International JavaScript Conference

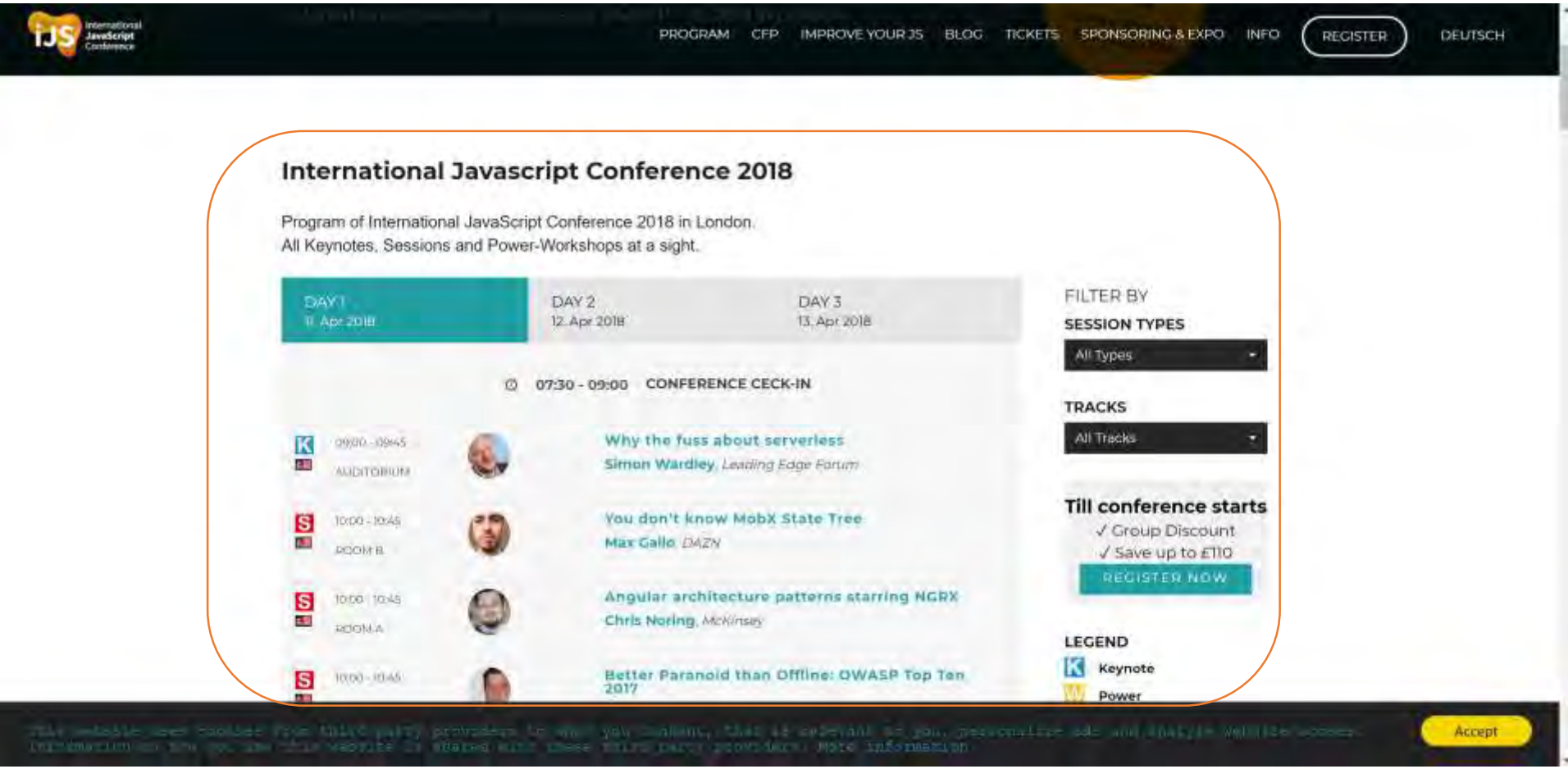
Stencil: The Time for Vanilla Web Components has Arrived

Gil Fink

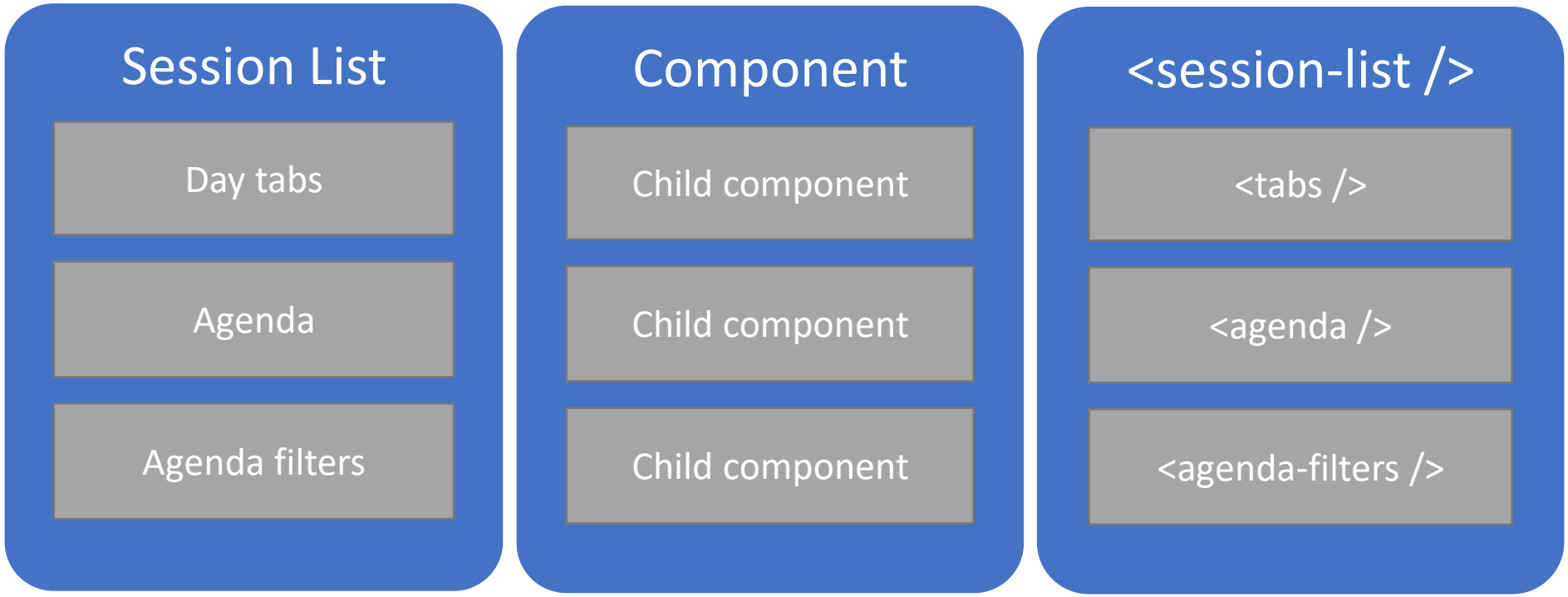
sparXys CEO

@gilfink / www.gilfink.net

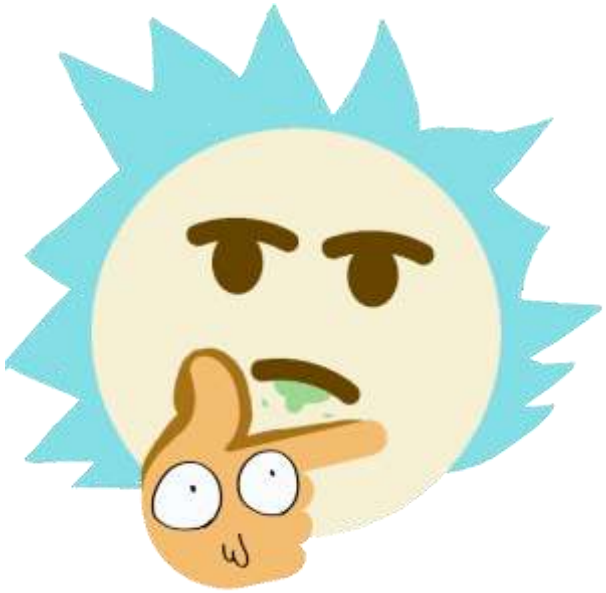
Typical Application Web Page Design



From Design to Implementation



How would you build that page?



Do we really need all these frameworks/libraries?

What if we could teach the browser new elements?

Each Element Instance

- Will be a DOM element
- Creates its own DOM tree
- Can be accessed and manipulated using DOM functions or its own API
- Is a JavaScript object

- Is this possible?

This is where our journey begins

About Me

- sparXys CEO and senior consultant
- Microsoft MVP in the last 9 years
- Pro Single Page Application Development (Apress) co-author
- 4 Microsoft Official Courses (MOCs) co-author
- GDG Rishon and AngularUP co-organizer

sparXys

Agenda

- The Problems We Faced
- Web Components APIs
- Stencil

Undescriptive Markup

Poor Separation of Concerns

You want HTML, CSS and JavaScript to work together

You end up with a mess

The wiring gets in your way!

Bundling is Hard

- You want to bundle a complex component

  The component includes HTML, CSS and JavaScript

  how would you do that?
  - Use a server side mechanism?
  - Bundler? (Webpack/Browserify)

Web Components Standard to The Rescue

- Natively-supported, standardized JavaScript components

- Some general goals:

Code Reuse	Encapsulation	Separation of Concerns	Composition	Theming	Expressive	Semantic

The Web Components Standard

Templates	• Reusable DOM fragments
Imports	• Load HTML declaratively
Shadow DOM	• DOM encapsulation
Custom Elements	• Create your own elements

Custom Elements

- Enable to extend or create custom HTML elements
- Defined using the **customElements.define** function:

```
var myInput = window.customElements.define('my-input',
  class x extends HTMLElement {…});
```

or extend an existing element:

```
var myInput = window.customElements.define('my-input',
  class y extends HTMLInputElement {...});
```

Custom Elements – Usage

- Use the element in your DOM:

```
<my-input></my-input>
```

or use the **createElement** function:

```
var  elm = document.createElement('my-input');
```

Custom Element Life Cycle Events

- connectedCallback
- disconnectedCallback
- attributeChangedCallback

```
class MyInput extends HTMLElement {
        constructor() {
          super();
          // your initialization code goes here
        }
        connectedCallback() {…}
        disconnectedCallback() {…}
        attributeChangedCallback() {…}
}
```

Demo

Custom Elements

A Problem with Web Development Today

- Catholic wedding with frameworks/libraries
- Infrastructure is based on a framework/library
- Infrastructure isn’t reusable if other company projects use another framework/library

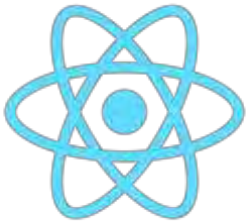
Problem with Web Development Today – Cont.

- Custom Elements can remove the barrier of framework/library coupling
- Can be used by any framework/library
- Encapsulate their functionality and style
- Suitable for component infrastructure development

But there are problems with custom elements

Problems with Custom Elements

- We are used to runtime framework/library goodies such as:
  - Virtual DOM
  - Data binding
  - Performance
  - Server side rendering
  - And etc.




Problems with Custom Elements – Cont.

- Verbose syntax
  - Too much boilerplate
  - We need to craft everything by ourselves

Problems with Custom Elements – Cont.

- Still W3C working draft
- Need Polyfills in some browsers

Is there a better way?

What if I told you that you can solve all the previous problems?


STENCIL

What is Stencil?

- A compiler that generates Custom Elements
- Not a framework/library
  - Output is 100% standards-compliant web components
- Adds powerful framework features to Web Components
  - Virtual DOM
  - Reactivity
  - JSX
  - TypeScript
  - And etc.
- Created and used by Ionic Framework

Stencil Component Example

```
import { Component, Prop } from '@stencil/core';
@Component({
  tag: 'my-name',
  styleUrl: 'my-name.scss'
})
export class MyName {
  @Prop() name: string;

  render() {
    return (
      <p>
        Hello, my name is {this.name}
      </p>
    );
  }
}
```

From Stencil to Custom Elements


```
import { Component, Prop } from
'@stencil/core';

@Component({

  …

})

export class CollapsiblePanel {

  …

}
```

Stencil Compiler

```
var CollapsiblePanel = (function ()
{
    function CollapsiblePanel() {
    }

    … // implementation


    return CollapsiblePanel;
}());
```


DESIGN TIME
YEAH BABY!
imgflip.com

Getting Started with Stencil

```
git clone https://github.com/ionic-team/stencil-component-starter.git my-component
cd my-component
git remote rm origin

npm install
npm start
```

Demo

Hello Stencil

Stencil Generated Component Advantages

- Virtual DOM
  - fast DOM updates without common DOM performance pitfalls
- Lazy Loading
  - By default components load asynchronously and can be bundled with related components
- Reactivity
  - Efficient updates based on property and state changes
- High-performance Rendering
  - async rendering system, similar to React Fiber

Stencil API

- Based on JavaScript decorators
- Written with TypeScript
- You can use the following decorators:
  - @Component()
  - @Prop()
  - @State()
  - @Event()
  - @Listen()
  - @Element()
  - @Method()

@Component Decorator

- The main Stencil decorator
- Configures the entire component including
  - Tag
  - Style
  - Shadow DOM
  - Host
  - Assets

```
import { Component } from '@stencil/core';
@Component({
  tag: 'st-comp',
  styleUrl: 'comp.scss',
  shadow: true
})
export class Comp {
  ...
}
```

@Prop and @State Decorators

- The Prop decorator is used to indicate that a member is exposed as component attribute
- The State decorator is used to indicate that a member is part of the component state
- Reactivity

```
import {Component, Prop, State} from '@stencil/core';
@Component({
  tag: 'collapsible-panel',
  styleUrl: 'collapsible-panel.css'
})
export class CollapsiblePanel {
  @Prop() title: string;
  @State() collapsed: boolean;
  ...
}
```

@Method Decorator

- The Method decorator is used to expose component API

```
import { Component, Element, Method } from '@stencil/core';
@Component({
  ...
})
export class Toaster {
  @Element() toasterDiv: HTMLElement;

  @Method()
  showToast() {
    this.toasterDiv.style.display = 'block';
  };
}
```

Demo

Creating a Stencil Component

Deploying a Stencil Component

- Update the **stencil.config.js** file, if needed
  - **stencil.config.js** in Stencil starter already has these things configured

```
exports.config = {
  namespace: 'myname',
  generateDistribution: true,
  generateWWW: false,
  ...
};
```

Deploying a Stencil Component – Cont.

- Update the **package.json** file, if needed

```
{
  "main": "dist/collection/index.js",
  "types": "dist/collection/index.d.ts",
  "collection": "dist/collection/collection-manifest.json",
  "files": [
    "dist/"
  ],
  "browser": "dist/myname.js",
  ...
}
```

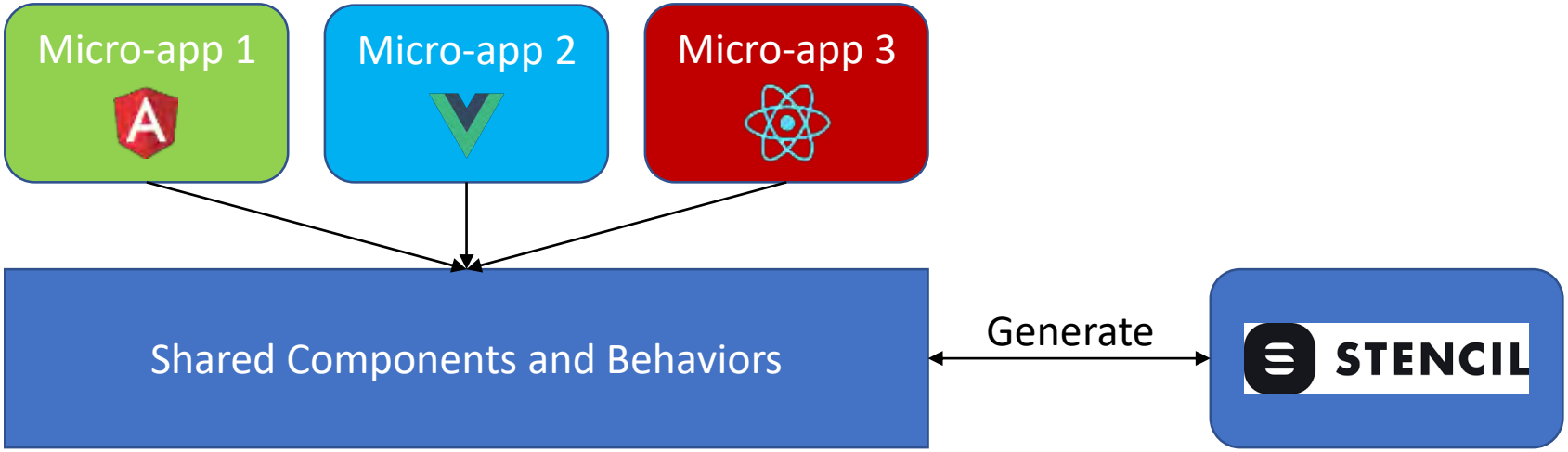
How Stencil Solves the Frameworks Problem?

- Stencil works primarily in build time
- Any framework/library (such as React or Angular) can consume the generated component
  - As a script tag
  - As a node module
  - Using the stencil-starter-app
- Stencil is suitable for infrastructure components

Demo

Consuming a Stencil component from Angular

A Word About Micro Frontends



Summary

- Web Component standard is very powerful
  - But… still in development
- Stencil compiler can ease the pain of creating custom elements
  - Includes a lot of advantages such as JSX, TypeScript and more
  - Generates standard-compliant web components

Resources

- Stencil website: https://stenciljs.com/
- Custom Elements: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Custom_Elements

- My Website – http://www.gilfink.net
- Follow me on Twitter – @gilfink

#UseThePlatform

Thank You!