Evo-WC

Evo-WC is a web component transpiler that takes template file based on HTML and XML and created raw JavaScript Web Component files. These components rely solely on the baseclass `EvoElement` that is included. No other runtime frameworks, libraries or code is needed.

You can create one or more components in the template file. But you can only have components or comments in the template file. Any other top level element will result in a compile error.

There are sample components in the `components` folder. They are XML based files. Take a look at a few and read the docs to see how to create your own.

Parts of a component file

The component file is based on XML. Your component is defined with a `<component>` element. You must include the `tag="component-name"` attribute in the `<component>` element.

The valid elements inside the `<component>` element are:

- `<template>` - The HTML for your component.
- `<style>` - The CSS for this component.
- `<script>` - Custom JavaScript for this component.
- `<import>` - List needed imports at the top of the generated JavaScript file.

The minimal component is made up by using both the `<component>` and `<template>` tags like this:

```
<component tag="my-tag">
  <template>Hi</template>
</component>
```

Component Element

Each component is created by using the `<component></component>` element. Each component must have a `tag` attribute that defines the element tag to be used in your HTML to access this component.

Each component must also include a `<template></template>` that represents the HTML template for your component

Example:

Template file:

```
<component tag="special-thing">
  <template>
    <p>Hello world.</p>
  </template>
</component>
```

Your HTML:

```
<html>
<body>
  <h1>Example file</h1>
  <special-thing></special-thing>
  <script type="module">
    import { SpecialThingElement } from './js/components/SpecialThingElement.js';
  </script>
</body>
</html>
```

Property Definition Attributes (PDA)

Most components will need to provide a way for the parent code to interact with the component. This is done by adding attributes into the `<component>` element that starts with a colon (`:`). These attributes will become properties on the component object as well as attributes that are monitored by the component's `observedAttributes` callback.

Example:

```
<component tag="special-thing" :message>
  <template>
    <p :text="message"></p>
  </template>
</component>
```

The attribute `:message` in the `<component>` element will create a public property called `message` in the component class and the component will observe changes made to the `message` attribute. If you change the `message` attribute the code will also change the component property.

When you create a PDA you can also define the data type of the generated property as well as an optional default value.

The variable type names that can be used are `'array'`, `'bool'`, `'date'`, `'int'`, `'number'`, `'object'`, `'string'` with the default type being `'string'`.

We also support shortened type names of `'arr'` for `'array'`, `'bool'` for `'boolean'`, `'num'` for `'number'`, `'obj'` for `'object'`, and `'str'` for `'string'`.

Example

```
<component tag="special-thing" :age="int:10" :name="str" :enable="bool:false">
  <template>
    <p>Hello <span :text="name"></span>.</p>
    <p>You are <span :text="age"></span> years old.</p>
  </template>
</component>
```

In the example above the property `age` will be a number and will have a default value of `10` when the component is constructed. Any value passed into `component.age` will be converted into a number. So `component.age = '33'` will store a numeric value of `33` and not a string of `"33"`.

You can use URI encoding in your default values. If you want to add a double quote in the default value it needs to be written as a URI encoded value of `%22`. So, instead of writing the PDA like this: `:animals="arr:[1,2,"%22"dogs"]"` which is invalid you would write it like this: `:animals="arr:[%22cats%22,%22dogs%22]"`.

Example

```
<component tag="special-list" :animals="arr:[%22cats%22,%22dogs%22]">
  <template></template>
</component>
```

Here is an example component that has a PDA called `message`. WHenever the `message` attribute on this component is set or when the `message` property is set then the text of the `<h1>` tag will be set to that value.

Example:

Component definition:

```
<component tag="my-message" :message>
  <template>
```

```
    <h1 :text="message"></h1>
  </template>
</component>
```

HTML that uses the component:

```
<html>
<body>
  <my-message message="Initial Message"></my-message>
  <button onclick="changeMessage()">Change message</button>
  <script>
    function changeMessage() {
      const target = document.querySelector('my-message');
      target.message = "Second Message";
    }
  </script>
</body>
</html>
```

Both before and after the user clicks on the button the message attribute for the `<my-message>` component will be `"Initial Message"`. But, the value for `target.message` will be `"Initial Message"` before the user clicks the button and `"Second Message"` after they click on the button.

> All public component properties are defined in the component class using private class fields. These private fields are exposed outside of the class by using public `getter` and `setter` methods.

In the above example there is a private class field called `#message` and both a getter and a setter called `message`. When the setter is called is when the magic of Evo-WC happens. The setters know everything in the UI that needs to be updated. And, when you call the setter it does update only the things that use this variable.

Private properties

You can create private properties that are not accessable outside the component and are not settable by changing an attribute. These propertied are defined with a `#` like this `:#propName` These properties can not be set outside of the generated class. The component code can call the private getter and setter for these properties like this `const a = this.#propName;` or `this.#propName = 'something';`

Updating component attributes

Sometimes you may want an attribute on the component to change when a property changes. This is a common need for CSS. If you want the property to also set the attribute on the component then your PDA needs to to include a `+` like this `:+show`. In this case, when you set the `show` property it will also set the attribute `show` on the component.

dataset / data attributes

We support setting the `data` attributes through the `dataset` property. If you create an attribute like this `:data-dog-food="varName"` then the property `element.dataset.dogFood` will be set every time the setter for `varName` is called.

We recommend using dataset attributes for passing needed data to an event handler.

example using dataset to pass params

```
<component tag="my-el" :#locale>
  <template>
    <p>Current locale is <span :text="#locale"></span></p>
    <button data-locale="en" .click="#handleLocale">EN</button>
    <button data-locale="ja" .click="#handleLocale">JS</button>
    <button data-locale="ru" .click="#handleLocale">RU</button>
  </tamplate>
  <script>
    #handleLocale(event) {
      this.#locale = event.target.dataset.locale;
    }
  </script>
<component>
```

Style element

Place all of your CSS for the component into this `<style>` element.

> Possible future plans to allow importing external CSS files.

Script element

Place all of your JavaScript for the component into this `<script>` element. All of the code you place in here is inserted into the component class as class methods.

> **Important: Do not use fat arrow functions for your class methods.**

Lifecycle Methods

You can provide, in your script, lifecycle functions that will be called at specific times in the lifecycle of the component.

Method	Description
`init()`	Called at the end of the constructor. Follow all of the requirements for custom element constructors and reactions.
`update(member, oldVal, newVal)`	Called every time any property is updated when the component is connected to the DOM. You can adjust any private properties in this function. If this componenet was previously disconnected from the DOM then this also is called when the component is reconnected. The value for `member` is the name of the member variable that was updated just before calling this function.
`connected()`	This is called by the base class `connectedCallback` function.
`disconnected()`	This is called by the base class `disconnectedCallback` function.
`adopted()`	This is called by the base class `adoptedCallback` function.
`attrChanged(attr, oldVal, newval)`	Called when an attribute has changed. This is called by the base class `attributeChangedCallback` function.

Import element

Place any imports that are needed for this class and only for this class in this element.

Bindings

Properties and Attributes

Within the template an attribute that starts with a colon (`:`) indicates the name of the property that will be set by the variable defined in the quotes.

Example:

If we have this image tag in the template: `<img :src="imageUrl" :title="#imageTitle">`

- Every time the component's public property `imageUrl` changes Evo-WC will set the `src` property of the image tag like this: `img.src = this.imageUrl;`

- Every time the component's private property `#imageTitle` changes Evo-WC will set the `title` property of the image tag like this: `img.title = this.#imageTitle;`

> *You can only place public or private propert names in the quotes and NOT call code like Angular.*

Generaly these values will be put into the property getter and setter methods. There are certain properties that do not exist in an HTML element and so we auto-set the attribute instead. For example there is no `alt` property on an `<img>` element so Evo WC will set the attribute `alt` instead. *Evo WC will only convert from properties to attributes for the well defined standard HTML attributes.*

> ***As of 2023-02-27 We have not compleated the attribute conversion list.***

The template element attributes can be set like this:

Attr	Code generated
`:title="val"`	`element.title = this.val`
`:title="#val"`	`element.title = this.#val`
`:attr.state="st"`	`element.setAttribute('state', this.st)` or, if the value of `this.st` is `null`, `element.removeAttribute('state')`
`:html="value`	`element.innerHTML = this.value`
`:text="name"`	`element.textContext = this.name`

2-way-binding - `:value`

`:value` provides two way binding for `<input>` and `<textarea>` elements. Like: `<input :value="firstName" />` will 2-way bind to the component's `get firstName()` and `set firstName()` methods.

Events

An attribute that starts with a period (`.`) are event handlers:

- `.click="functionName"`
- `.mouseup="#myMouseUp"`

> You can use private member functions by starting the function name with `#` both here and for the acctual function name. (JavaScript private class fields)

All event handlers receive one argument. That is the `event`. You can not pass any parameters into the event handler. Instead you add data attributes with anything you need to access in the event

handler. For example you can have several buttons to change the component's locale and each has a different value for the attribute `data-locale` like this:

```
<component tag="my-thing" :#locale>
  <template>
    <button .click="#setLocale" data-locale="en">EN</button>
    <button .click="#setLocale" data-locale="fr">FR</button>
    <button .click="#setLocale" data-locale="es">ES</button>
  </template>
  <script>
    #setLocale(event) {
      this.#local = event.target.dataset.locale;
    }
  </script>
</component>
```

Pipes

Property variables can use pipes to alter or format data without affecting the original values. Use the bar character (**|**) to searate pipes

- Like `:name="name|upperCase"` or `:name="name|#upperCase|#reverse"`
- All pipe methods take a single parameter as treat it as a `string`. The methods must return a `string`.

> ***TODO:*** *This may need to take whatever type the var is and return anything*

Example pipes

Pipes are very easy to write. You have only one incoming parameter and your pipe must return a string. Here are two example pipes:

```
 #upperCase(value) {
   return value.toUpperCase();
 }

 #revserse(value) {
   return [...value].reverse().join('');
 }
```

Conditionals

You can use `$if="variable"` or `$if="!variable"` to conditionaly hide and show sections of the html template.

```
<component tag="if-one" :state="bool:true">
  <template>
    <div>this.state is currently set to <span :text="state"></span></div>
    <button .click="#toggleState">Toggle</button>
    <div class="red" $if="state">TRUE - This shows if this.state is set to true.
      <p>Go watcha fun movie!</p>
    </div>
    <div class="blue" $if="!state">FALSE - If this.state is set to false then this sho
      <p>Listen to an audio book.</p>
    </div>
  </template>
  <style>
    .red {
      background-color: #F00;
      margin: 20px;
      padding: 20px;
    }
    .blue {
      background-color: #00F;
      color: #FFF;
      margin: 20px;
      padding: 20px;
    }
  </s>
  <script>
    #toggleState(event) {
      this.state = !this.state;
    }
  </script>
</component>
```

File `EvoElement.js`

There are three things exported from the file `EvoElement.js`.

- The class `EvoElement` - This is the base class for all of the generated Web Components.

- The function `setAttr` - A helper function that calls either `setAttribute` or `removeAttribute` based on the value passed in.
- The function `handleCondition` - A helper function that hides and shows a DOM element based on a value of the `condition` parameter.

Exported class `EvoElement`

This is used by Evo WC as the base class for all component. Normally you will not use this base class yourself.

In your component class you must not override the built in methods `connectedCallback`, `disconnectedCallback`, `adoptedCallback`, and `attributeChangedCallback`. Instead you will create your own methods `connected`, `disconnected`, `adopted`, and `attrChanged`.

Exported function `setAttr(el, attr, value)`

- `el` is the element to be affected.
- `attr` is the attribute to be set or reset.
- `value` is the value to set for the attribute.
  - If `value` is `null` then the attribute will be removed from the element `el`
  - Any other value will set the attribute to that value.

`setAttr` is a helper function that calls `el.removeAttribute(attr)` if you pass in `null` as the `value`. If the `value` is `true` then it calls `el.setAttribute(attr, '')`. For all other values it calls `el.setAttribute(attr, value)`.

While it is not likely that you will need to, you can call `setAttr` from your own code. For example you can do the following to set the `name` attribute on your component:

```
setAttr(this, 'name', 'SomeValue');
```

Exported function `handleCondition(el, condition, commentEl)`

`handleCondition` is a helper function that is used with conditional attributes `$if` and `$switch`.

> 2023-02-27 - `$switch` works but is poorly designed and will be replaced.

You pass in the element `el` that is conditionally to hide or show, the conditional value `condition` that is `true` or `false` and the comment element `commentEl` that will replace the element if the condition is `false`. It is important to have a unique comment element for each conditional.

`handleCondition` will place either the element or the comment element into the DOM based on the condition.

> Normally, this is not to be called by your code.

Notes of things that still need to be finished

Date Types

- Support bigint

Compile options

- Debug Mode that adds lost of debug code.
- Generates JSDOC comments
- Provide a way to add doc info into the template
- On compile error: set the output of the JS file to display the error message in the UI
- Do I allow TypeScript in the code??
- SourceMaps? Would they work?

Pipes

- Regular pipes work.
- Importable pipes?
  - Is this just an import and then calling that import from a member function?
  - Is there a special way to indicate an imported pipe?

Conditional Attributes

- Add docs that explain how to use the `state` attribute and CSS to improve performance
- Conditionals should really only be used when large chunks of DOM are involved.
- `$if` works
- `$swich` is coming
- Add Looping (`$for`)
  - require a *key* field.
  - Improve it to *limit DOM changes* (Reuse DOM where possible)
- Only allow one conditional attribute per element
- Should I add any others??

- https://angular.io/api/common#directives

Things to prevent

- ERROR: Do not allow `<script>` in the template or style sections
- ERROR: Do not allow `<style>` in the template or script sections
- ERROR: Do not allow any `on????` event attributes in the template.
  - Indicate that they need to convert to a `.event` handler instead.

Other language version (increase tool usage)

Sometime in the near future we plan to release translpilers in different languages since not all backends are written in JavaScript. We want to support as many tools and lanuguages as possible. Here is the suggested list:

- .NET
- Java
- Python
- Gulp
- Grunt
- Webpack
- Babble
- Rollup

Element properties that are renamed or changed to attributes:

converted from	converted to
accept	attr.accept
alt	attr.alt
for	attr.for
html	innerHTML
class	className
style	attr.style
text	textContent

> 2023-02-27 - List not complete

Reference

- Custom elements
- JavaScript private class fields
- Web Components Can Now Be Native Form Elements
- How to style shadow DOM with ::part() and ::slotted()