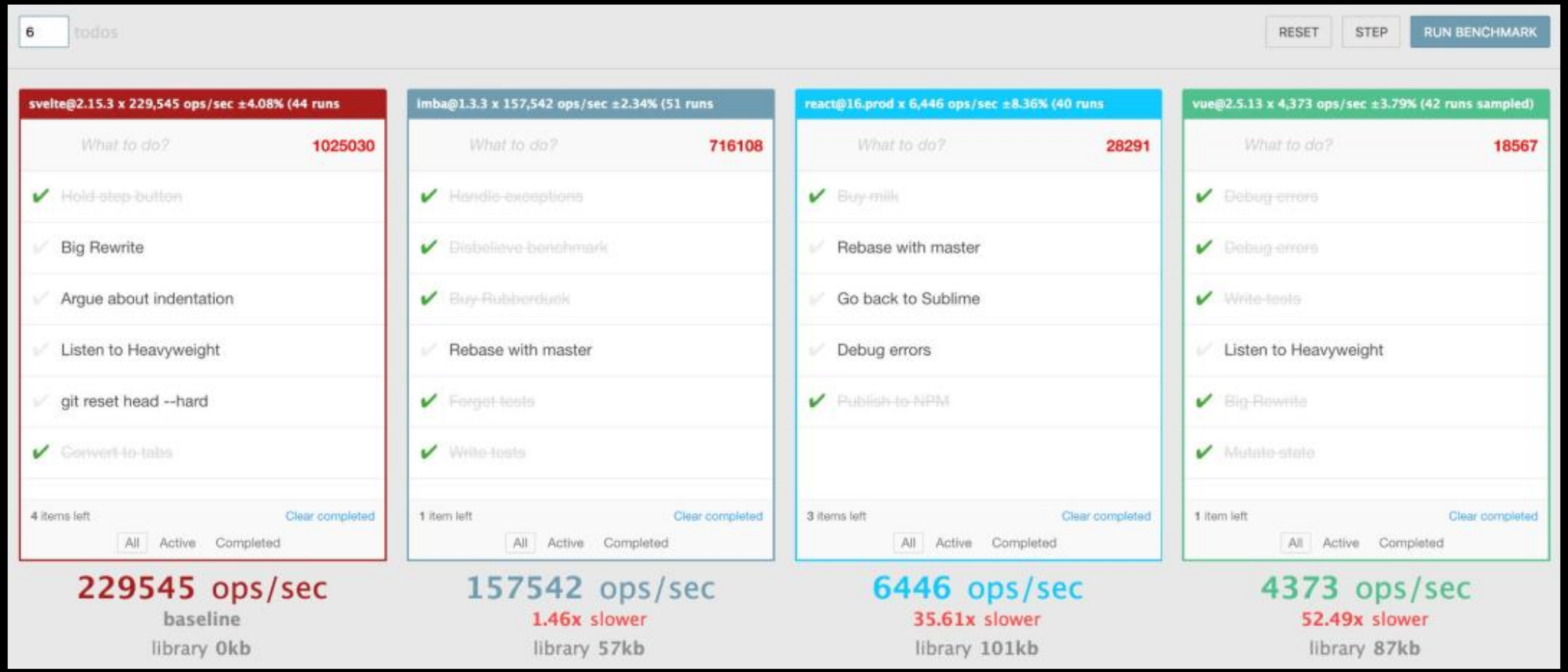
Svelte's Todo MVC weighs 3.6kb zipped. For comparison, React plus ReactDOM without any app code weighs about 45kb zipped

Though they are getting smaller with Angular Ivy, React Prepack, etc.

Runtimes provide unnecessary code

- Difficult to tree-shake all of React
- Better to remove dead code or just don’t generate it?

Runtimes are slow

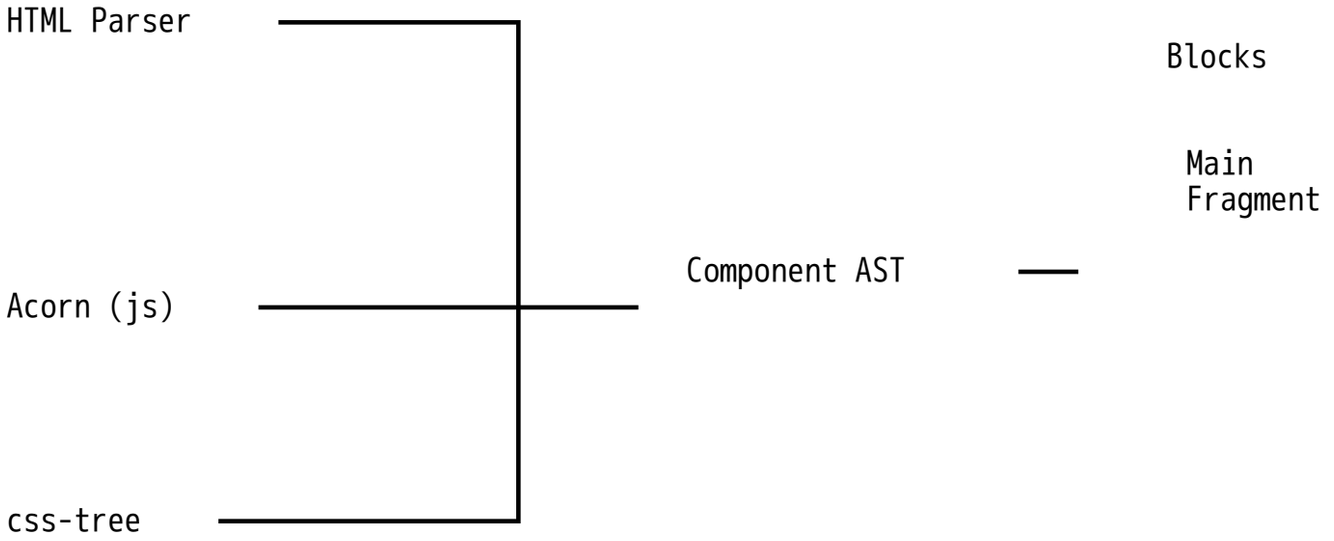


Runtimes lock you in

No runtime means code can be re-used in any framework, or no framework.

Re-using Vue in React or vice-versa requires jumping through hoops. Don’t even think about Angular (but maybe it’s easier with Ivy)

How does Svelte work?



Each block has methods for creating, updating, and destroying the DOM. No need for Virtual DOM.

Svelte works with the DOM directly.

How do you write Svelte?

Components are done in a single file, like Vue or React.

```
<!-- Greeting.html -->
<h1>Hi, {name}, I'm your component!</h1>

<script>
  export let name = 'Bob';
</script>

<style>
  h1 {
    color: blue;
  }
</style>
```

How do you use Svelte components?

```
const greet = new Greeting({
  target: document.body,
  props: {
    name: 'Fred',
  },
});

// Plain prop accessors
greet.name = 'Bobby';

// Set multiple props
greet.$set({ name: 'Jimmy' });
```

How you do you build with components?

```
<!-- App.html -->
<div>
  <Greeting name={name}></Greeting>
</div>

<script>
  import Greeting from './Greeting.html';

  let name = 'Randall';
</script>
```

HTML Syntax

(Spoiler: It's HTMLX)

Interpolated Tags

```
<div>{name}</div>
<div>{Math.floor(counter / 2)}</div>
<div style="color: {color}">Foo</div>
<div hidden={isHidden}>Bar</div>
<div {hidden}>Maybe I'm hidden</div>

<!-- Transitions -->
<p in:fly="{y: 50}" out:fade>flies up, fades out</p>

<script>
  import { fade, fly } from 'svelte-transitions';
</script>
```

Logic

```
<div>
  {#if counter > 0}
    More than zero
  {:elseif counter < 0}
    Less than zero
  {:else}
    Zero
  {/if}
</div>
```

Loops

```
<ul>
  <!-- Destructure loop variables -->
  {#each developers as {name, location}, i}
    <li>{i} - {name}, {location}</li>
  {:else}
    <li>No developers!</li>
  {/each}
</ul>
```

Async

```
<script>
  let promise = fetch('https://reqres.in/api/users')
    .then(res => res.json())
    .then(res => res.data);
</script>

<div>
  {#await promise}
    Fetching...
  {:then users}
    {#each users as {name}} {name} {/each}
  {:catch error}
    <p>Couldn't fetch users: {error}</p>
  {/await}
</div>
```

Directives

x : y - Event handlers, bindings, transitions

```
<button on:click="{() => counter += 1}">+1</button>

<!-- Two-way binding -->
<input bind:value={name}>

<!-- Attribute binding -->
<span class:active={isActive}>Am I active?</span>

<!-- Bring element into js -->
<canvas bind:this={canvas}></canvas>
```

Special elements

- `<svelte:self>` recursion
- `<svelte:component>` dynamic components
- `<svelte:window>` window event listeners
- `<svelte:body>` similar
- `<svelte:head>` title element, etc

Dynamic content with <slot>

```
<!-- Box.html -->
<div>
  <slot></slot>

  <small><slot name="comment"></slot></small>
</div>
```

```
<Box>
  Box content here

  <span slot="comment">Comment here</span>
</Box>

<script>
  import Box from './Box.html';
</script>
```

Styles

Scoped CSS

Styles are scoped by default

```
<style>
  .red {
    color: red
  }
</style>
```

becomes

```
.red.svelte-1xi0wsh {
  color:red
}
```

:global()

Modifier makes classes global

```
<style>
  .user :global(p) {
    font-size: 2rem;
  }
</style>
```

Unused styles

Removed automatically! ✨

State managemenet

Internal state

```
<script>
  // count is available internal
    let count = 0;
</script>

<div>{count}</div>
```

External props

```
<!-- Counter.html -->
<div>{count}</div>

<script>
  export let count = 0;
</script>
```

```
<!-- App.html -->
<Counter {count}></Counter>

<script>
  import Counter from './Counter.html';

  let count = 20;
</script>
```

Lifecycle hooks

```
<script>
  import { onMount, beforeUpdate, afterUpdate, onDestroy } from 'svelte';

  let data;

  onMount(async () => {
    data = await (await fetch('/api/data')).json();
  });
</script>
```

Event modifiers

```
<div on:click|stopPropagation|preventDefault="{() => foo = !foo}">...</div>
```

Component events

- Callbacks
- Event dispatching

Binding callbacks

Callbacks are simply exposed function props

```
<div>
  <!-- `changed` is an exposed prop -->
    <Select changed="{val => changedVal = val}"></Select>
    <br/>
    Changed to: {changedVal || ''}
</div>

<script>
    import Select from './Select.html';

    let changedVal;
</script>
```

Triggering callbacks

```
<!-- call `changed` prop -->
<select on:change="{e => changed(e.target.value)}">
    <option>1</option>
    <option>2</option>
</select>

<script>
  // Expose changed prop
  export let changed;
</script>
```

Event dispatching

```
<!-- Component.html -->
<script>
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();

  setInterval(() => {
    dispatch('tick', { date: new Date() });
  }, 1000);
</script>
```

```
<div>
  {newTickdate}
  <!-- e is a CustomEvent instance -->
  <Component on:tick="{e => newTickdate = e.detail.date}"></Component>
</div>

<script>
  let newTickDate;
</script>
```

Reactivity

- Stores
- Reactive declarations

Stores

```
// stores.js
import { writable } from 'svelte/store';

export const user = writable({ });
```

```
<!-- $ prefix for stores -->
<div><b>User:</b> {$user.name}</div>

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

  onMount(async () => {
    const data = await (await fetch('/api/users')).json();
    user.$set(data);
  });
</script>
```

Reactive declarations

$: the destiny operator

Replaces `computed` values. Tells compiler to run statement any time value on eight-side changes.

```
<p>Counter: {counter}</p>
<p>Double: {double}</p>
<button on:click="{counter += 1 }">+1</button>

<script>
  let counter = 1;
  $: double = counter * 2;
</script>
```

Other things

- Transitions
- Sapper

Example!

REPL

Demo time!

- https://api.hackerwebapp.com/news
- https://v3.svelte.technology/repl