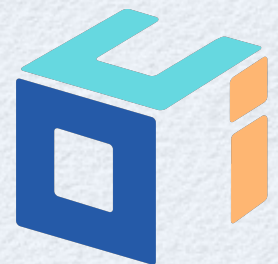




Svelte

slides at <https://github.com/mvolkmann/talks>

R. Mark Volkmann
Object Computing, Inc.
<http://objectcomputing.com>
Email: mark@objectcomputing.com
Twitter: @mark_volkman
GitHub: mvolkmann



OCI | TRAINING

What Is It?

- Alternative to web frameworks like React, Vue, and Angular
- A web application **compiler**, not a runtime library
 - implemented in TypeScript
 - compiles `.svelte` files to a single JavaScript file
 - no Svelte runtime **dependencies**, only **devDependencies**
- Doesn't use a virtual DOM
- Developed by **Rich Harris**
 - formerly at "The Guardian"; currently at "The New York Times"
 - previously created **Ractive** web framework - <https://ractive.js.org/>
 - used at "The Guardian"
 - inspired parts of Vue
 - created **Rollup** module bundler - <https://rollupjs.org/>
 - alternative to Webpack and Parcel

An Example

- Since you are all experienced web developers, let's look at an example app so you can compare the look of the code to your current favorite web framework
- On to the classic ... todo app!

Todo App ...

code and tests at
<https://github.com/mvolkmann/svelte-todo>

To Do List

1 of 2 remaining Archive Completed

Add

☒ learn Svelte Delete

☐ build a Svelte app Delete

← a Todo component

```
src/main.js
import ToDoList from './ToDoList.svelte';
const app = new ToDoList({target: document.body});
export default app;
```


... Todo App ...

script and
style sections
are optional

```
<script>
  import {createEventDispatcher} from 'svelte';
  import {fade} from 'svelte/transition';
  const dispatch = createEventDispatcher();
  export let todo; // the only prop
</script>

<style>
  .done-true {
    color: gray;
    text-decoration: line-through;
  }
  li {
    margin-top: 5px;
  }
</style>

<li transition:fade>
  <input
    type="checkbox"
    checked={todo.done}
    on:change={ () => dispatch('toggleDone') }
  />
  <span class={ 'done-' + todo.done }>{todo.text}</span>
  <button on:click={ () => dispatch('delete') }>Delete</button>
</li>
```

src/Todo.svelte

export makes it a prop

interpolation

What is the name of this component?
Can't tell.
Names are assigned when other
components import this one.

... Todo App ...

```
<script>
  import Todo from './Todo.svelte';

  let lastId = 0;
  const createTodo = (text, done = false) => ({id: ++lastId, text, done});

  let todoText = '';
  let todos = [
    createTodo('learn Svelte', true),
    createTodo('build a Svelte app')
  ];

  $: uncompletedCount = todos.filter(t => !t.done).length;
  $: status = `${uncompletedCount} of ${todos.length} remaining`;

  function addTodo() {
    todos = todos.concat(createTodo(todoText));
    todoText = '';
  }

  const archiveCompleted = () => todos = todos.filter(t => !t.done);

  const deleteTodo = todoId => todos = todos.filter(t => t.id !== todoId);

  function toggleDone(todo) {
    const {id} = todo;
    todos = todos.map(t => t.id === id ? {...t, done: !t.done} : t);
  }
</script>
```

src/TodoList.svelte

Top-level variables ARE the component state if used in HTML!
When state changes, only the relevant part of DOM are updated.

reactive
declarations

No methods,
just functions.

Not really archiving in this
simple implementation,
just deleting.

No this anywhere,
just plain functions!

... Todo App ...

```
<style>                                     src/ToDoList.svelte
  button {
    margin-left: 10px;
  }

  /* This removes bullets from a bulleted list. */
  ul.unstyled {
    list-style: none;
    margin-left: 0;
    padding-left: 0;
  }
</style>
```

... Todo App

```
<div>
  <h2>To Do List</h2>
  <div>
    {status}
    <button on:click={archiveCompleted}>Archive Completed</button>
  </div>
  <form on:submit|preventDefault>
    <input
      type="text"
      size="30"
      autofocus
      placeholder="enter new todo here"
      bind:value={todoText}
    />
    <button disabled={!todoText} on:click={addTodo}>
      Add
    </button>
  </form>
  <ul class="unstyled">
    {#each todos as todo}
      <Todo
        todo={todo}
        on:delete={ () => deleteTodo(todo.id) }
        on:toggleDone={ () => toggleDone(todo) }
      />
    {/each}
  </ul>
</div>
```

src/TodoList.svelte

not doing anything on submit

binds value of form element to a variable;
gives two-way data binding;
provides current value and
event handling for updating variable
when user changes value

Mustache-style markup

passing a prop; can be any JavaScript expression

Logic in Markup

- Three approaches for conditional and iteration logic

- **React**

- uses JSX where logic is implemented with JavaScript code in curly braces

- **Angular and Vue**

- support framework-specific attributes for logic
 - ex. `ngIf`, `ngFor`, `v-if`, `v-for`, ...

- **Svelte**

- supports mustache-like custom syntax that wraps elements
 - ex. `{#if}` and `{#each}`
 - can wrap multiple elements without introducing a new, common parent

Why does it make sense to specify conditional and iteration logic INSIDE elements using attributes?

Imagine if you could do that with JavaScript functions.

```
doSomething(  
  arg1,  
  arg2,  
  if (arg1 > 10),  
  for (arg1 in someCollection));
```

Isn't that weird?

Markup `if` Statement

- Begin with `{#if condition}`
 - starting with `#` indicates a block opening tag
- Can use `{:else if condition}` and `{:else}`
 - starting with `:` indicates a block continuation tag
- End with `{/if}`
 - starting with `/` indicates a block ending tag
- Include markup to be conditionally rendered
- Example

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


Markup each Statement

- Begin with `{#each iterable as element}`
 - include markup to be rendered for each element
- Optional `{:else}`
 - renders when iterable is empty
- End with `{/each}`

- Examples

```
{#each colors as color}  
  <div style='color: {color}'>{color}</div>  
{/each}
```

```
red  
green  
blue
```

```
{#each colors as color, index}  
  <div>{index + 1}) {color}</div>  
{/each}
```

```
1) red  
2) green  
3) blue
```

```
{#each people as {name, age}}  
  <div>{name} is {age} years old.</div>  
{:else}  
  <div>There are no people.</div>  
{/each}
```

can use destructuring
when elements are objects

Promises in Markup

- Can wait for promises to resolve or reject in markup and render different markup for each and while pending a bit like React Suspense
- Examples
 - assuming `getData` function returns a `Promise`

```
{#await getData()}  
  <div>Waiting for data ...</div>  
{:then result}  
  <div>result = {result}</div>  
{:catch error}  
  <div class="error">Error: {error.message}</div>  
{/await}
```

can use any name for variables
after `:then` and `:catch`

```
{#await getData() then result}  
  <div>result = {result}</div>  
{:catch error}  
  <div class="error">Error: {error.message}</div>  
{/await}
```

can omit block to render while
waiting for resolve or reject

can omit `catch` block

Top Svelte Features

- It's fast!
 - see <https://krausest.github.io/js-framework-benchmark/current.html>
 - can select frameworks to compare
- Small bundle sizes
- File-based component definitions
- CSS scoped by default
- Clear place to put global CSS
- Easy component state management (reactivity)
- Reactive statements (`$:`)
- Easy app state management (stores)
- Easy way to pass data from components to descendant components (context)
- Two-way data bindings
- Easy animations built-in
- Runtime warnings for accessibility issues
 - ex. missing an `alt` attribute on an `img` element

Small Bundle Sizes

- Delivered code is much smaller, so loads faster in browsers
- Uses Rollup by default for module bundling, but can also use Webpack or Parcel
- Create production build with `npm run build`
- A RealWorld Comparison of Front-End Frameworks with Benchmarks
 - <https://www.freecodecamp.org/news/a-realworld-comparison-of-front-end-frameworks-with-benchmarks-2019-update-4be0d3c78075/>

Gzipped App Size in KBs

Angular+ngrx: 134
React+Redux: 193
Vue: 41.8
Svelte: 9.7

Lines of Code

Angular+ngrx: 4210
React+Redux: 2050
Vue: 2076
Svelte: 1116

Does It Disappear?

- Some say Svelte disappears once an app is built
- The Svelte library is mostly defined by `.js` files in `node_modules/svelte`
 - main functions are defined in `internal.js` which is currently ~1400 line of code
 - other library files used for specific features
 - `easing.js`, `motion.js`, `register.js`, `store.js`, `transition.js`
- `npm run build` produces files in `public` directory
 - including `bundle.js`
- Svelte library functions that are used by the app are copied to the top of `bundle.js`
 - in the case of the Todo app shown earlier, this is ~500 lines of code
- So Svelte doesn't disappear, it is just very small

File-based Component Defs

- **Angular** uses classes
- **React** uses functions or classes
- **Vue** uses object literals
- **Svelte** doesn't use any JavaScript container
 - JavaScript, CSS, and HTML in source files are combined to form the component definition which automatically becomes the default export
 - name is associated when imported and must start uppercase
 - can't tell from looking at source file what names might be used
 - lowercase names are reserved
 - for predefined elements like those in HTML and SVG

CSS

- Scoped by default
 - CSS specified in a component `style` tag is automatically scoped to the component
 - achieved by adding the same generated CSS class name, `svelte-hash`, to each rendered element of the component affected by these CSS rules
 - CSS rules for component only apply to elements with this class name
- Clear place for global CSS
 - `public/global.css`
- svelte3 ESLint plugin warns about unused CSS selectors
 - see <https://github.com/sveltejs/eslint-plugin-svelte3>

Easy Component State Mgmt.

("reactivity")

- Changes to top-level variables referenced in interpolations automatically cause those interpolations to be reevaluated
- Example

```
<script>
  let count = 0;
  const increment = () => count++;
</script>

<div>count = {count}</div>
<button on:click={increment}>+</button>
```

- Must assign a new value to trigger
 - pushing new elements onto an array doesn't do this

```
myArr = myArr.concat(newValue);
```

works

```
myArr = [...myArr, newValue];
```

works

```
// Alternative trick
myArr.push(newValue);
myArr = myArr;
```

works

Reactive Statements

a.k.a. "destiny operator"

- `$:` is a "labeled statement" with label name "`$`" that Svelte treats as a "reactive statement"

Labeled statements can be used as targets of `break` and `continue` statements. It is not an error in JavaScript to use same label more than once in same scope.

- Add as a prefix on top-level statements that should be repeated whenever any referenced variables change
- Examples

```
$: average = total / count;  
$: console.log('count =', count);
```

like "computed properties" in Vue

great for debugging

When applied to an assignment to an undeclared variable it is called a "reactive declaration" and the `let` keyword is not allowed.

- Can apply to a block


```
$: {  
  // statements to repeat go here  
}
```

- Can apply to multiline statements like `if` statements

```
$: if (someCondition) {  
  // body statements  
}
```

re-evaluates condition if any variables it references change, and executes body only when true

Easy App State Mgmt.

- “Stores” hold application state outside any component
- Alternative to using props or context to make data available in components 
- Where to define?
 - 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
 - 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

Kinds of Stores

- **Writable**

- only kind that can be modified by components
- methods
 - `set(newValue)`
 - `update(currentValue => newValue)`

calculates new value from current value

- **Readable**

- handle computing their data
- components cannot modify

- **Derived**

- derive data from current values of other stores

- **Custom**

- must implement `subscribe` method
- can provide custom methods to update state and not expose `set` and `update` methods

Defining Writable Stores

stores.js

```
import {writable} from 'svelte/store';  
export const dogStore = writable([]);
```

initial value

```
export const fancyStore = writable(  
  initialValue,  
  async set => {  
    // Called when subscribe count goes from 0 to 1.  
    // Compute initial value and pass to set function.  
    const res = await fetch('/some/url');  
    const data = await res.json();  
    set(data);  
  
    return () => {  
      // Called when subscriber count goes to 0.  
    }  
  }  
);
```

using optional
second argument

error handling omitted

Using Stores

- Option #1 - **subscribe** method - very verbose!
- Option #2 - **\$** auto-subscription shorthand - much better!
 - variables whose names begin with **\$** must be stores
 - automatically subscribes when first used and unsubscribes when removed from DOM

```
<script>
  import {onDestroy} from 'svelte';
  import {dogStore} from './stores.js';
  let dogs;
  const unsubscribe = dogStore.subscribe(value => dogs = value);
  onDestroy(unsubscribe);
</script>
```

uses **subscribe** method

```
<!-- Use dogs in HTML. -->
```

```
<script>
  import {dogStore} from './stores.js';
</script>
```

uses auto-subscription

```
<!-- Use $dogStore in HTML. -->
```

Easy Passing Data to Descendants

- Use “context”
- Alternative to props and stores for making data available in descendant components

```
import {getContext, setContext} from 'svelte';
```

- Ancestor components set context associated with the component

```
setContext(key, value);
```

- must be called during component initialization
- Descendant components get context from closest ancestor that has context with given key

```
const value = getContext(key);
```

- must be called during component initialization

More on Context

- Keys can be any kind of value, not just strings
- Values can be any kind of value including functions and objects with methods
- Context is not reactive!
 - descendant components are not re-rendered when context they use is changed in an ancestor component
 - use stores if reactivity is needed

Context Example

A.svelte

```
<script>
  import {setContext} from 'svelte';
  import B from './B.svelte';
  setContext('favorites', {color: 'yellow', number: 19});
</script>

<div>
  This is in A.
  <B />
</div>
```

Output

```
This is in A.
This is in B.
This is in C.
favorite color is yellow
favorite number is 19
```

B.svelte

```
<script>
  import C from './C.svelte';
</script>

<div>
  This is in B.
  <C />
</div>
```

C.svelte

```
<script>
  import {getContext} from 'svelte';
  const {color, number} = getContext('favorites');
</script>

<div>
  This is in C.
  <div>favorite color is {color}</div>
  <div>favorite number is {number}</div>
</div>
```

context data is **not reactive**;
use stores when that is needed

Two-way Data Bindings

- Form elements can be bound to a variable
- Simulates two-way data binding
- Provides current value and event handling for updating variable when user changes value

Binding Example ...

Name

Happy? ☒

Favorite Flavors ☒ vanilla ☐ chocolate ☒ strawberry

Favorite Season ☒ Spring ☐ Summer ☐ Fall ☐ Winter

Favorite Color

Life Story

Mark likes yellow, Spring, and is happy.
Mark's favorite flavors are vanilla, strawberry.
Story: Once upon a time ...

```
<script>
  const colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple'];
  const flavors = ['vanilla', 'chocolate', 'strawberry'];
  const seasons = ['Spring', 'Summer', 'Fall', 'Winter']
  let favoriteColor = '';
  let favoriteFlavors = [];
  let favoriteSeason = '';
  let happy = true;
  let name = '';
  let story = '';
</script>
```

these variables
are bound to
form elements on
next two slides

... Binding Example ...

```
<div class="form">
  <div>
    <label>Name</label>
    <input type="text" bind:value={name} />
  </div>
  <div>
    <label>Happy?</label>
    <input type="checkbox" bind:checked={happy} />
  </div>
  <div>
    <label>Favorite Flavors</label>
    {#each flavors as flavor}
      <label>
        <input type="checkbox" value={flavor} bind:group={favoriteFlavors} />
        {flavor}
      </label>
    {/each}
  </div>
```

for checkboxes, bind to
checked property
rather than **value**

using **bind:group** with a set
of related checkboxes makes
the value an array of strings

... Binding Example

```
<div>
  <label>Favorite Season</label>
  {#each seasons as season}
    <label>
      <input type="radio" value={season} bind:group={favoriteSeason} />
      {season}
    </label>
  {/each}
</div>
<div>
  <label>Favorite Color</label>
  <select bind:value={favoriteColor}>
    <option />
    {#each colors as color}
      <option>{color}</option>
    {/each}
  </select>
</div>
<div>
  <label>Life Story</label>
  <textarea bind:value={story} />
</div>
```

using `bind:group` with a set of related radio buttons makes the value a single string

to change a select to a scrollable list that allows selecting multiple options add `multiple` attribute; makes the value an array of selected values

Easy Animations Built-in

- **svelte/animate** provides
 - **flip**
- **svelte/motion** provides
 - **spring**
 - **tweened**
- **svelte/transition** provides
 - **crossfade**
 - **draw** - for SVG elements
 - **fade**
 - **fly** - set **x** and/or **y**
 - **scale**
 - **slide**
- Also see **svelte/easing**

basic example

```
<script>
  import {fade} from 'svelte/transition';
  ...
</script>

<li transition:fade>
  <!-- some content -->
</li>
```

this fades in when mounted
and fades out when destroyed

Outstanding Issues

- TypeScript support
 - it's coming, but not ready yet
 - <https://github.com/sveltejs/svelte/issues/1639>
 - but `svelte.preprocess` can be used now to enable use of TypeScript inside `<script>` tags
 - won't type check code in HTML interpolations
- Popularity
 - perhaps Svelte is now considered the #4 most popular approach for building web apps
 - isn't easy to find developers that already know it
 - but it's very easy to learn and there is less to learn than other approaches

Creating a Project

- Install Node.js from <https://nodejs.org>
 - installs **node**, **npm**, and **npm** commands
- Approach #1
 - **npm** `deg`it `sveltejs/template` *app-name*
 - **deg**it is useful for project scaffolding
- Approach #2
 - browse <https://svelte.dev/repl>
 - click download button
 - unzip downloaded zip file
 - demo this!

degit was created by Rich Harris.
It downloads a git repo,
by default the master branch.
In this case **sveltejs** is the user name
and **template** is the repo.
The second argument is the
name of the directory to create.

Installing and Running

- `cd app-name`
- `npm install`
- `npm run dev`
 - provides live reload, unlike `npm run start`
 - syntax errors are reported in window where this is running, not in browser because it doesn't produce a new version of the app if there are errors
- Browse `localhost:5000`
- Just renders "Hello world!"

Topics Not Covered Here

but covered at <https://objectcomputing.com/resources/publications/sett/july-2019-web-dev-simplified-with-svelte>

- Inserting HTML
- Slots
 - for passing child elements to a component
- Event details
 - handling, modifiers, dispatching
- Lifecycle functions
 - `onMount`, `beforeUpdate`, `afterUpdate`, and `onDestroy`
- Actions
 - register a function to be called when a specific element is added to DOM
 - ex. moving focus
- Routing
 - can use `page` on npm or Sapper
- Module Context
 - to run JavaScript code in a component source file only once instead of once for each component instance created
- Special Elements
 - `<svelte:name ...>`
- Debugging with `{@debug}`
 - debugger breaks on state changes
- Unit tests
 - with Jest and Svelte Testing Library
- End-to-end tests
 - with Cypress
- Compiling to custom elements
 - can be used with any framework

Related Tools

- **"Svelte" VS Code extension**
 - provides syntax highlighting and intellisense
- **Sapper** - <https://sapper.svelte.dev/>
 - "application framework powered by Svelte"
 - similar to Next and Gatsby
 - provides routing, server-side rendering, and code splitting
- **Svelte Native** - <https://svelte-native.technology/>
 - for implementing native mobile apps
 - based on NativeScript
 - community-driven project
- **Svelte Testing Library** - <https://testing-library.com/docs/svelte-testing-library/intro>
- **Storybook** with Svelte - <https://storybook.js.org/docs/guides/guide-svelte/>

Svelte Resources

- **“Rethinking Reactivity” talk by Rich Harris**
 - delivered multiple times, most recently at “Shift Conference” June 20, 2019
 - explains issues with using virtual DOM (like React and Vue) and motivation for Svelte
- **Home page** - <https://svelte.dev>
 - contains **Tutorial**, **API Docs**, **Examples**, online **REPL**, **Blog**, and **Sapper** link
 - REPL is great for trying small amounts of Svelte code
 - REPL can save for sharing and submitting issues
- **Discord chat room** - <https://discordapp.com/invite/yy75DKs>
- **GitHub** - <https://github.com/sveltejs/svelte>
- **Awesome Svelte** - <https://github.com/CalvinWalzel/awesome-svelte>
- **Awesome Svelte Resources** - <https://github.com/ryanatkn/awesome-svelte-resources>

Conclusion

- Svelte is a worthy alternative to the current popular options of React, Vue, and Angular
- For more, see my long article
 - <https://objectcomputing.com/resources/publications/sett/july-2019-web-dev-simplified-with-svelte>