FREE eBook

LEARNING Web Component

Free unaffiliated eBook created from
Stack Overflow contributors.

#web-component

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: web-component

It is an unofficial and free Web Component ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Web Component.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Web Component

Remarks

This section provides an overview of what Web Components are, and why a developer might want to use them.

Web Components are a set of new web technologies implemented in modern web browsers, and used to design reusable web elements with the only help of HTML, JavaScript and CSS.

Topics covered by the term *Web Compoments* are:

- Custom Elements
- HTML Templates
- Shadow DOM
- HTML Imports

These technologies are complementary, and can be used together or separately.

Versions

Components	Specification	Last Release
HTML Templates	W3C HTML5 Recommendation	2014-10-28
Custom Elements	W3C Working Drafts or WHATWG HTML and DOM Living Standard	2016-10-13
Shadow DOM	W3C Working Drafts or WHATWG HTML and DOM Living Standard	2017-01-16
HTML Imports	W3C Working Drafts	2016-02-25

Examples

Availability

Native implementations

The `<template>` element is implemented in every modern browsers:

- Chrome,
- Edge,
- Firefox,
- Opera,
- Safari,
- ...

Custom Elements `customElements.define()`, Shadow DOM `attachShadow()` and HTML Imports `<link rel="import">` are implemented in the latest versions of Chrome and Opera.

Polyfills

For other browsers, you can use a polyfill library:

- for Custom Elements: from WebReflection or Webcomponents.org,
- for Shadow DOM: from Webcomponents.org,
- for Template : from Neovov,
- for HTML Imports: from Webcomponents.org

HTML Template - Hello World

Use a `<template>` element to design a HTML template that you can then reuse in your code.

```
<template id="Template1">
    Hello, World !
<template>

<div id="Target1"></div>

<script>
   Target1.appendChild( Template1.content.cloneNode( true ) )
</script>
```

This will insert the content of the template in the `#Target1` div.

Custom Element - Hello World

Create a new HTML tag named `<hello-world>` that will display "Hello, World!":

```
<script>
//define a class extending HTMLElement
class HelloWorld extends HTMLElement {
    connectedCallback () {
      this.innerHTML = 'Hello, World!'
    }
}

//register the new custom element
customElements.define( 'hello-world', HelloWorld )
</script>
```

```
<!-- make use the custom element -->
<hello-world></hello-world>
```

Shadow DOM - Hello World

Add a Shadow DOM to a `div` that will display "Hello, World!" instead of its initial content.

```
<div id="Div1">intial content</div>

<script>
   var shadow = Div1.attachShadow( { mode: 'open' } )
   shadow.innerHTML = "Hello, World!"
</script>
```

HTML Import - Hello World

Import an HTML file that will add a div with "Hello, World!" at the end of the main document's DOM tree.

Imported file *hello.html*:

```
<script>
   var div = document.createElement( 'div' )
   div.innerHTML = 'Hello, World!'
   document.body.appendChild( div )
</script>
```

Main file *index.html*:

```
<html>
  <link rel="import" href="hello.html">
```

Hello World example

This example combines Custom Element, Template, Shadow DOM and HTML Import to display a the "Hello, World!" string in HTML.

In file `hello-world.html`:

```
<!-- 1. Define the template -->
<template>
   Hello, World!
</template>

<script>
  var template = document.currentScript.ownerDocument.querySelector( 'template' )

  //2. Define the custom element

  customElements.define( 'hello-world', class extends HTMLElement
  {
```

```
      constructor()
      {
          //3. Create a Shadow DOM
          var sh = this.attachShadow( { mode: 'open' } )
          sh.appendChild( document.importNode( template.content, true ) )
      }
  } )
</script>
```

In main file `index.html`:

```
<html>
<head>
    <!-- 4. Import the HTML component -->
    <link rel="import" href="hello-world.html">
</head>
<body>
    <hello-world></hello-world>
</body>
</html>
```

Read Getting started with Web Component online: https://riptutorial.com/web-component/topic/8239/getting-started-with-web-component

Chapter 2: Testing Web Components

Introduction

Things to consider when we want to test our components with: Styles, Templates, Component classes.

Examples

Webpack and Jest

Jest is used by Facebook to test all JavaScript code including React applications. One of Jest's philosophies is to provide an integrated "zero-configuration" experience. We observed that when engineers are provided with ready-to-use tools, they end up writing more tests, which in turn results in more stable and healthy code bases.

Full working example is available on GitHub as web-components-webpack-es6-boilerplate

Jest runs tests in NodeJS enviroment with jsdom. The whole process is easy. Let's consider following webpack setup, assuming our project structure looks like a following example:

```
-src
    --client
    --server
-webpack
    --config.js
package.json
```

A simple directory structure designed to separate the `server render` logic from the rest. **Webpack** `config.js` file would contain following modules:

```
  resolve: {
    modules: ["node_modules"],
    alias: {
      client: path.join(__dirname, "../src/client"),
      server: path.join(__dirname, "../src/server")
    },
    extensions: [".js", ".json", ".scss"]
  },
```

We can set up **Jest** to reflect our **Webpack** config.

```
module.exports = {
  setupTestFrameworkScriptFile: "<rootDir>/bin/jest.js",
  mapCoverage: true,
  moduleFileExtensions: ["js", "scss", "html"],
  moduleDirectories: ["node_modules"],
  moduleNameMapper: {
    "src/(.*)$": "<rootDir>/src/$1"
```

```
  },
  transform: {
    "^.+\\.(js|html|scss)$": "<rootDir>/bin/preprocessor.js"
  },
  testMatch: ["<rootDir>/test/**/?(*.)(spec|test).js"],
  testPathIgnorePatterns: ["<rootDir>/(node_modules|bin|build)"]
};
```

Where should we save this config ?

We can do it in the `package.json` file under `jest` key or create as in this example `jest.config.js` file in the project root.

What we want to achieve is to make sure that our `html` files are going to be imported correctly. That means by escaping them with custom `preprocessor`, as using only `babel-jest` would throw error when trying to parse non `js` files.

The other important thing here is `setupTestFrameworkScriptFile` script which actually includes `custom elements` polyfills to `jsdom`. Here is how our `preprocessor.js` looks like:

```
const babelJest = require("babel-jest");

const STYLE_URLS_REGEX = /styles:\s*\[\s*((?:'|").*\s*(?:'|")).*\s*.*\]/g;
const ESCAPE_TEMPLATE_REGEX = /(\${|\`)/g;

module.exports.process = (src, path, config) => {
  if (path.endsWith(".html")) {
    src = src.replace(ESCAPE_TEMPLATE_REGEX, "\\$1");
    src = "module.exports=`" + src + "`;";
  }
  src = src.replace(STYLE_URLS_REGEX, "styles: []");

  return babelJest.process(src, path, config);
};
```

What this script does, is simple: remove style files content as we do not need/want to test it, and escape templates, when we import them for example with `require('template.html')` syntax. Then it passes down content to babel transformer.

Last important thing to do is to include `web components polyfills`. As by default `jsdom` does not support them yet. To do it we can simply add `setupTestFrameworkScriptFile` in our example it is `jest.js` with the following content:

```
require("document-register-element/pony")(window);
```

This way we can access `web components` **API** in `jsdom`.

After setting up everything we should have structure like this:

```
-bin
    --jest.js
    --preprocessor.js
-src
```

```
    --client
    --server
-webpack
    --config.js
-test
package.json
jest.config.js
```

Where we keep our tests in the `test` directory and can run it with command: `yarn run jest --no-cache --config $(node jest.config.js).`

Read Testing Web Components online: https://riptutorial.com/web-component/topic/10057/testing-web-components

Credits

S. No	Chapters	Contributors
1	Getting started with Web Component	Community, Mike, Supersharp
2	Testing Web Components	Vardius