## Overview

This spike was to determine if we can use the new *Element* construct of Angular 6 to build up applications or *microApps* - using an element that is written in Angular.

As opposed to one monolithic codebase, using this strategy will allow for a configurable, vertical integration of subsystems (large, specific-focus epic features) into a core platform.

An element is compiled down to an HTML standard Web Component usable pretty much anywhere HTML is used. After building, the resulting distribution is straight HTML and Javascript that can be bundled into an npm (Node Package Manager) module.

## Executive Summary

Building Elements in Angular 6 works, and can be useful for building microApps. The elements are HTML Custom Element compliant and works with many browsers. Using this abstraction layer is a solid choice because it is a cross framework, web standard.   
Literally, use the element (Angular functionality) anywhere an HTML tag can be used.

The construct is straightforward to code, Angular knowledge helps. Elements, once defined, are registered into the browser’s element table by the resulting .js files produced in the \dist directory.

Developer already familiarity with Angular 1.5 have an easy transition to building elements, with contained subcomponents in ng6. An element is essentially wrapper which exposes an Angular component to the outside browser as a Web Cponent. Since components can be build out of other components, the element is just the highest level ‘shell’ component; a portable (outside of Angular) microApp.

Risks involved:

1. size of the Angular element distribution (dist directory) as the framework produces large files, a compiler is due out in ng7 to minify those files during the ng build.
2. There is a slight risk that ie needs to catch up with supporting elements natively (currently bridged with polyfill). Works in Edge, out of the box, IE 11 throwing error (needs to be solved)

Works in Chrome, Works in Firefox.

Using ng 6+ elements for modularization of a microApp UI works. Using elements as an outer shell for a microApp is a sound strategy that holds much promise in creating a flexible SaaS solution that is flexible, and configurable UI allowing Epiq to build up, the Facilitator application out of vertical functional layers (front end and back ends) using microApps and microService(s) architecture.

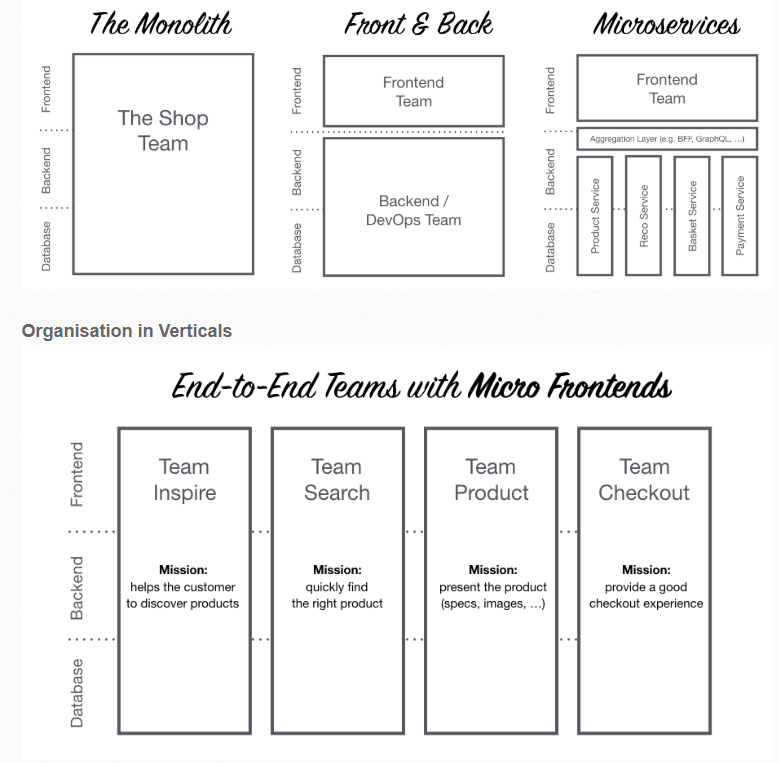
## What is a MicroApp ? Why do we care ?

MicroApps : Vertical Services – have a UI Frontend and likely, one or more Backend microServices they call.

This spike focuses on the Frontend. We care about modularization of the UI, so that separate functional areas can come, and go, and possibly be configured. It also forces the development teams that work on these areas to have stricter application domains (domain boundaries), lending to less adhesion in code.

The Facilitator platform can be (future-speak) a collection of vertical services, or *MicroApps*. The ability to build an instance of Facilitator by defining the specific set of services a client needs, maximizes the functionality, configurability and maintenance of the Facilitator offering.

How to configure is another spike (not sure if the ‘Control Panel’ ask fits this or not). If microApp front-end parts are available, it seems they could be configured together in a pick-n-choose fashion.

Vertical services (MicroApps) can be worked on by different teams.

In order to fully support a MicroApps we want to drive towards the bottom half of this diagram (End-to-End Teams with Micro Frontends)

Figure 1 from website: https://micro-frontends.org/

**Thesis:**  Angular 6 (ng6) is a framework that can support this concept of a microApp (UI, or front-end piece of functionality), prove the following:

1. The ng6 construct, called element, is suited to prescriptively build the pluggable Frontend layer
2. The building and maintenance of the microApp (both Frontend and Backend) within a Repository will simplify work on a particular vertical. Either both in one repo, or separate similarly named repo’s for microApp, microService.
3. W3 consortium Web standards are good for interconnect ability of the microApp piece. They can be dropped into an HTML code.
4. Team competencies mean a quicker refactor of existing codebase (from AngularJS 1.5) to Angular 6 as many concepts of the framework are similar, which simple to transition to syntactic differences only.
5. The Angular payload can be reduced, by the Ivy compiler so components don’t impact page load times, or by other methods (like minify).
6. Design considerations are understood for getting Guidelines established for architecting new microApps

**Assumptions**

Modern SPA (Single Page Apps) have become monolithic over time[[1]](#footnote-1). This makes them more difficult to maintain. We have experienced this with Facilitator Core. As apps grow larger, and more general they suffer from “over-generaliztion”(solve too many problems), adhesion, difficult to bring on new people (codebase too complex), lack of re-usability of compute functionality (re-invent the wheel), and a host of other issues.

**Findings**

## Elements

The ng6 Element is a new construct introduced by the Google team in Angular 6. The Google Angular team recognized they were doing 90% of the componentization already needed, to share UI features via ng components, but didn’t have a way to share or expose it outside of Angular framework; until elements came around.

An element is an abstraction layer, and a good one for encapsulating what this paper calls a MicroApp’s UI (or Frontend). The Element is designed it to be a compiled component, that can be a) self-bootstrapping, and b) dropped into any HTML page, another Angular “main App”, or even a React or Vue.js project. Other important points about ng6 elements:

* They are self Bootstrapping, and a dist dir contain the .js result of the build of the angular code.
* They actually host a plain-ole Angular 2+ Component inside an Custom Element (Web Component)
* They bridge the gap between the DOM APIs and Angular Components. Allowing dynamic insertion into the DOM.
* Encapsulated. Anyone code that can use HTML and include .JS can use the element (by its tag) without having the knowledge of how Angular works.

*Angular elements are ordinary Angular components packaged as custom elements, a web standard for defining new HTML elements in a framework-agnostic way*

### Using the ng CLI

The cli, from the command line has commands that can easily set up the scaffold to play with ng, and not create all files needed by the ng framework from scratch. Like starter projects, really.

1. From cmd line > npm install –g @angular/cli  
   401 Unauthorized: @angular/cli@latest
   1. And getting a 401, at this line:     http fetch GET 401 <https://epiqsystems.pkgs.visualstudio.com/_packaging/Epiq.npm/npm/registry/@angular%2fcli>
   2. Reached out to George Alexander, answers back quickly:
   3. Edited %UserProfile%\.npmrc (delete all lines, so an empty file). Fixed it for command in # 1)

The feed its trying to use is the old VSTS Feed.

There may be a .npmrc file stored with the source.

There may be a .npmrc file stored in “%UserProfile%\.npmrc”

Then I think it maybe the global is  “%programFiles%\nodejs\etc\.npmrc”

The source come from the npmrc file.

You should pull from the npm master feed in artifactory. If the npmrc file is in source control, then it should be removed.

I think there is a way to override the file using the command: <https://docs.npmjs.com/cli/install>

It looks limited to pulling from source out of a public git repo.

* 1. Installed the Angular CLI in global npm repo on my system. Location %appdata%/npm

1. Confirm install: > ng - -version yields Angular CLI 6.0.8, Node 8.9.4

## What is an Element ?

Definition of element *noun*1 Elements are sugar around Angular components, allowing packaging, to use Angular components outside of Angular.   
They conform to a web standard, called *custom elements.*  They are useful for defining new HTML elements in a *framework agnostic way* using Angular 6 (or later).

Because Angular components can contain other angular components, you can build up a subsystem’s UI, called a *microApp*. The microApp is a UI frontend usually calling one, or more, microService backend services.

Elements are a Web Platform, browser supported, construct which allows the extend HTML by defining essentially a new HTML tag.   
(Browsers: Chrome, Opera, Safari, IE 11, Edge ).   
Elements are also sometimes referred to as *Web Components* from the HTML side of things.

The browser keeps a registry of Web Components, or Angular Elements, in this spike.

## createCustomElement()

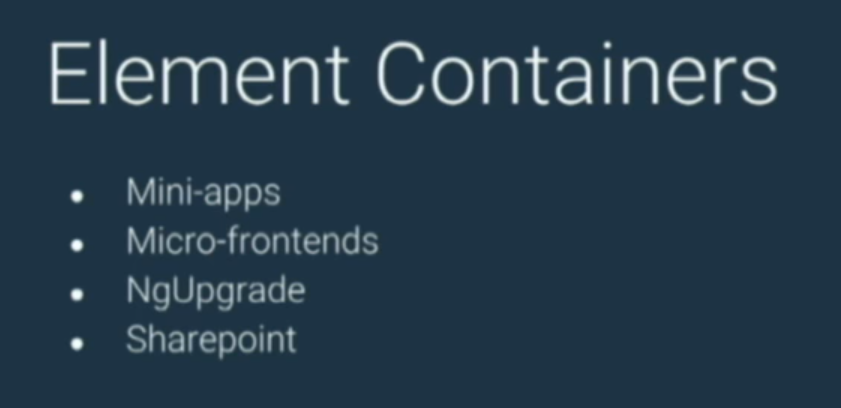
The @angular/elements package exports this API which provides a bridge from Angular’s component interface and change detection to the built-in DOM API.

I’ll construct in a separate directory (at same level as myapp) a microApp called quickSearchApp.  
It will look like the above (minimal components, no functionality tied to back end microService(s) ).  
The point of the spike is the inclusion of the angular element.   
  
This brings up a good point : any microApp should have its own solution. (its own root git).  
Later we will do a spike on packaging them up (likely in npm).

We’ll use a material design look-n-feel of the above.

## Self-training

Watched this video on Elements (:25 min) <https://www.youtube.com/watch?v=Z1gLFPLVJjY&t=4s> 17:34 in talks about our use case #2 microApp use.



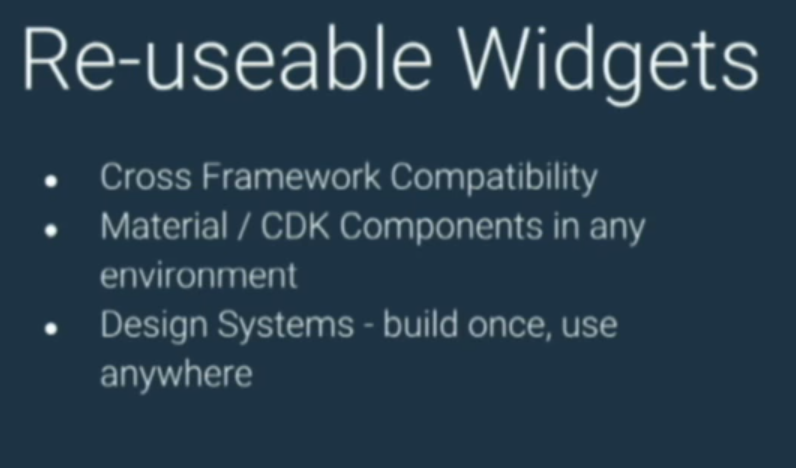


Figure 2 Reusable widgets - example DatePickers

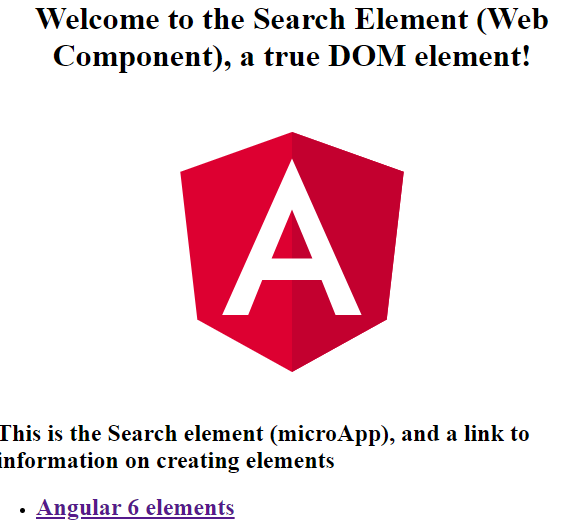
## Building code for a microApps from ng Elements

### Narrative

1. > ng generate component cuickSearchApp  
   This command can only be run inside of a CLI project.
2. Create new CLI app
   1. > ng new quickSearchApp
3. > Ng add @angular/elements
4. Add a component (that will be made into an element)  
   > ng g component quickSearchApp --inline-style --inline-template -v Native
5. I then in the quickSearchApp\src\app directory  
   purposefully rename the pattern quickSearchApp.component.ts, etc. and remove .component. so it becomes quickSearchApp.ts  
   Why? Even though an Element IS an ng component that is wrapped to be a Standard, ‘custom HTML Element’ (Web Component) we want to treat the microApp as the shell of possibly may other components. (those will have the .component. in the name when created). This allows us, along with a naming convention   
     
   Naming convention then: the outer SPA app that host the microApps is simply ‘app’ (as is the default in ng), and the microApp name follws the above example.

**Pattern:**

**{microApp-name}App** (example: quickSearchApp)

1. Adjust other files like the microApp.module.ts, ..\main.ts to reflect this subtle but important naming convention change.
2. Make mods to the html template to resemble something of a new top-level container for functionality  
   
3. See source files for correlations, but this is the gist:

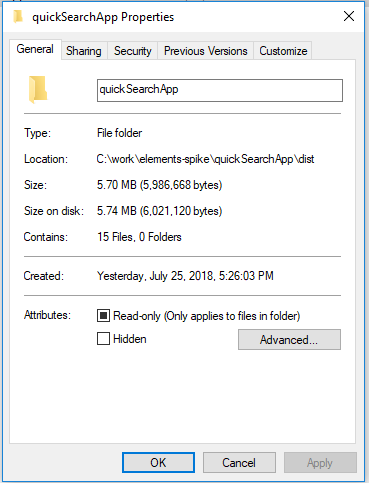


1. > ng build
2. Then in .\dist, >source-map-explorer vendor.js to see files in the
3. Mapped \dist to a virtual dir in IIS
4. Run index.html (after changing directory to find the .js files)

TypeError: Failed to construct 'HTMLElement': Please use the 'new' operator, this DOM object constructor cannot be called as a function.

at NgElementImpl.NgElement [as constructor] (elements.js:384)

1. FIX: Change in tsconfig.json : "target": "es2015",
2. Bundling – building source files into a bundle is done using broccoli and webpack
   1. Webpack converts commonjs modules to something that browsers understand
   2. Proccoli.js is responsible for running all build tasks even webpack transformations are part of broccoli tasks  
      Bundling is not part of this Spike.
3. Size of dist folder. Angular is known to be a fairly heavyweight framework,   
   the > ng build will create the dist folder, where we see this fairly large (two couple large files, 2 mb each: vendor.js and vendor.js map.  
   Assumption: this is where the ng7 (next release) ‘Ivy’ compiler will come in to help distill the javascript and map files down, de-dupe (if having several Custom Element/components) and produce a more streamlined (smaller) set of files.



## Design Considerations

* Deep initial evaluation is necessary to find logical breaks of verticals. I have no doubt that the Alpha team (Scott, Ron, Richard, Nina, Petar) can help in defining logical breaks of vertical functionality of the Facilitator platform.
* Messaging contract between. I think Pub/Sub messaging as discussed as a necessary cross-cutting concern (globally available service) allows for the messaging between verticals.
* Test strategy: I think more automation of interface testing, for individual vertical’s solutions is in order, the more automated the better.
* Self-contained: any features should be self-contained (not rely on other MicroApps)

## Alternatives

Some other packaging platforms or libraries

1. React – a separate spike is slated, to see how to do Custom Elements (HTML Web Components) are accomplished using the React framework and ecosystem.   
   During the spike we will reach out to the React team (downtown), and see about this for componentization at the ‘Element’ level, too.
2. Ionic (heavy weight, multi-faceted) framework. Larger company buy-in, likely to get package.
3. Svelte. <https://svelte.technology/guide> more singular focus of combining things,  
   seems like Component focused, and tries to eliminate framework bloat at compile time,  
   like a javascript reduction (redux) by compile philosophy. Very new, however.
4. Stencil. Compiler for Custom Web Components. Async rendering by React Fiber.  
   Created by Ionic team, to build faster, more capable components that worked across all major frameworks.
5. SkateJS – a wrapper around Web Components for cross platform use

What they all have in common mostly is the leveraging of Custom Elements, as the direction that HTML and Web standards have gone to combine elements and get away from JS Framework restrictiveness or allegiances.

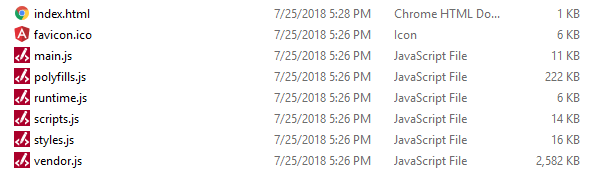
## Risks

1. Bloat: slightly ahead-of-the-curve. Elements are available in ng6, however ‘Elements + Ivy’ is not due until ng7. While getting Elements written in ng6 Angular syntax is currently possible there is potential issues with the size of individual package (npm) elements.
2. Lack of IE direct support for elements; Edge and IE will be there, not yet; however apparently a workaround is available using “polyfills”, so 95% of functionality is currently supportable back to IE9.  
   Note I did launch in Edge, no problem. IE 11 choked on something, I did not figure out the issue.
3. microApps colliding. This ng6 release is going to improve in ng7. There has been noted issues with multiple elements, with workarounds suggested. <https://github.com/angular/angular/issues/23732>

## Distribution Review

> ng build

Produced a dist with these javascript files:



Thesis: it appears the complete functionality for the execution of the element, will be self-contained in the six (6) files produced.

Therefore the only thing needed to run the element is,

* 1. Make sure the .js files load (prior to the reference to the tag of the element, in this case :  
     <quick-search-app></quick-search-app>
  2. Drop the tag in any html page or html template, in another front end JS framework or emit from any of the backend html processing engine technologies (PHP, ASP.NET, CMS platforms, etc)

This thesis has proven to be accurate.

### Where is the microApp’s code?

It appears to be nested/included in main.ts through the ng framework build.

## How should this new microApp be distributed (made available)?

Thesis: To be compliant with other ng type components and elements we would use an > npm type package. Then store distributions in Artifactory’s npm repository. We should spike : ng element-based microApp – build , dist, release cycles.

Issues to discover solutions, in this spike:

1. Release cycle engineering, what to use, Octobpus, Jenkins, Python, to get the build to occur (ng build and the distribution packages up, and put into Artifactory)
2. How to reference Artifactory from a host application, so the npm package is installed at the target host server location, in node\_modules
3. Should the microService (backend) be in the same or different source hierarchies? Should they be released together as a pair (for dependent containerized services?)

## Angular / Typescript Code Review

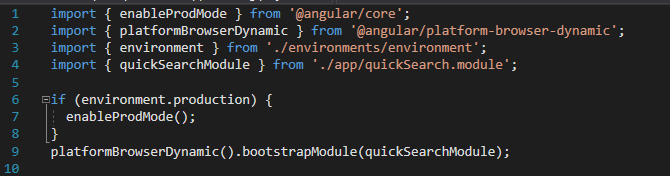


Figure 3 main.ts

Line 4: the special component (sans .component. in the name), is imported

Line 9: module bootstrap occurs  
  
Todo: can we dynamically bootstrap modules in run time? Also, can microApps configuration drive this so a single configuration file specifies the bootstrap for pick-and-choose type conglomeration. Finally, how is a dynamic inclusion managed in the host app. (real-estate negotiation, etc) based on the product (FAC Core) desired design?

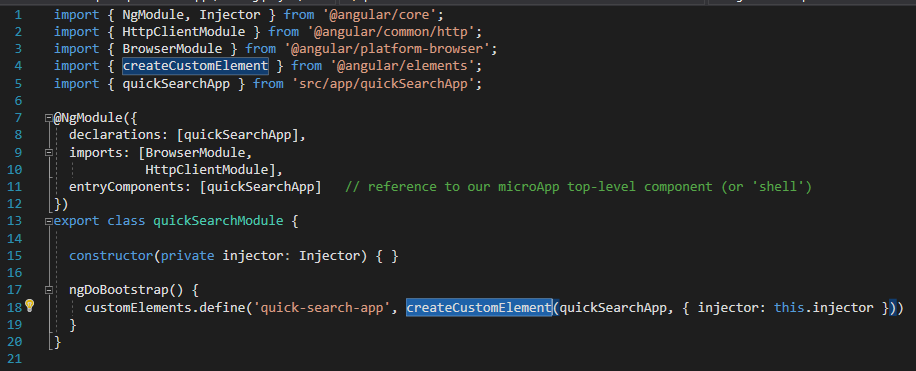


Figure 4 quickSearch.module.ts

Line 1: need Injector, from ng core

Line 4: need createCustomElement from installed npm module   
( see previous notes, for >ng add @angular/elements )

Line 5: the top-level component of the microApps

Line 8: declaration makes top-level component (microApps visible in Module)

Line 11*: entry component*; def: any component that Angular loads imperatively (bootstraps it)

Line 15: constructor injects ‘injector’

Line 17-18: definition of the custom element to the outside world  
<quick-search-app></quick-search-app> looks like any HTML tag, but will bootstrap our microApps

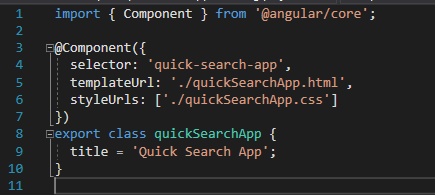


Figure 5 quickSearchApp.ts

Line 4: selector, I didn’t check to see if this is different than the element, what happens. I’d name it the same, regardless, though it may be superfluous (not used).

The rest of this is standard ng component design.

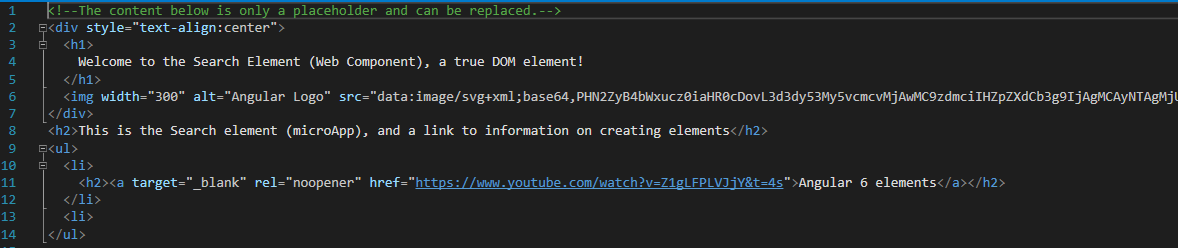


Figure 6 quickSearchApp.html

1. No comments to make here about the code. This is anything it needs to be, it’s the template for the visual UI of the top-level container (component) for the microApp

1. <https://itnext.io/building-micro-frontend-applications-with-angular-elements-34483da08bcb> [↑](#footnote-ref-1)