Kostenloses eBook

LERNEN Web Component

Free unaffiliated eBook created from
Stack Overflow contributors.

#web-component

Inhaltsverzeichnis

Über

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.

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

Kapitel 1: Erste Schritte mit der Webkomponente

Bemerkungen

Dieser Abschnitt enthält eine Übersicht über die Webkomponenten und warum ein Entwickler sie verwenden möchte.

Webkomponenten sind eine Reihe neuer Webtechnologien, die in modernen Webbrowsern implementiert werden und zum Erstellen wiederverwendbarer Webelemente mit der einzigen Hilfe von HTML, JavaScript und CSS verwendet werden.

Unter dem Begriff *Web-Komponenten* werden folgende Themen behandelt:

- Benutzerdefinierte Elemente
- HTML-Vorlagen
- Schatten-DOM
- HTML-Importe

Diese Technologien ergänzen sich und können zusammen oder getrennt verwendet werden.

Versionen

Komponenten	Spezifikation	Letzte Veröffentlichung
HTML-Vorlagen	W3C HTML5-Empfehlung	2014-10-28
Benutzerdefinierte Elemente	W3C Working Drafts oder WHATWG HTML und DOM Living Standard	2016-10-13
Schatten-DOM	W3C Working Drafts oder WHATWG HTML und DOM Living Standard	2017-01-16
HTML-Importe	W3C-Arbeitsentwürfe	2016-02-25

Examples

Verfügbarkeit

Native Implementierungen

Das `<template>` -Element ist in jedem modernen Browser implementiert:

- Chrom,
- Kante,
- Feuerfuchs,
- Oper,
- Safari,
- ...

Benutzerdefinierte Elemente `customElements.define()` , Shadow DOM `attachShadow()` und HTML-Importe `<link rel="import">` sind in den neuesten Versionen von Chrome und Opera implementiert.

Polyfills

Für andere Browser können Sie eine Polyfill-Bibliothek verwenden:

- für benutzerdefinierte Elemente: von WebReflection oder Webcomponents.org ,
- für Shadow DOM: von Webcomponents.org ,
- für Vorlage: aus Neovov ,
- für HTML-Importe: von Webcomponents.org

HTML-Vorlage - Hallo Welt

Verwenden Sie ein `<template>` -Element, um eine HTML-Vorlage zu entwerfen, die Sie anschließend in Ihrem Code wiederverwenden können.

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

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

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

Dadurch wird der Inhalt der Vorlage in das `#Target1` Div `#Target1` .

Benutzerdefiniertes Element - Hallo Welt

Erstellen Sie ein neues HTML-Tag mit dem Namen `<hello-world>` , das "Hallo, Welt!" Anzeigt:

```
<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 - Hallo Welt

Füge ein Schatten-DOM zu einem `div` , das "Hello, World!" `div` anstelle des ursprünglichen Inhalts.

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

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

HTML-Import - Hallo Welt

Importieren Sie eine HTML-Datei, die ein Div mit "Hallo, Welt!" am Ende des DOM-Baums des Hauptdokuments.

Importierte Datei *hello.html* :

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

Hauptdatei index.html :

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

Hallo Weltbeispiel

In diesem Beispiel werden benutzerdefiniertes Element, Vorlage, Schatten-DOM und HTML-Import kombiniert. Zeichenfolge in HTML.

In der Datei `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 der Hauptdatei `index.html` :

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

Erste Schritte mit der Webkomponente online lesen: https://riptutorial.com/de/web-component/topic/8239/erste-schritte-mit-der-webkomponente

Kapitel 2: Webkomponenten testen

Einführung

Dinge, die zu berücksichtigen sind, wenn wir unsere Komponenten mit folgenden Klassen testen möchten: Stile, Vorlagen, Komponentenklassen

Examples

Webpack und Jest

Jest wird von Facebook verwendet, um den gesamten JavaScript-Code einschließlich der React-Anwendungen zu testen. Eine der Philosophien von Jest ist es, eine integrierte "Zero-Configuration" Erfahrung zu bieten. Wir haben beobachtet, dass Ingenieure mit einsatzbereiten Tools mehr Tests schreiben, was zu stabileren und gesünderen Codebasen führt.

Ein voll funktionsfähiges Beispiel ist auf GitHub als web-components-webpack-es6-boilerplate verfügbar

Jest führt Tests in NodeJS-Umgebung mit Jsdom durch . Der ganze Prozess ist einfach. Betrachten wir folgendes Webpack- Setup, vorausgesetzt, unsere Projektstruktur sieht wie folgt aus:

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

Eine einfache Verzeichnisstruktur, mit der die `server render` des `server render` vom Rest getrennt wird. **Die** Datei " **Webpack** `config.js` " enthält folgende Module:

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

Wir können **Jest** so einrichten, dass es unserer **Webpack-** Konfiguration entspricht.

```
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)"]
};
```

Wo soll diese Konfiguration gespeichert werden?

Wir können dies in der `package.json` Datei unter `jest` key tun oder wie in diesem Beispiel die `jest.config.js` Datei im Projektstammverzeichnis erstellen.

Was wir erreichen wollen, ist sicherzustellen, dass unsere `html` Dateien korrekt importiert werden. Dies bedeutet, dass durch das Umgehen mit einem benutzerdefinierten `preprocessor` die Verwendung von nur `babel-jest` einen Fehler `js` wenn versucht wird, Nicht- `js` Dateien zu analysieren.

Die andere wichtige Sache hier ist `setupTestFrameworkScriptFile` Skript `setupTestFrameworkScriptFile` das tatsächlich `custom elements` , die `jsdom` in `jsdom` . So sieht unser `preprocessor.js` aus:

```
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);
};
```

Was dieses Skript macht, ist einfach: Entfernen Sie den Inhalt von Style-Dateien, da wir ihn nicht testen müssen und wollen, und entziehen Sie Vorlagen, wenn wir sie zum Beispiel mit der Syntax „ `require('template.html')` importieren. Dann wird der Inhalt an den Babel-Transformator weitergeleitet.

Das Wichtigste, das Sie tun müssen, ist das `polyfills` für `web components` . Standardmäßig werden sie von `jsdom` noch nicht unterstützt. Dazu können wir einfach `setupTestFrameworkScriptFile` hinzufügen `setupTestFrameworkScriptFile` In unserem Beispiel handelt es sich um `jest.js` mit dem folgenden Inhalt:

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

Auf diese Weise können wir auf die `web components` **API** in `jsdom` .

Nachdem wir alles eingerichtet haben, sollten wir folgende Struktur haben:

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

Wo wir unsere Tests im halten `test` und es mit dem Befehl ausführen können: `yarn run jest --no-cache --config $(node jest.config.js) .`

Webkomponenten testen online lesen: https://riptutorial.com/de/web-component/topic/10057/webkomponenten-testen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit der Webkomponente	Community, Mike, Supersharp
2	Webkomponenten testen	Vardius