WebComponents i Polymer - niespełniony sen?

Spis treści

Nie tak dawno zachwycałem się potęgą Web Components (http://webroad.pl/javascript/3505-web-components), wieszcząc im świetlaną przyszłość w webmasterskim świecie. Po kilku miesiącach używania tej technologii przyszedł czas na pierwsze refleksje i ostudzenie zbyt wcześnie zapalonego entuzjazmu. Trzeba posypać głowę popiołem i powiedzieć wprost: "To nie tak miało być".

Założenia a rzeczywistość

Nie ukrywam, że w Web Components od samego początku upatrywałem nie tyle całkowitej zmiany paradygmatu tworzenia zaawansowanych aplikacji internetowych, co po prostu naturalnej i wymaganej obecnym stanem Sieci ewolucji. Ewolucji na polu **tworzenia responsywnych (http://www.reactivemanifesto.org/) interfejsów użytkownika**. Tyle. Tylko tyle i aż tyle.

Nigdy nie myślałem, żeby do znacznika móc włożyć logikę aplikacji, która nijak się ma do interfejsu… Okazało się jednak, że jestem w przytłaczającej mniejszości. Większość komponentów bowiem, które możemy znaleźć w Sieci, dotyczy rzeczy, które niegdyś były domeną potężnych frameworków MVW. Deklaratywny router (http://component.kitchen/components/app-router), który wczytuje strony, będące… komponentami? Może i początkowo ta idea wydaje się interesująca, ale bardzo łatwo się przekonać jak bardzo strzelamy sobie w kolano, przenosząc lwią część logiki do HTML. To przerost formy nad treścią w swoim najdoskonalszym wydaniu - zamiast zrobić normalne linki i normalnie wczytywać strony z wykorzystaniem History API, tworzymy rozwiązanie problemu, który… sami na szybko stwarzamy. Tak, Web Components są fajne, ale to nie znaczy, że mamy wszystko na nie przepisać, tworząc rozwiązania niezwykle trudne do rozwijania i utrzymywania. Obawiam się jednak, że takie wykorzystanie Web Components będzie się stawać coraz bardziej popularne (wystarczy tutaj powiedzieć, że przecież Angular 2.0 będzie intensywnie korzystał z Polymera). Czy na pewno chcemy w pełni deklaratywnej Sieci? A jeśli tak, to dlaczego kiedyś protestowaliśmy przeciwko XHTML 2.0?

Polymer - wrzód na tyłku

Nikt o zdrowych zmysłach nie tworzy aplikacji obecnie w "czystych" Web Components. Byłaby to sadomasochistyczna sztuka dla sztuki, ponieważ żadna przeglądarka (nawet bleeding edge Chrome) nie wspiera w pełni wszystkich specyfikacji wchodzących w skład Web Components. Dlatego też de facto wszyscy muszą używać jakichś polyfillów (profillów?). Nie będę ukrywał, że najpopularniejszym jest projekt Polymer (https://www.polymer-project.org/), z którego korzystają także… inne polyfiille (takie jak Mozillowe X-Tags (http://www.x-tags.org/) czy Bosonic (http://bosonic.github.io/)). Mówiąc zatem krótko: na chwilę obecną tworzenie przy użyciu otwartego standardu, jakim są Web Components, sprowadza się do zamknięcia się w bibliotece serwowanej przez Google. Otwarta Sieć, czyż nie?

Stosunek Google do ich projektów wszyscy znamy - w każdej chwili mogą je uwalić. Co więcej, Polymer jest na tyle specyficznym projektem, że - będąc "na rynku" już 2 lata - wciąż jest uznawany za skrajnie eksperymentalny. Co to oznacza? Że średnio co tydzień można obudzić się z ręką w nocniku, po nieuważnym wklupaniu `bower update` w konsoli. Tak, Polymer zmienia się na pniu. Ostatnio przeżyłem szok, gdy okazało

się, że jego stare, dobre `platform.js` odeszło na emeryturę, zostając zastąpione bardziej modułowym `webcomponents.js`. Główna różnica? Zamiast dużego pliku, Polymer zasysa teraz **kilkadziesiąt** małych. Po prostu wydajność maksymalna out of box. Ale oczywiście inżynierowie Google problem widzą, dlatego przygotowali dla niego rozwiązanie (czy już wspominałem, że w świecie Web Components lubi się rozwiązywać problemy, które samemu się wcześniej stworzyło?) - Vulcanizer (https://www.polymer-project.org/articles/concatenating-web-components.html). Oczywiście to rozwiązanie tymczasowe, bo tuż za rogiem czeka już HTTP 2.0 i czegoś takiego nie trzeba będzie używać… Tak samo, jak od dwóch lat nie powinniśmy używać dziwnego owijacza w formie Polymera.

Częste zmiany to nie tylko sajgon w plikach, ale także sajgon przy testowaniu swoich komponentów - Polymer jest podzielony na tak szaloną ilość plików, że prawie zawsze występują jakieś nieporozumienia z cache. Nieporozumienia na tyle trudne, że aż byłem zmuszony wyłączyć cache dla komponentów (https://github.com/Comandeer/dGUI/blob/18f8f1c08471bdc8c6b3a6f490203a1cc242bf89/components/.htaccess) - jak na razie nie wróży to dobrze Polymerowi, bo zasysanie go przy każdym żądaniu od nowa to czysta kpina. Dodajmy do tego konieczność importowania także samych komponentów dGUI (http://dgui.comandeer.pl) i dostajemy co najmniej 3 sekundy do czasu pokazania czegokolwiek na stronie (bo Polymer łaskawie ukrywa wszystko, aż się komponenty nie doczytają…). **O 2.5 sekundy za długo.**

I wreszcie - sama konwencja Polymera. Każdy komponent (https://github.com/Comandeer/dGUI/blob/18f8f1c08471bdc8c6b3a6f490203a1cc242bf89/components/dgui-keyboard/dgui-keyboard.html) to style i JS wrzucone w znacznik `polymer-element`. Serio? Zważając na to jak specyficzne są projektowane zastosowania Web Components, uważam to co najmniej za dziwne (CSP, anyone?). Oczywiście Vulcanizer jest w stanie się tym zająć (czy już wspominałem o tworzeniu niepotrzebnych problemów?) - pytanie brzmi: czemu to nie jest domyślnym ustawieniem? Nagle z HTML-a zrobiono worek na wszystko. Witamy w XUL-u?

Do tego dochodzi fakt, że całe środowisko Polymera opiera się na Bowerze. Zatem, żeby użyć Polymera muszę zainstalować node.js, zainstalować bowera przez npm i dopiero wówczas mogę zainstalować samego Polymera. Na tej samej zasadzie działają wszelkie komponenty w tym wielkim ekosystemie. Czy to plus, czy minus - to zależy. Czasami bardzo przydaje się ustandaryzowany sposób rozprowadzania zależności, innym razem jest to kula u nogi.

I na sam koniec, komunikat z dev tools Firefoksa:

> mutating the [[Prototype]] of an object will cause your code to run very slowly; instead create the object with the correct initial [[Prototype]] value using Object.create
> *— Konsola JS*

Nie oszukujmy się - żeby Web Components działały, wraz z emulacją Shadow DOM, gdzie np. trzeba zatrzymać leakage `[id]` na zewnątrz, Polymer musi nadpisać **kilkadziesiąt prototypów DOM-owych**. Takie zabawy nigdy nie były wydajne, a co więcej są narażone na wszelkie możliwe błędy (nie bez przyczyny istnieje złota zasada, że prototypów hosta się nie tyka).

Oczywiście reszta punktów jest pisana z perspektywy właśnie Polymera - jedynej słusznej drogi ku Web Components.

Szablony

Początkowo idea szablonów w DOM, tworzonych przy pomocy znacznika `template` brzmi sensownie. Ot, mamy wydzielone na zewnątrz drzewko DOM, na którym możemy sobie do woli operować i dopóki nie wsadzimy go na stronę, nic się nie dzieje - zdarzenia się w nim nie odpalają, obrazki nie wczytują… Do tego dostajemy do pracy z nim potężne narzędzia związane z trawersacją drzewka DOM. Precz z płaskimi stringami, bawimy się żywą strukturą!

Problem polega na tym, że takie wykorzystanie szablonów jest skrajnie prymitywne i starcza dla podstawowych zastosowań. Dlatego w Polymerze wymyślili sobie tzw. "template bindings", które miały być zgłoszone do procesu standaryzacji w W3C, ale wygląda na to - całe szczęście - że tam nie dotarły. Na czym pomysł polega? Na tym, żeby móc używać w `template` uproszczonej składni wąsów (http://mustache.github.io/) + logiki ukrytej skrzętnie w atrybutach elementów. Przykład prosto z dokumentacji:

```
<template repeat="{{ foo, i in foos }}">
    <template repeat="{{ value, j in foo }}">
        {{ i }}:{{ j }}. {{ value }}
    </template>
</template>
```

Oczywiście to dodatkowo dostaje 2-way data binding. Brzmi fajnie, ale w praktyce to jest piekło na ziemi. Widać tu 3 poważne problemy:

- z systemu DOM-owego niepostrzeżenie przeszliśmy z powrotem na parsowanie stringów. Tym samym nie ma **najmniejszych** powodów, żeby dalej korzystać z tagu `template` - odebrana nam zostaje jego główna zaleta: DOM. To pociąga za sobą bardzo poważne konsekwencje, związane z debugowaniem takiego kodu (nic niemówiące błędy o funkcjach anonimowych w funkcjach anonimowych, które jeszcze przechodzą przez `eval`; bardzo podobny problem dotyka wszystkie JS-owe parsery HTML-owych stringów, zatem Angulara, jak i JSX z Reacta). W tym wypadku lepiej skorzystać ze standardowych wąsów.
- przenieśliśmy logikę na poziom HTML-a. Stare, przerabiane i za każdym razem ten pomysł upadał (patrz: Web Forms 2.0 i właśnie `[repeat]` pól). HTML nie służy do takich rzeczy - jest językiem **opisu strony**, nie językiem deklaratywnych szablonów. Od tego jest inny język (XSLT - i znów: czemu odrzuciliśmy XHTML 2.0, a teraz przepisujemy XML na HTML?)
- 2-way data binding wprowadza więcej problemów niż je rozwiązuje. Bindowanie model → widok jest naturalne i de facto tak działają od zawsze wszystkie MVW. Bindowanie model ↔ widok już takie nie jest. Czy model musi odpowiadać na zmiany, które zachodzą w widoku? To zależy (http://stackoverflow.com/questions/19481/is-data-binding-a-bad-idea). Niemniej wiązanie modelu bezpośrednio z DOM brzmi… podejrzanie.

Polymer psuje to, co było dobre w `template`, sprowadzając go do poziomu rozwiązań, które istniały przed nim. A sens `template` leży w jego DOM-owej naturze, nie w logice wepchniętej w ledwo trzymające się atrybuty.

ShadowDOM

ShadowDOM to HTML-owy sposób na enkapsulację. Tym samym tworzymy sobie czarne dziury - konkretny custom element ukrywa przed nami swoją implementację, udostępniając nam ładne, eventowe API (https://github.com/Comandeer/dGUI/blob/18f8f1c08471bdc8c6b3a6f490203a1cc242bf89/components/dgui-colorpicker/demo.html#L17) i nic więcej. Fajna sprawa, jeśli potrzebujemy konkretnego elementu interfejsu, przy którym całkowicie nie interesuje nas jego wewnętrzny sposób działania, a tylko i wyłącznie to, co w

zamian dostajemy. Colorpicker jest doskonałym tego przykładem (Po co mi, jako autorowi edytora graficznego, potrzebna jest wiedza, że to kółko kolorów generowane jest na `canvas`, przechwytuje zdarzenia myszki itd? Ja po prostu potrzebuję narzędzia do łatwego wybierania kolorów!).

Oczywiście nie może być za dobrze. Problemy pojawiają się bardzo szybko, aż za szybko… Podstawowym jest sposób obsługi zdarzeń na elemencie, będącym shadow rootem. Można z góry zapomnieć o takich wynalazkach, jak event delegation: zdarzenie przypięte do elementu z ShadowDOM zarówno w `this`, jak i w `event.target` będzie wskazywać na ten element - granicy cienia nie da się przełamać. Bardzo szybko przekonałem się o tym i to dość boleśnie, próbując naskrobać swój 1. Web Component, jakim stała się klawiatura ekranowa (http://dgui.comandeer.pl/components/dgui-keyboard/demo.html): odczytanie który dokładnie klawisz został wciśnięty nagle okazało się zajęciem dosyć karkołomnym. Cień udostępnia co prawda metody do pobierania jego dzieci i można bezpośrednio do nich przypinać zdarzenia, jednak ma to dwie wady: musimy pobrać wszystkie potrzebne nam dzieci i musimy do każdego przypiąć konkretne zdarzenie (łatwo policzyć, że dla kilku instancji takiej klawiatury na stronie będziemy mieli ponad 100 zdarzeń dla samych klawiszy!). Niezbyt wydajne, ale w ostateczności da się przeboleć.

Polymer nie byłby jednak sobą, gdyby nie zaoferował szybkiego, wygodnego i **całkowicie nieprzemyślanego** rozwiązania. Mowa o atrybutach `[on-[event]]` dla każdego elementu z ShadowDOM, które musi mieć zdarzenie. Mam wrażenie, że gdzieś to już widziałem (https://pornel.net/onclick). Wracamy do mieszania warstwy zachowania z… no właśnie - czym jest HTML w Web Components? Bo im bardziej wgłębiam się w ten temat, tym bardziej mam wrażenie, że HTML jest tutaj zawoalowanym odpowiednikiem JS. Niemniej te atrybuty jak dotąd są jedyną sensowną metodą dowiązywania zdarzeń do elementów w ShadowDOM (nie licząc iterowania po wszystkich możliwych węzłach shadow roota, co w Polymerze jest tak zgrabnie ukryte, że i tak lepiej dodać te atrybuty). Sądzę jednak, że `[on-click="{{clicker}}"]` wciąż wygląda lepiej niż `[ng-click="function()"]` - bo co do tego drugiego nie mamy żadnych wątpliwości, że całość przechodzi przez `eval`.

Warto tu także wspomnieć o dostępnym w każdym elemencie `this.$`, przechowującym referencje do wszystkich elementów Shadow DOM, które mają nadane `[id]`. Miły akcent, jednak raczej bym oszalał dodając `[id]` do każdego elementu.

HTML Imports

HTML Imports są HTML-ową wersją modułów CJS - są synchroniczne z natury. Problem polega na tym, że naturalnie występują tylko w Chrome i tylko w nim są synchroniczne. Wszystkie inne przeglądarki dostają zatem rozwiązanie asynchroniczne. Co to oznacza? Problemy.

Postanowiłem w dGUI pewne wspólne helpery i inne tego typu dziwne rzeczy wydzielić do osobnego pliku JS i importować go jako właśnie HTML Import. Szczęśliwy, że działa w Chrome, scomittowałem zmiany. Oczywiście w lisku nie działało. Czemu? Z prostej przyczyny - wiedząc, że importy są synchroniczne, pozwoliłem sobie na założenie, że globalny obiekt `dGUI` (tak, globalny obiekt - jeśli ktoś mi pokaże sensowny przykład wykorzystania UMD/AMD z Web Components, bez setki niepotrzebnych udziwnień, wówczas chętnie to zmienię) istnieje. W lisku ewidentnie nie było to prawdą, bo całe importy fallbackują tam do żądań Ajaksem. Tym sposobem skrypt się wyglebił.

Zatem jeśli chcemy się pobawić w dynamiczne zasysanie skryptów przy pomocy HTML Imports, a od tych skryptów zależeć ma cały nasz skomplikowany system, to lepiej zawczasu przygotować się na dziwne zabawy z asynchronicznością (zdarzenia, wdrożenie mimo wszystko AMD itp. dziwne praktyki, ocierające się o voodoo).

Dostępność

Web Components to nie HTML. Tutaj nie ma żadnych wartości semantycznych - wszystko należy budować od podstaw. Co to oznacza? ARIA, naprawdę sporo ARIA. I to do oznaczenia rzeczy najbardziej podstawowych, od zera. Taka jest cena za innowację.

I co dalej?

Web Components, jako zbiór naprawdę nowych technologii, cierpi na bardzo poważne problemy wieku dziecięcego. Mimo wszystko uważam jednak, że - gdy w końcu sytuacja się ustabilizuje - stanie się sensownym sposobem na implementację zenkapsulowanego, wydajnego i prostego w użyciu interfejsu użytkownika. Mam także nadzieję, że inni webmasterzy również dojdą do tego samego wniosku i przestaną przepisywać na Web Components wszystko, łącznie z rzeczami, które nigdy nie powinny być deklaratywne. Na razie jednak Web Components pozostają ciekawostką - potężną, lecz tylko ciekawostką. Jedynym sensownym jej użyciem prawdopodobnie są ściśle kontrolowane środowiska (typu node-webkit (https://github.com/rogerwang/node-webkit)). I na tym jak na razie zakres stosowania Web Components się kończy.

Sama technologia nie jest zła - większość tutaj opisanych zarzutów de facto tyczy się Polymera a nie Web Components per se. Niemniej - nie ma obecnie jakiegokolwiek sensu pisać w Web Components nie używając Polymera. I koło się zamyka. W przyszłości pewnie powstaną lekkie i o wiele przyjaźniejsze wrappery na Web Components niż Polymer. No właśnie - wrappery. A do tego jeszcze bardzo długa droga, po wyboistych grzbietach polyfillów…