2 modi per integrare Angular in un sistema esterno (anche legacy)

Come funzionano gli Angular elements, dove utilizzarli e le tecniche per integrare Angular in sistemi legacy (caso d'uso reale di integrazione in un framework GWT).

In questo articolo impareremo ad utilizzare Angular Elements per estrarre singoli componenti Angular in una libreria e utilizzarli in progetti non Angular come se fossero tag HTML nativi. Poiché però Angular Elements è una tecnologia ancora acerba, vedremo anche una soluzione alternativa per integrare intere applicazioni Angular in sistemi legacy implementati in GWT, ASP o PHP.

Esigenza

Spesso nelle aziende si sviluppano sistemi da molti anni. Un sistema sviluppato diversi anni fa ha alcune caratteristiche:

- usa una tecnologia che era disponibile diversi anni fa, spesso ormai obsoleta
- è complesso al punto che è difficile e costoso ricrearlo sperando che funzioni sin da subito con la stessa affidabilità del precedente

Questo tipo di sistemi va sotto il nome di sistemi *legacy*.

Se è vero quanto detto sopra, è però anche vero che quando aggiungiamo nuove funzioni sarebbe più veloce e, quindi, meno costoso, svilupparle con una tecnologia moderna come Angular.

Esiste allora un modo per integrare in maniera affidabile un nuovo progetto Angular in un sistema esistente? senza ovviamente impattare il progetto e senza comprometterne la funzionalità? In questo modo potremo aggiungere nuove funzioni usando Angular e mano a mano migrare le parti *legacy* verso il nuovo sistema.

Problema

Un'applicazione Angular è generalmente pensata come applicazione SPA, cioè applicazione a singola pagina, che comprende tutte le parti del sistema, dal menù alle varie sezioni.

Difficilmente si riesce a far convivere un'applicazione SPA con una tecnologia come GWT, PHP, Spring, o Asp .net. In questi framework è infatti generalmente prevista un'integrazione importante tra front-end e back-end e il server spesso controlla direttamente la parte HTML. Dal punto di vista del design e della manutenibilità avere back-end e front-end integrati non è la scelta migliore: molto meglio separare le due parti nettamente. Questo è uno dei motivi per i quali le applicazioni SPA hanno avuto successo.

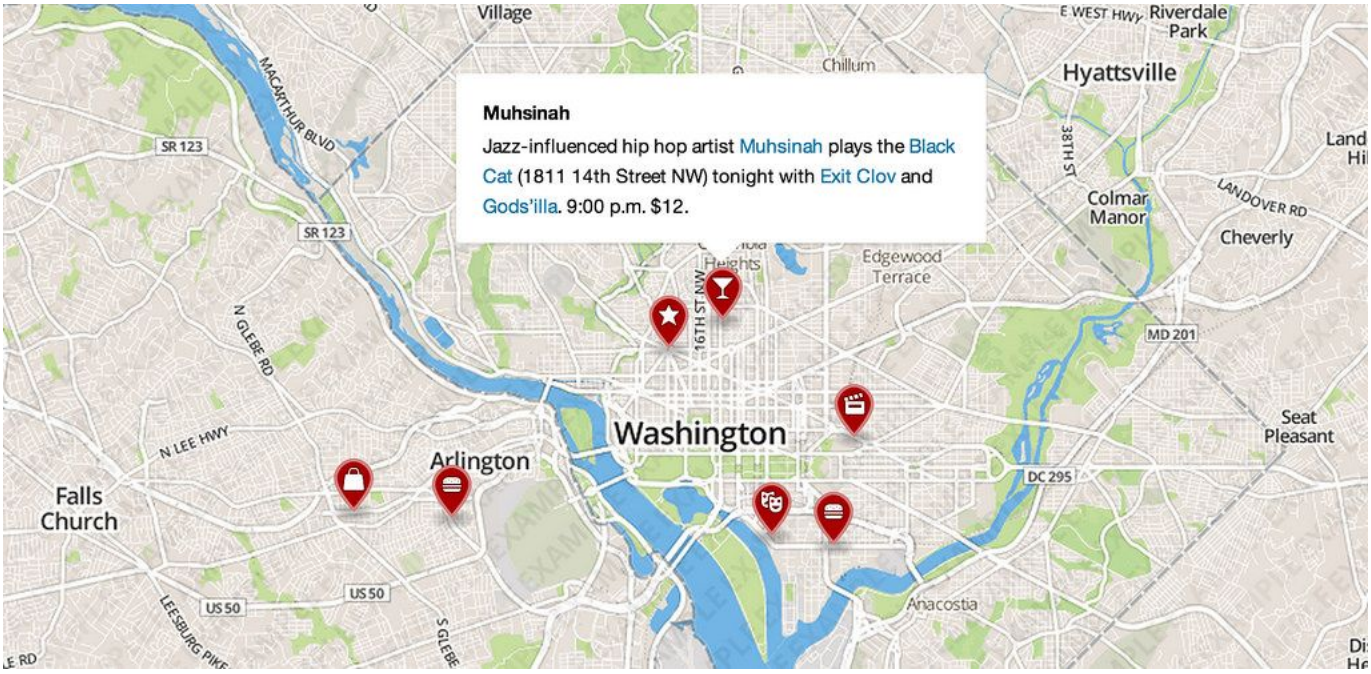
Ci sono fortunatamente oggi alcune tecniche che ci permettono di integrare singoli componenti o intere applicazioni Angular in un sistema preesistente sviluppato con tecnologie GWT, PHP, Spring o Asp.net. Andiamole ad analizzare.

Soluzione con Angular 6: creare librerie di componenti usando Angular Elements

Ormai da qualche anno si parla di *web components* e *custom elements*. L'idea è di estendere HTML con tag custom per avere comportamenti complessi semplicemente usando un tag. Immagina di avere una mappa con dei marker semplicemente scrivendo un codice di questo tipo:

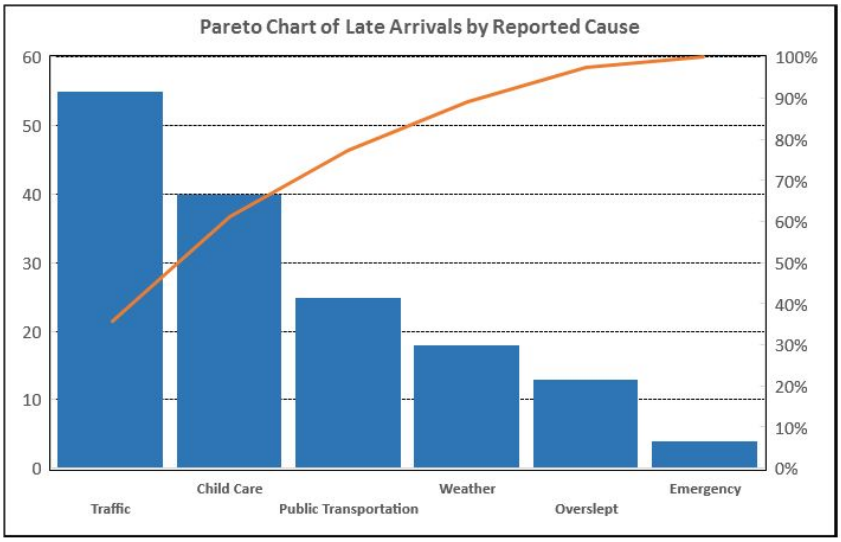
lang:html

```
<map>
  <marker lat="11.1231" lng="12.423"></marker>
  <marker lat="11.2316" lng="11.423"></marker>
  <marker lat="10.1231" lng="10.423"></marker>
  <marker lat="12.1231" lng="11.423"></marker>
</map>
```



Oppure un grafico utilizzando il seguente markup:

```
lang:html
<chart>
  <bar value="55">Traffic</bar>
  <bar value="40">Child care</bar>
  <bar value="25">Public Transportation</bar>
  ...
</chart>
```



Nonostante esiste uno standard, i [web components] (anche detti custom elements), per sviluppare tag custom, ancora non tutti i browser supportano questa tecnologia. Esistono però delle librerie, chiamate *polyfill*, per far funzionare i web components anche con browser più datati.

Potremmo allora usare questi web components in sistemi legacy scrivendo i relativi codici HTML nella pagina.

Se allora abbiamo la possibilità di integrare tag HTML custom, come si sviluppano i web components?

Nella versione 6 di Angular è stata introdotta proprio la possibilità di esportare componenti Angular per usarli come web components. Ancora l'uso di Angular Elements è un po' acerbo e il pacchetto finale generato ammonta a 200-300KB anche per semplici componenti, ma nella versione 7 entrambi questi problemi saranno risolti: il team Angular ha infatti promesso un motore di rendering per creare bundle più snelli e di migliorare il processo di creazione di custom elements.
Vediamo comunque come funziona questa tecnologia.

Creazione di un web component usando Angular Elements

Per prima cosa creiamo un nuovo progetto Angular (tutti i codici sono disponibili su github: https://github.com/devexp-io/codici-articoli/tree/master/angular-elements).

```
lang:cmd
ng new progetto-angular-element
```

Poi installiamo i package *Angular Elements* e Web Components che ci permetteranno di esportare un componente in formato web component:

```
lang:cmd
npm install @angular/elements
npm install @webcomponents/custom-elements
```

Il secondo package, in particolare, ci serve per supportare i browser più datati che non hanno un supporto nativo ai web components. Aggiungiamo quindi degli imports nel file *polyfills.ts*:

```
lang:typescript
// Used for browsers with partially native support of Custom Elements
import '@webcomponents/custom-elements/src/native-shim';

// Used for browsers without a native support of Custom Elements
import '@webcomponents/custom-elements/custom-elements.min';
```

A questo punto possiamo procedere con lo sviluppo del componente. Per l'esempio che segue creeremo un componente di switch, simile nel comportamento ad una checkbox, ma con la grafica di uno switcher.

Usiamo un svg per disegnare lo switcher nello stato attivo o non attivo:

Non è importante investigare il codice SVG per ottenere questa grafica, ma è possibile trovarlo comunque qui:
https://github.com/devexp-io/codici-articoli/blob/master/angular-elements/src/app/switcher/switcher.component.html

Nello switcher definiamo poi un input in modo da settare il valore iniziale dello switcher:

```
lang:typescript
@Input()
value: boolean;
```

Potremmo anche definire un output e, nel caso volessimo richiamare funzioni del componente direttamente dall'oggetto DOM, possiamo definire funzioni con il seguente decorator:

```
lang:typescript
@HostBinding('switched') public currentValueFunction = () => {
 return this.value;
}
```

In questo modo la funzione switched è disponibile nell'istanza DOM.

Per evitare di creare conflitti CSS con altri elementi della pagina, dobbiamo aggiungere una istruzione al decorator @Component:

```
lang:typescript
@Component({
 encapsulation: ViewEncapsulation.Native,
 selector: 'switch-box',
```

Non è un caso che il nome del tag contenga un trattino: i custom elements devono necessariamente contenerne almeno uno per come è implementato lo standard.

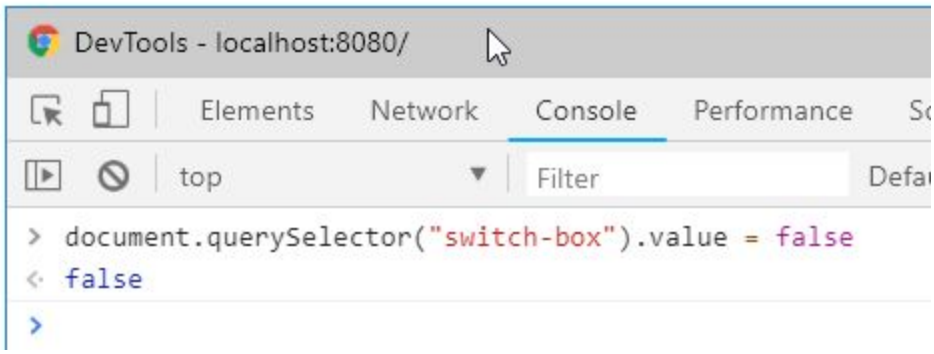
Ora dobbiamo apportare alcune modifiche all'`@NgModule`:

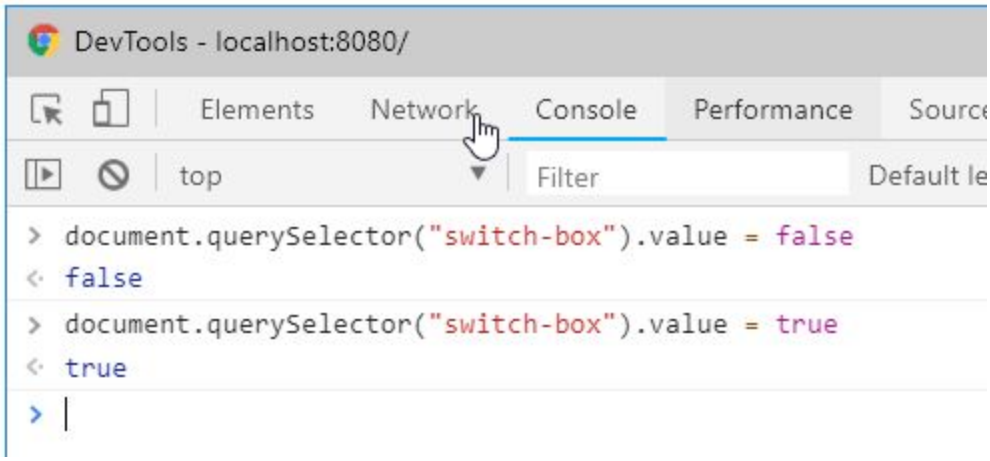
1. poiché non ci serve avviare l'app-root, eliminiamo il campo `bootstrap: [AppComponent],`
2. aggiungiamo il campo `schemas: [CUSTOM_ELEMENTS_SCHEMA],`
3. al fine di esportare il nostro switcher aggiungiamo il campo `entryComponents: [SwitcherComponent]`

Non ci resta che creare il bundle con il solito comando Angular *ng build --prod*
Ora abbiamo i nostri file nella cartella *dist*. Se vogliamo usare il nostro custom element in un file html, dobbiamo importare i tre file javascript *polyfills*, *runtime* e *main*. Poi possiamo usare il seguente codice:

```
lang:html
<switch-box value="true"></switch-box>


<script>
 var switcher = document.querySelector("switch-box")
 console.log(document.querySelector("switch-box").value)

 setTimeout(function () {
   document.querySelector("switch-box").value = false
   console.log(document.querySelector("switch-box").value)
 }, 3000)


</script>
```

Per accedere al nostro custom component basta usare il *query-selector* del *document*. Possiamo accedere al campo *value*, o alla funzione *switched()*. Possiamo anche aggiungere un evento con *addEventListener*.
Dalla *console* ovviamente tutto funziona live:





Come possiamo vedere, i 3 file insieme ammontano a più di 200Kb. E' evidente che ancora la tecnica è acerba e il bundle finale troppo oneroso anche per questo piccolo componente. La tecnologia però è interessante e con Angular 7, che dovrebbe essere pubblicato entro ottobre 2018, potremo usare questa tecnica in maniera semplice ed efficace.

Soluzione per integrare applicazioni complete: postMessage, tab esterni e iframe

Visti i limiti della soluzione precedente, ora andremo ad esplorare un'altra soluzione attualmente utilizzata in un progetto reale.
La soluzione che vi propongo in questa sezione usa delle API HTML per la comunicazione tra Tab ed è stata implementata in un sistema GWT esistente. Con GWT non si accede direttamente al codice HTML ed è difficile l'interoperabilità con altre tecnologie. Come anticipato, tale soluzione è stabilmente utilizzata in un sistema di produzione.

Le API HTML che andremo ad utilizzare sono disponibili su tutti i browser e servono per la comunicazione tra pagine o tra una pagina e un *iframe*. So che ci hanno sempre detto che l'*iframe* e il male assoluto, ma in realtà usando l'API *postMessage* il fatto che si tratti di un *iframe*, di un altro Tab o di un'altra finestra del browser è indifferente per ottenere lo stesso risultato: la comunicazione tra la pagina legacy e la nostra applicazione Angular.

Andiamo a vedere il codice.

L'API postMessage

L'API *postMessage* permette la comunicazione tra due tab del browser, inviando messaggi in formato stringa e restando in ascolto di messaggi provenienti da altri tab. A livello di sicurezza, possiamo implementare dei controlli per evitare di ricevere messaggi da tab non sicuri.

postMessage

Quando apriamo un nuovo tab con l'istruzione `var w = window.open(url, '_blank')`, oppure quando otteniamo la reference di un iframe, l'istanza dell'oggetto di tipo *window* ha un metodo, *postMessage* appunto, che ci permette di inviare messaggi in formato stringa a quella finestra:

```
lang:javascript
w.postMessage(message, domain)
```

Possiamo invece restare in ascolto di messaggi da altre finestre con l'evento *message*:

```
lang:javascript
window.addEventListener('message', function (event) {
```

```
 var domain = "http://" + window.location.host
 if (event.origin !== domain) return;
 var message = event.data;
 // fai qualcosa col messaggio
}, false);
```

Combinando questa API con l'uso di *JSON.parse* e *JSON.stringify* possiamo comunicare da una pagina *legacy* verso un'applicazione *Angular*.

Codici per la comunicazione tra una pagina legacy e un'app Angular

Il codice nella pagina *legacy* è il seguente:

```
lang:javascript
var data = {name: "Salvatore"};

function startSending() {
 openPopup(window.location.href.substr(0,
window.location.href.lastIndexOf("/")) + "/index.html", data);
}


function openPopup(url, payload) {

 var domain = "http://" + window.location.host
 var nuovoSistemaPopup = window.open(url, '_blank');
 setTimeout(function () {
   var message = JSON.stringify(payload)
   nuovoSistemaPopup.postMessage(message, domain);
 }, 2000);
}


window.addEventListener('message', function (event) {
 var domain = "http://" + window.location.host
 if (event.origin !== domain) return;
 var receivedPayload = JSON.parse(event.data);
 console.log("nuovo messaggio modificato: ", receivedPayload);
 document.getElementById("idInput").value = event.data;
}, false);
```

Nell'app *Angular* dobbiamo anche inserire un codice analogo per ricevere messaggi e inviare messaggi da un qualsiasi componente.

La soluzione più semplice è aggiungere una snippet di codice subito dopo il *bootstrap* nel file *main.ts*:

```typescript
platformBrowserDynamic().bootstrapModule(AppModule);

window.addEventListener("message", receiveMessage, false);

function receiveMessage(event)
{
 console.log(event)
 console.log(event.data)
 if (event.origin !== "http://localhost:4200" )
   return;

 (window as any).store = JSON.parse(event.data);

 if(!(window as any).store.name)
   return;


 (window as any).sender = {
   data: JSON.parse(event.data),
   send: (data) => {
     event.source.postMessage(JSON.stringify(data), event.origin)
   }
 };
}
```

Nel codice sopra salviamo il messaggio nel campo *store* di *window* e inviamo un messaggio usando il metodo *send* dell'oggetto *sender*, creato da noi e salvato ancora nell'oggetto *window*. Possiamo usare quindi *store* e *sender* in qualunque componente. Possiamo volendo anche sofisticare la tecnica introducendo un servizio *CrossTabClientService*, anzi per un'applicazione reale suggerisco fortemente di incapsulare il comportamento sopra proprio in un servizio.

I codici sono disponibili su github:

https://github.com/devexp-io/codici-articoli/tree/master/comunicazione-cross-tabs

Conclusioni

Anche se abbiamo visto la tecnica *postMessage* applicata ad *Angular*, in realtà potremmo usare la stessa tecnica per integrare qualsiasi tecnologia in un sistema datato. Tale tecnica è inoltre utile se abbiamo necessità di comunicare tra diverse finestre del browser.
Per quanto riguarda i web components, invece, il futuro si sta muovendo in questa direzione e già esistono componenti di terze parti, ancora una volta non necessariamente sviluppati in Angular, ma che possono essere usati nel nostro sistema. E se te lo stai chiedendo, creando un nuovo progetto Angular

possiamo riutilizzare un web component direttamente nel nuovo sistema come se fosse un tag HTML nativo. In Angular, rispetto ad altri framework, la compatibilità con i web components è totale, mentre React e Vue hanno ancora qualche incompatibilità.

Buon lavoro e per chiarimenti e dubbi, non esitare a contattarmi.