Integrazione di React nel Framework

Sommario

[Web-components 3](#_Toc34922764)

[Integrazione 4](#_Toc34922765)

[React e web-components 4](#_Toc34922766)

[Data binding 4](#_Toc34922767)

[Stile 4](#_Toc34922768)

[Cookie e storage 5](#_Toc34922769)

[File system 5](#_Toc34922770)

[Caricamento dei tag 6](#_Toc34922771)

[Esempi 6](#_Toc34922772)

[App.jsx 6](#_Toc34922773)

[App-Bridge.jsx 6](#_Toc34922774)

[presenter.js 8](#_Toc34922775)

[template.html 8](#_Toc34922776)

[Librerie 9](#_Toc34922777)

[System 9](#_Toc34922778)

[Workspaces 10](#_Toc34922779)

[Link utili 12](#_Toc34922780)

# Web-components

L’integrazione di React all’interno del framework, è possibile tramite l’uso dei web component. Questa tecnologia permette di estendere i componenti HTML esistenti o di crearne di nuovi, indifferentemente dalla libreria o framework JS in uso.

Se l’implementazione all’interno di codice HTML o JS è identica a quella di un normale tag, la dichiarazione di un web component avviene tramite l’uso di:

**customElements.define("my-element", MyElement);**

dove **my-element** corrisponderà alla sintassi HTML, mentre **MyElement** alla classe JS che definisce l’elemento.

# Integrazione

## React e web-components

Una volta definita la classe dell’elemento web component, è possibile inserire il container dell’applicativo React nel DOM in due modi:

* Direttamente tramite innerHTML. Questa procedura permette di riutilizzare il CSS globale all’interno dell’applicativo React, ma non ne permette una separazione fisica dal resto dell’applicazione, con la conseguenza di potenziali e rischiose interferenze tra le due parti
* Tramite Shadow DOM. L’uso dello Shadow DOM garantisce una separazione netta tra il DOM primario dell’applicazione e il DOM secondario relativo all’applicativo React; questo permette di incapsulare e riutilizzare HTML, CSS e JS in più parti del codice senza dover tenere in considerazione eventuali conflitti nel codice, a discapito dell’uso di CSS globale all’interno del web component

Definito il container di React, implementare la funzione di **render** e di **unmountComponentAtNode** fornite da ReactDOM all’interno dei seguenti metodi del web component:

* *render 🡪 connectedCallback e attributeChangedCallback*
* *unmountComponentAtNode 🡪 disconnectedCallback*

## Data binding

Per eseguire il binding dell’applicativo React con il resto dell’applicazione, sono disponibili due opzioni:

* La prima consiste nell’uso di uno store (es: Redux), ideale nel caso si cerchi di prevenire un re-rendering eccessivo delle componenti React, s’intenda passare dati diversi dal tipo stringa o si cerchi una maggiore integrazione tra i due DOM, a scapito della riutilizzabilità del web component.
* La seconda opzione consiste nello sfruttare gli attributi e gli eventi del web component per generare il binding tra il DOM primario e lo Shadow DOM dove risiede l’applicativo React. Questa implementazione presenta una maggiore complessità, oltre a causare un re-rendering completo dell’applicativo React in caso di aggiornamento dei dati degli attributi e permettere di passare i dati solo in formato stringa, ma garantisce una separazione netta della logica, permettendo di conseguenza di generare una vera e propria “scatola nera”.

## Stile

Nel caso si decida di far uso dello Shadow DOM, non sarà possibile utilizzare lo stile globale per l’applicativo React, ma sarà necessario implementare uno stile locale proprio del web component.

Per implementare lo stile è possibile seguire tre possibili vie:

* Aggiungendo un tag style nello Shadow DOM, sarà possibile indicare direttamente nel codice lo stile proprio del web component
* Aggiungendo un tag link o un tag style (da cui sarà eseguito @import) nello Shadow DOM, sarà possibile importare un foglio di stile che definirà solo e soltanto il CSS del web component
* Utilizzando la classe **CSSStyleSheet** per la gestione e l’import del CSS da iniettare nello Shadow DOM tramite il metodo **adoptedStyleSheet** di **shadowRoot**, sarà generato, a livello dell’HTML, un tag style contente lo stile proprio del web component. \*

Tutte e tre i metodi permettono di fornire uno stile locale proprio al web component, senza intaccare l’implementazione del CSS globale per il resto dell’applicazione.

\*: per motivi di compatibilità con i browser, anche moderni, utilizzare la libreria construct-style-sheets

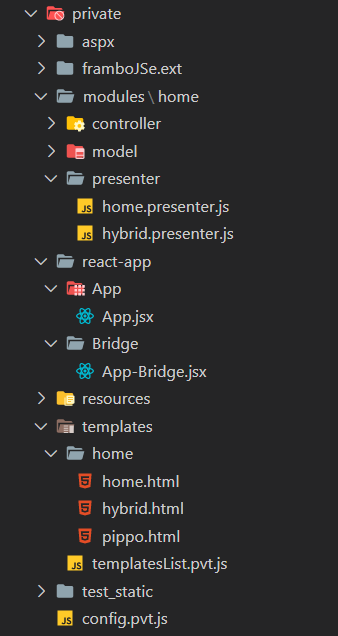
N.B.: nel caso si usi webpack, non eseguire l’import da codice del CSS, verrebbe implementato a livello globale, rendendolo inaccessibile all’interno dello shadow DOM.

## Cookie e storage

A causa della natura intrinseca di React, l’uso di cookie, sessionStorage e localStorage è fortemente sconsigliato, eccezione fatta per dati salvati precedentemente la creazione dell’applicativo React, di cui non si preveda alcun tipo di possibile aggiornamento futuro.

Nel caso invece di cookie con dati “dinamici”, si incorre nel seguente problema: all’aggiornamento dei cookie (o degli storage) non corrisponde un aggiornamento dell’applicazione React, sia per quanto riguarda il rendering della view, che rimarrà immutata, sia per quanto riguarda i dati recuperati dal cookie, che potrebbero, in alcuni casi, non corrispondere a quelli effettivamente presenti in quel momento nel cookie.

## File system

All’interno del workspace, dove si andrà ad aggiungere l’applicazione React, creare una nuova cartella contenente a sua volta le sotto-cartelle **App** e **Bridge** (i nomi appena indicati sono da considerarsi best practice e non vincolanti per il corretto funzionamento dell’integrazione). La cartella App conterrà l’intero applicativo React, mentre all’interno di Bridge sarà salvato il file relativo al web-component tramite il quale sarà caricato l’applicativo all’interno del framework.

## Caricamento dei tag

I tag dei web-components sono da considerarsi alla stregua dei normali tag HTML, per tanto sarà possibile dichiararli sia all’interno di un template che di un presenter del framework, tante volte quante sono il numero di applicazioni React necessarie in quel preciso momento. Unico requisito obbligatorio, sarà necessario caricare il file JS del web-component tramite SystemJS, il quale si occuperà del transpiling del codice; è possibile caricare il file indifferentemente sia da un template che da un presenter, in quest’ultimo caso, ricordarsi però di inserire il percorso del file come ultimo valore dell’array di import delle librerie nel define e di non dichiarare nella callback il relativo parametro, in quanto superfluo.

## Esempi

Di seguito un esempio d’integrazione:

### App.jsx

import React from 'react';

const App = (props) => (

    <div>

        <p>Hello world!</p>

        <p>{props.name}</p>

        <p>{props.value}</p>

    </div>

);

export default App;

### App-Bridge.jsx

import 'CEadapter';

import 'webcomponents';

import React from 'react';

import ReactDOM from 'react-dom';

import retargetEvents from 'retargetEvents';

import App from '../App/App.jsx';

class ReactBridge extends HTMLElement {

    static get observedAttributes() {

        return ['name', 'value'];

    }

    get value() {

        return this.getAttribute('value');

    }

    set value(value) {

        this.setAttribute('value', value);

    }

    get name() {

        return this.getAttribute('name');

    }

    set name(value) {

        this.setAttribute('name', value);

    }

    constructor() {

        super();

        this.attachShadow({ mode: 'open' });

    }

    connectedCallback() {

        this.shadowRoot.innerHTML = '<div id="root"></div>';

        this.container = this.shadowRoot.getElementById('root');

        this.render({ name: this.name, value: this.value });

    }

    disconnectedCallback() {

        ReactDOM.unmountComponentAtNode(this.container);

    }

    attributeChangedCallback(name, oldValue, newValue) {

        switch (name) {

            case 'name': {

                this.render({ name: newValue, value: this.value });

                break;

            }

            case 'value': {

                this.render({ name: this.name, value: newValue });

                break;

            }

        }

    }

    render(data) {

        if (this.container) {

            ReactDOM.render(

                <App name={data.name} value={data.value} />,

                this.container,

                () => retargetEvents(this.shadowRoot)

            );

        }

    }

}

customElements.define("react-bridge", ReactBridge);

### presenter.js

define([

  'base\_presenter',

  'templatesHandler',

  'snippets',

  'modals',

  '/private/react-app/Bridge/App-Bridge.jsx',

], function (pBase, th, snippets, modals) {

*// ... codice ...*

});

### template.html

<reserved\_header\_section templateID="303"

                         templateFile="hybrid.html"

                         templateDescription="HTML content template (hybrid)">

</reserved\_header\_section>

<h1>first title</h1>

<react-bridge name="hello" value="world!"></react-bridge>

*<!-- ... codice HTML ... -->*

<h2>N sub-title</h2>

<react-bridge name="123" value="456"></react-bridge>

# Librerie

## System

Per eseguire una corretta integrazione è necessario installare nella cartella **3rd-parties** di **system**, tramite npm, la libreria **systemjs-plugin-babel**. Quest’ultima si occuperà del transpiling del codice, abilitando il JSX per React e garantendo il corretto utilizzo dell’import/export di ES6.

Come ultimo passo, è necessario modificare la configurazione globale di SystemJS aggiungendo le chiavi **plugin-babel** e **systemjs-babel-build** all’interno di **map**, in modo da caricare le funzionalità della libreria precedentemente installata. Inoltre sarà necessario indicare, all’interno della configurazione, il traspiler, che si occuperà della gestione del codice a runtime, e indicare i relativi file target.

Di seguito un esempio completo:

const \_sysNodeModules = '/System/3rd-parties/node\_modules/';

\_\_SYS\_config = {

*// ... altre configurazioni ...*

  map: {

*// ... altre librerie ...*

    'plugin-babel'        : \_sysNodeModules + 'systemjs-plugin-babel/plugin-babel.js',

    'systemjs-babel-build': \_sysNodeModules + 'systemjs-plugin-babel/systemjs-babel-browser.js',

  },

  meta: {

    '\*.js': {

      babelOptions: {

        stage1: true,

        es2015: true,

        react: true,

      }

    },

    '\*.jsx': {

      babelOptions: {

        stage1: true,

        es2015: true,

        react: true,

      }

    }

  },

  transpiler: 'plugin-babel'

*// ... altre configurazioni ...*

};

Rimando al link in calce per maggiori informazioni riguardo alla libreria

## Workspaces

All’interno di ogni workspace installare, tramite npm, nel percorso **{workspace}/resources/js** le seguenti librerie:

* @webcomponents/webcomponentsjs
* react-shadow-dom-retarget-events

La libreria **@webcomponents/webcomponentsjs** si occuperà del polyfill dei web-components in browser che non li supportano, mentre **react-shadow-dom-retarget-events** permetterà di abilitare gli eventi all’interno dello Shadow DOM contenente l’applicazione React.

Aggiungere infine, tramite CDN \*, le librerie di react e react-dom. L’uso dei CDN, invece di npm, è una scelta obbligata, a causa di problemi relativi alla configurazione delle librerie di React, le quali, se installate tramite npm, obbligano all’uso di webpack (i path delle librerie di supporto puntano ad origin, invece che alla cartella **node\_modules** presente nel workspace).

\*: volendo, è possibile scaricare le librerie, ma sarà necessario salvarle all’interno di una o più cartelle al percorso **{workspace}/resources/js**per una questione di coerenza del file system.

A questo punto, aggiungere le librerie appena installate al file di configurazione del workspace come di seguito:

const pathResources = '/{workspace}/resources/js/';

const pathNode\_modules = pathResources + 'node\_modules/';

var \_\_PVT\_config = {

  map: {

*// ... altre librerie ...*

    React : pathResources + 'React/react.development.js',

    'react-dom' : pathResources + 'React/react-dom.development.js',

    retargetEvents: pathNode\_modules + 'react-shadow-dom-retarget-events/index.js',

    webcomponents : pathNode\_modules + '@webcomponents/webcomponentsjs/webcomponents-loader.js',

    CEadapter : pathNode\_modules + '@webcomponents/webcomponentsjs/custom-elements-es5-adapter.js',

  }

};

Rimando al link in calce per maggiori informazioni riguardo alla libreria

# Link utili

Funzionalità:

* <https://www.html.it/guide/guida-ai-web-components/>
* <https://developer.mozilla.org/it/docs/Web/Web_Components>
* <https://it.reactjs.org/>

Librerie

* **systemjs-plugin-babel**
  + <https://github.com/systemjs/plugin-babel>
  + <https://www.npmjs.com/package/systemjs-plugin-babel>
* **@webcomponents/webcomponentsjs**
  + <https://github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs>
  + <https://www.npmjs.com/package/@webcomponents/webcomponentsjs>
* **react-shadow-dom-retarget-events**
  + <https://github.com/spring-media/react-shadow-dom-retarget-events>
  + <https://www.npmjs.com/package/react-shadow-dom-retarget-events>
* **react e react-dom**
  + <https://it.reactjs.org/docs/cdn-links.html>

Librerie opzionali

* **construct-style-sheets**
  + <https://github.com/calebdwilliams/construct-style-sheets>