

Creating Web Components with Stencil

Web Components

Have you ever struggled with integrating UI components implemented for different JavaScript frameworks or libraries, say, for example, *Angular* or *React* or *Vue* or whatever?

Are you tired of reimplementing the same UI component for each new framework or library?

Do you know that a solution to this problem has already existed for some years?

It is called [Web Components](#).

Web Components are a set of [standard specifications](#) that allow you to create custom and reusable components by simply using HTML, CSS, and JavaScript

This should be the end of any JavaScript library interoperability nightmare, but... but there are still a few problems: mainly [the browser support for all the features](#) and the low level of Web Components APIs.

Web Components

Have you ever struggled with integrating UI components implemented for different JavaScript frameworks or libraries, say, for example, *Angular* or *React* or *Vue* or whatever?

Are you tired of reimplementing the same UI component for each new framework or library?

Do you know that a solution to this problem has already existed for some years?

It is called [Web Components](#).

Web Components are a set of [standard specifications](#) that allow you to create custom and reusable components by simply using HTML, CSS, and JavaScript

Stencil

[Stencil](#) is an open-source compiler that generates standards-compliant web components. [Team Ionic](#) announced Stencil during the [Polymer Summit 2017](#).

Stencil compiles components into pure web components which can be used in other frameworks like [Preact](#), [React](#), and even with no framework at all.

Why Use Stencil

Why use Stencil?

We can build web components using vanilla JS of course. But Stencil provides some syntactic sugar with TSX ([JSX](#) with [TypeScript](#)) that makes it a lot easier to build web components with cleaner, reduced code.

Stencil's core API can be used to create components, manage state with the component lifecycle methods, and inputs and outputs to pass attributes into components and emit events from the component respectively.

Setup of the Stencil Environment

To become familiar with *Stencil*, we are going to build a rating Web Component, that is a UI component allowing the user to provide their feedback about a product, an article or whatever by assigning number of stars like in the following picture:



As a first step towards this goal, you need to set up the *Stencil* development environment. So, be sure to get [Node.js](#) installed on your machine and then type the following command in a terminal window:

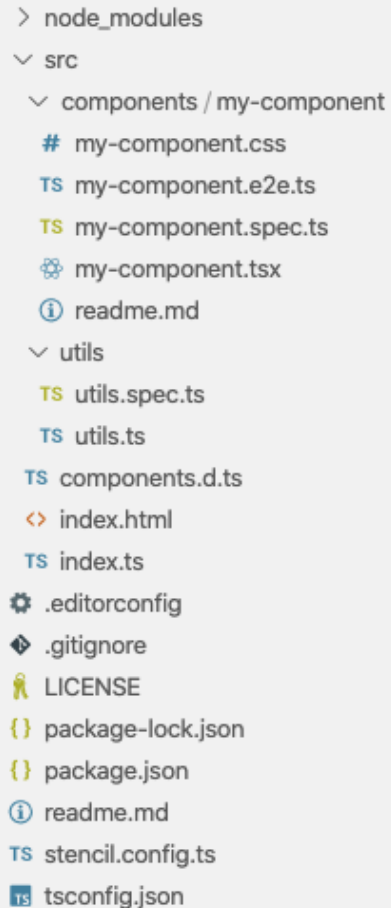
```
npm init stencil component rating-stencil-component
```

This command creates a new Stencil component project by using the component starter. It creates the project into the `rating-stencil-component` folder.

Setup of the Stencil Environment

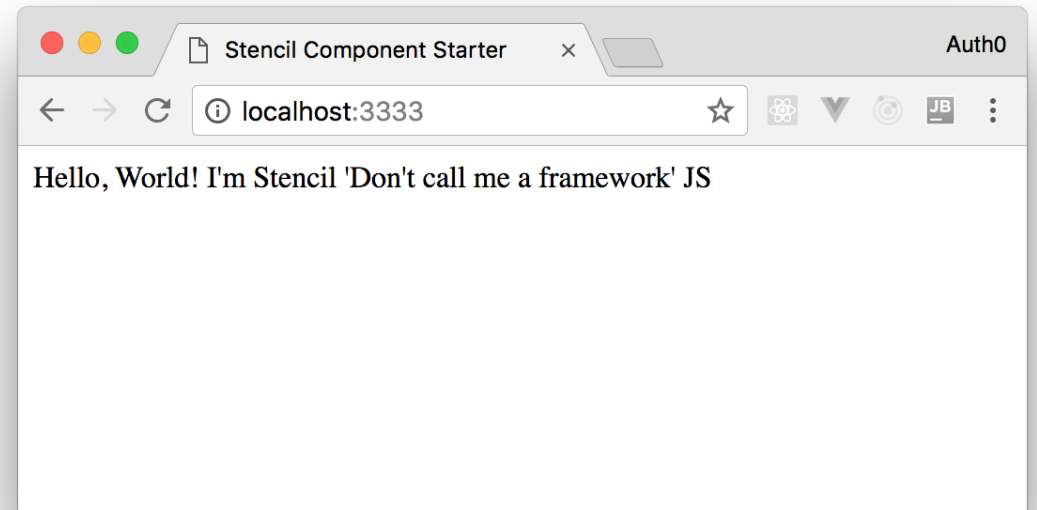
The component starter project provides a standard Node.js development environment.

In particular, you can see a few configuration files in the root folder and the *src* folder containing a folder structure, as shown in the following picture:



```
> node_modules
  < src
    < components / my-component
      # my-component.css
      TS my-component.e2e.ts
      TS my-component.spec.ts
      my-component.tsx
      readme.md
    < utils
      TS utils.spec.ts
      TS utils.ts
    TS components.d.ts
    < index.html
    TS index.ts
    .editorconfig
    .gitignore
    LICENSE
    {} package-lock.json
    {} package.json
    readme.md
    TS stencil.config.ts
    tsconfig.json
```

The component starter project contains a very basic and working component that you can see in action by typing `npm start`. After a few seconds a browser window like the following will be open:



Creating a Basic Stencil Component

Let us build a component by exploiting the infrastructure of this basic project.

<https://auth0.com/blog/creating-web-components-with-stencil/>

In order to implement our rating component, let's create a `my-rating-component` folder inside the `/src/components` folder.

In this newly created folder, create a file named **`my-rating-component.tsx`** and **`my-rating-component.css`**

`my-rating-component.css`

```
.rating {  
  color: orange;  
}
```


Creating a Basic Stencil Component

my-rating-component.tsx

```
import { Component, h } from '@stencil/core';
@Component({
  tag: 'my-rating',
  styleUrls: 'my-rating-component.css',
  shadow: true
})
export class MyRatingComponent {
  render() {
    return (
      <div>
        <span class="rating">&#x2605;</span>
        <span class="rating">&#x2605;</span>
        <span class="rating">&#x2605;</span>
        <span class="rating">&#x2606;</span>
        <span class="rating">&#x2606;</span>
        <span class="rating">&#x2606;</span>
      </div>
    );
  }
}
```

There are 6 span elements, three of them contain the HTML entity for the full star (★), and the other three contain the code for the empty star (☆)

A Stencil component is a TypeScript class, MyRatingComponent which implements a render() method.

This class is decorated by the @Component decorator, previously imported from the stencil/core module. This decorator allows you to define some meta-data about the component itself.

In particular, we defined the tag name that will be associated with the component. This means that we will use the <my-rating></my-rating> element to put this component inside an HTML page.

We also defined the CSS file containing styling settings for the component via the styleUrls property.

The last property, shadow, isolates the internal component DOM and styles so that it is shielded by name conflicts and accidental collisions.

Note that the h() function has been imported from the stencil/core module. This function is needed to allow Stencil to turn the JSX into Virtual DOM elements. In fact, Stencil uses a tiny Virtual DOM layer similar to the one used by React.

Manually Testing a Stencil Component

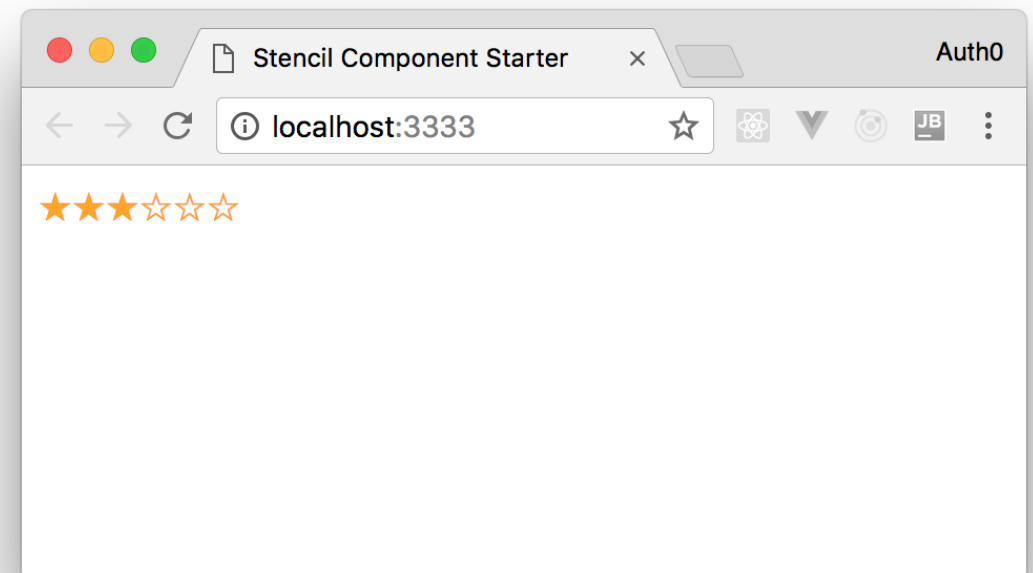
Remove the default component included in the starter component project i.e. the `/src/my-component` folder.

Then open the `index.html` file in the `src` folder and replace its content with the following markup:

```
<!DOCTYPE html>
<html dir="ltr" lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0, minimum-scale=1.0, maximum-scale=5.0">
  <title>Stencil Component Starter</title>
  <script type="module" src="/build/rating-stencil-
component.esm.js"></script>
  <script nomodule src="/build/rating-stencil-
component.js"></script>
</head>
<body>
  <my-rating></my-rating>
</body>
</html>
```

The body of the HTML page contains the newly defined `<my-rating>` tag that identifies our component.

After saving the `index.html` file, type `npm start` in a console window, and you will see your component in a browser as in the following picture:



Adding Properties to Stencil Components

Remove the default component included in the starter component project i.e. the `/src/my-component` folder. Then open the `index.html` file in the `src` folder and replace its content with the following markup:

```
// src/components/my-rating-component/my-rating-component.tsx
```

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

```
@Component({  
  tag: 'my-rating',  
  styleUrls: 'my-rating-component.css',  
  shadow: true  
})
```

```
export class MyRatingComponent {  
  @Prop() maxVal: number = 5;  
  @Prop() value: number = 0;
```

```
  createStarList() {  
    let starList = [];
```

```
    for (let i = 1; i <= this.maxVal; i++) {  
      if (i <= this.value) {  
        starList.push(<span class="rating">&#x2605;</span>);  
      } else {  
        starList.push(<span class="rating">&#x2606;</span>);  
      }  
    }  
  }
```

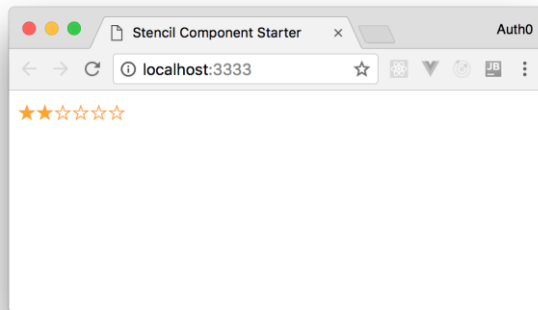
```
  return starList;  
}
```

```
render() {  
  return (  
    <div>  
      {this.createStarList()}  
    </div>  
  );  
}
```

We have imported the **@Prop()** decorator. This decorator allows you to map the properties of the component class to attributes in the markup side of the component.

We added the **maxValue** property, which represents the maximum number of stars to show, and the **value** property, which indicates the current rating value and so the number of full stars to be shown.

Each property has a default value. These properties decorated with **@Prop()** allows to use the component's markup as follows:



<my-rating max-value="6" value="2"></my-rating>

The Reactive Nature of Properties on Stencil

The component's properties are not only a way to set customized initial values through HTML attributes. The mapping between the attributes and the properties is reactive. This means that any change to the attribute fires the `render()` method so that the component's UI is updated.

We can verify this behavior by changing the content of the `index.html` file as follows

```
<!-- src/index.html -->
<!DOCTYPE html>
<html dir="ltr" lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=1.0, maximum-scale=5.0">
  <title>Stencil Component Starter</title>
  <script type="module" src="/build/rating-stencil-component.esm.js"></script>
  <script nomodule src="/build/rating-stencil-component.js"></script>
  <script>
    setTimeout(function() {
      let myRatingComponent = document.getElementById("myRatingComponent");
      myRatingComponent.value = 4;
    }, 5000)
  </script>
</head>
<body>
  <my-rating id="myRatingComponent" max-value="6" value="2"></my-rating>
</body>
</html>
```

We assigned an **id** attribute to the component's markup and added a script block calling the **setTimeout()** JavaScript function that schedules the execution of a function after 5 seconds. The scheduled function changes the `value` property of the component.

So we will see your rating component with an initial number of two full stars, and after five seconds, you will see it with four full stars.

Managing State of Stencil Components

We want to add more interactivity to your rating component.

We want the number of full stars of the component to follow the mouse movement when it is over it. It should return to its original number when the mouse is out of its area, like in the following animation:



In addition, we want to set a new value when the user clicks on one of the component's stars.

In order to manage this dynamic change of stars, we can assign an internal state to our component.

The state of a component is a set of data internally managed by the component itself. This data cannot be directly changed by the user, but the component can modify it according to its internal logic. Any change to the state causes the execution of the `render()` method.

Managing State of Stencil Components

Stencil allows us to define the component state through the `@State()` decorator, so we can add a new property to internally track the stars to display in a given moment

```
// src/components/my-rating-component/my-rating-component.tsx
```

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

```
@Component({
  tag: 'my-rating',
  styleUrls: 'my-rating-component.css',
  shadow: true
})
export class MyRatingComponent {
  @Prop() maxValue: number = 5;
  @Prop() value: number = 0;
```

```
  @State() starList: Array<object> = [];
```

```
  createStarList() {
    let starList = [];

    for (let i = 1; i <= this.maxValue; i++) {
      if (i <= this.value) {
        starList.push(<span class="rating">&#x2605;</span>);
      } else {
        starList.push(<span class="rating">&#x2606;</span>);
      }
    }
  }
```

```
  this.starList = starList;
}
```

```
render() {
  return (
    <div>
      {this.starList}
    </div>
  );
}
```

With respect to the previous version, this code imports the `@State()` decorator and apply it to the newly introduced `starList` property.

This property is an array of objects and represents the component state that will contain the JSX description of the stars to display.

Consequently, the **`createStarList()`** method has been modified so that the resulting array is assigned to the state property.

Finally, the **`starList`** property is used inside the JSX expression returned by the `render()` method

