Technology Conversations

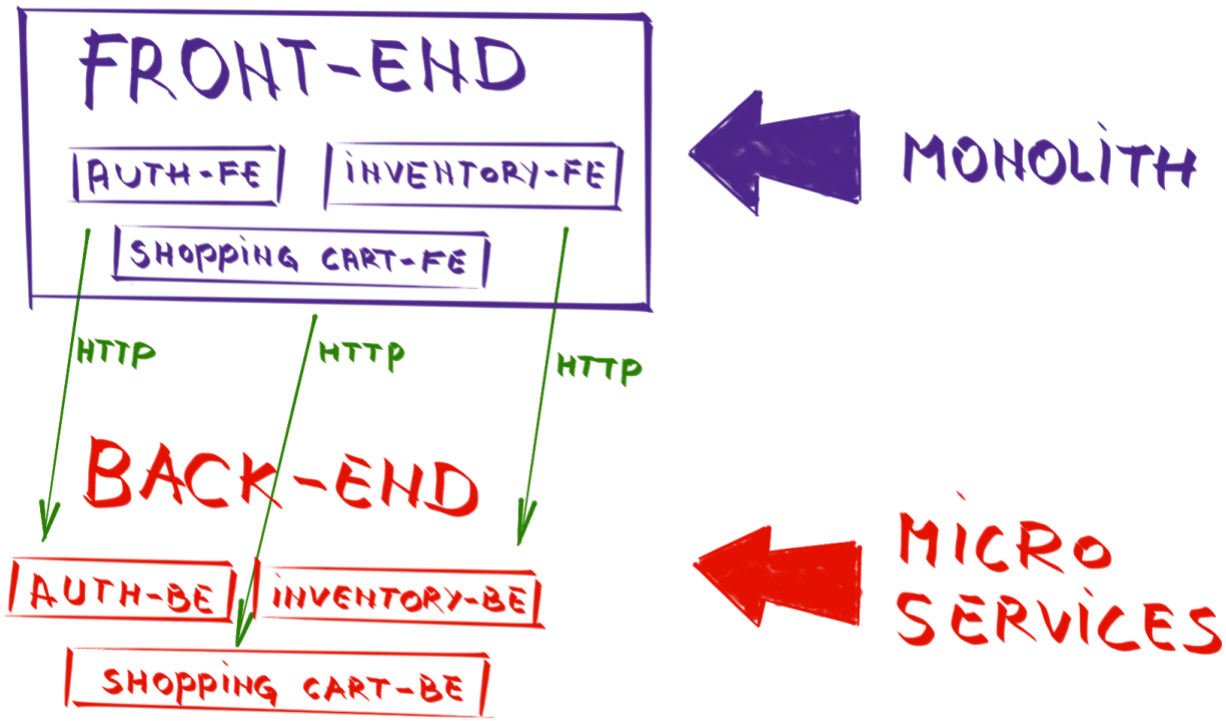
Including Front-End Web Components Into Microservices

Microservices and Front-End

Microservices are becoming more and more popular and many are choosing to transition away from monolithic architecture. However, this approach was mostly limited to back-end services. While it made a lot of sense to split them into smaller independent pieces that can be accessed only through their APIs, same did not apply to front-end. Why is that? I think that the answer lies in technologies we're using. The way we are developing front-end is not designed to be split into smaller pieces.

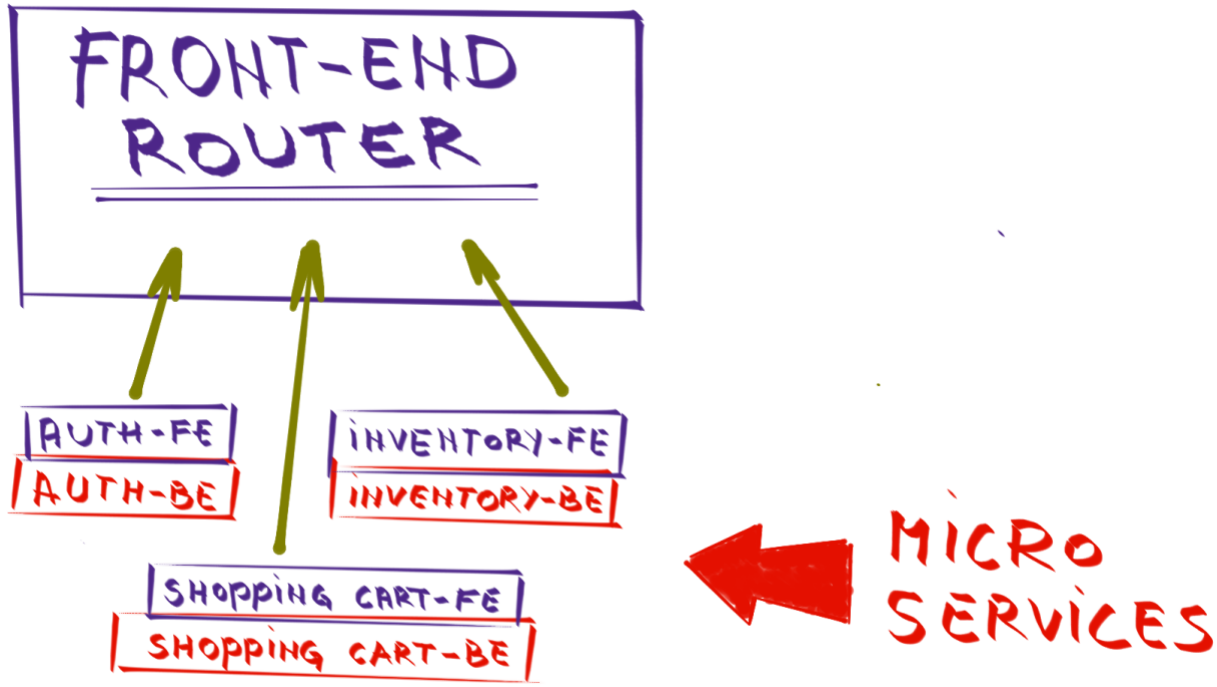
Server-side rendering is becoming history. While enterprise might not agree with that statement and continues pushing for server-side frameworks that "magically" transform, for example, Java objects to HTML and JavaScript, client frameworks will continue to increase in popularity slowly sending server-side page rendering into oblivion. That leaves us with client-side frameworks. Single-Page Applications are what we tend to use today. AngularJS, React, ExtJS, ember.js and others proved to be a next step in evolution of front-end development. However, Single-Page Applications or not, most of them are promoting monolithic approach to front-end architecture.

With back-end being split into microservices and front-end being monolithic, services we are building do not truly adhere to the idea that each should provide a full functionality. We are supposed to apply vertical decomposition and build small loosely coupled applications. However, in most cases we're missing visual aspect inside those services.



All front-end functionalities (authentication, inventory, shopping cart, etc) are part of a single application and communicate with back-end (most of the time through HTTP) that is split into microservices. This approach is a big advancement when compared with a single monolithic application. By keeping back-end services small, loosely coupled, designed for single purpose and easy to scale, some of the problems we had with monoliths become mitigated. While nothing is ideal and microservices have their own set of problems, finding production bugs, testing, understanding the code, changing framework or even language, isolation, responsibility and other things became easier to handle. The price we had to pay was deployment but that as well was greatly improved with containers (Docker and Rocket) and the concept of immutable servers.

If we see the benefits microservices are providing with back-end, wouldn't it be a step forward if we could apply those benefits to front-end as well and design microservices to be complete with not only back-end logic but also visual parts of our applications? Wouldn't it be beneficial if a developer or a team could fully develop a feature and let someone else just import it to the application? If we could do business in that way, front-end (SPA or not) would be reduced to a scaffold that is in charge only of routing and deciding which services to import.



I'm not trying to say that no one is developing microservices in such a way that both front-end and back-end are part of it. I know that there are projects that do just that. However, I was not convinced that benefits of splitting front-end into parts and packing them together with back-end outweights down-sides of such an approach. That is, until I discovered web components.

Web Components

Web components are a group of standards proposed as a W3C specification. They allow creation of reusable components that can be imported into Web applications. They are like widgets that can be imported into any Web page.

They are currently supported in browsers based on WebKit; Chrome, Opera and FireFox (with manual configuration change). As usual, Microsoft Internet Explorer is falling behind and does not have them implemented. In cases where browser does not support web components natively, compatibility is accomplished using JavaScript Polyfills.

web components consist of 4 main elements which can be used separately or all together:

- Custom Elements
- Shadow DOM
- HTML Imports
- HTML Templates

Custom Elements

With Custom Elements we can create our own custom HTML tags and elements. Each element can have its own scripts and CSS styles. The question that might arise is why do we need Custom Elements when the ability to create custom tags already exists? For a long time now we can create our own tags, apply CSS styles and add behaviors through scripts. If, for example, we would like to create a **list of books**, both with Custom Elements and custom tags we would end up with something like following.

```
<books-list></books-list>
```

What web components bring to the table are, among other things, lifecycle callbacks. They allow us to define behaviors specific to the component we're developing.

We can use the following lifecycle callbacks with Custom Elements:

- **createdCallback** defines behavior that occurs when the component is registered.
- **attachedCallback** defines behavior that occurs when the component is inserted into the DOM.
- **detachedCallback** defines behavior that occurs when the element is removed from the DOM.
- **attributeChangedCallback** defines behavior that occurs when an attribute of the element is added, changed, or removed

Shadow DOM

Shadow DOM allows us to encapsulate JavaScript, CSS and HTML inside a Web Component. When inside a component, those things are separated from the DOM of the main document. In a way, this separation is similar to the one we're using when building API services. Consumer of an API service does not know nor cares of its internals. The only thing that matters for a consumer are the API requests it can make. Such a service does not have access to the "outside world" except to make requests to APIs of other services. Similar features can be observed in web components. Their internal behavior cannot be accessed from outside (except when allowed by design) nor can they affect the DOM document they reside in. Main way of communication between web components is by firing events.

HTML Imports

HTML Imports are the packaging mechanism for web components. They are the way to tell DOM the location of a Web Component. In context of microservices, import can be a remote location of a service that contains the component we want to use.

```
1 <link rel="import" href="/services/books/books-list.html">
```

HTML Templates

The HTML template element can be used to hold client-side content that will not be rendered when a page is loaded. It can be, however, instantiated through JavaScript. It is a fragment of code that can be used in the document.

Microservices With Front-End

Web components provide a very elegant way to create pieces of front-end that can be imported into Web applications. Those pieces can be packaged into microservices together with back-end. That way, services we are building can be complete with both logic and visual representation packed together. If this approach is taken, front-end applications can be reduced to routing, making decisions which set of components to display and orchestration of events between different web components.

Now that we are equipped with (very) basic information about web components and desire to try a new approach to develop microservices, we can start building a microservice with both front-end and back-end included.

In the Developing Front-End Microservices With Polymer Web Components And Test-Driven Development series we'll explore one of the ways to put discussion from this article into practice. We'll use Polymer, Google library for creating web components, Docker, Docker Compose and few more tools and libraries. Development will be done using test-driven development (TDD) approach.

The first article in the series is Developing Front-End Microservices With Polymer Web Components And Test-Driven Development (Part 1/5): The First Component.

Advertisements

This entry was posted in Architecture, JavaScript and tagged Front-end, Microservices, Polyfills, Web Components on August 9, 2015 [https://technology-conversations.com/2015/08/09/including-front-end-web-components-into-microservices/] by Viktor Farcic.

19 thoughts on “Including Front-End Web Components Into Microservices”

AnonymousCoward
November 24, 2015 at 9:58 pm

I see the additional complexity of sharding your html files. What benefit(s) do you think best justify the additional complexity?

Viktor Farcic Post author
November 25, 2015 at 12:27 pm

I think that the question should be answered in a much broader context of the approach to the whole architecture and not be limited only to front-end. It is the question whether to use microservices or monolithic applications in general. My stand is very much in favor of microservices. You'll find many articles related to this question on this blog (for example https://technologyconversations.com/2015/11/03/microservices-when-the-stars-aligned/). If one chooses microservices as the overall approach to architecture, I think that Polymer provides a nice way to make them truly complete.

AnonymousCoward
December 2, 2015 at 12:04 am

That doesn't really answer the question. Why bundle a portion of the html with the API? What benefit does this added complexity provide?

Viktor Farcic Post author
December 2, 2015 at 12:35 am

The benefit is the same as with backend separation into services. Easier development, testing and deployment of separate parts of the application. If front-end is developed and packaged together, a single team can more effectively deliver the feature. Ofcourse, that (and microservices in general) makes sense if the system is big. If we're talking about something smaller where the whole system is developed by a single team (lets say up to 8 people), this separation would be an overhead in many cases.

Chizl

June 5, 2017 at 4:32 pm

I disagree.. This goes back to MFC in the 90's. The low end developers can write 1/10 of the code that is executed, to do the same thing, Including MFC was overwhelming that amount of memory, CPU, and disk space.. Same goes here by including a ton of libraries that do 10 times what you need, just so you can process one portion of it; makes no sense. I put this in the same category has a new technology, means new mark on the resume. People are not learning to develop code anymore. They rely on 3rd party component to do the work for them. Developers these days are looking for an easy way out and when things break, they can have someone else to blame. This is why we need gigabit networks to process what we did back in the 90's on a 10MB network. I feel sorry for the kids 30 years from now as it will need TB network connection in your home to hit Google's home page, which could be done with 10 lines of code, but if you right click, view source, you will see 1000's of lines and 2MB download. There is something to say about simplicity, but developers are building rockets to fly to the moon, just so they can drive something down the road.

Viktor Farcic Post author

June 5, 2017 at 5:02 pm

I would agree with you if JavaScript would be anything but a failed programming language. It does not work well by itself so the usage of libraries is almost a must. Actually, I think that microservices are fostering simplicity by defining a very small scope and fostering the usage of things that are required for that scope, not for all possible combinations one can imagine in a monolithic application. Polymer is a different story and it's pros and cons are equally valid no matter whether the application is monolithic or broken into smaller pieces.

Chizl

June 6, 2017 at 1:24 am

JavaSript was never intended to be the one all be all language. I work for a company that has us use WebFOCUS, which is a reporting tool. Next thing I know the business case is to use it to write full functioning web applications. Developers over use something, because it's what they are use to. That in it-self destroys the language.

Richard Clayton
May 28, 2016 at 3:46 am

In our architecture, we treat the UI as a microservice that gets deployed independently of other components in the system. The UI is served out of an NG-inx container and proxied by our API Gateway. This has worked out well for us. We find ourselves deploying the UI three times as much as other backend microservices. Plus, we can optimize calls by composing requests on the Gateway. This is one of the problems I see with associating UI components direct-ly to microservices. I like the concept of a microservice providing a UI fragment, but in implementation, it seems the UI is better architected as a monolith.

Ganesh
October 7, 2016 at 11:48 am

@RichardClayton
I completely agree to your last statement where you fragmented and ended saying UI being MONOLITHIC..
•UI should be treated as a client application. Do not couple UI to microservices or attempt to serve bits of your app from each miscroservices.

Robert Karczewski
October 24, 2016 at 8:17 am

I completely disagree. I would są that IT depend on size of UI. Small application that probably does not require microservices in back end should be monolithic. My last project was new environment for telco operator covering product, customer and fullfilment management. I cold not image how to design and develop it as monolithic. So it was implemented as microservices with UI.

Johan CHOUQUET
February 13, 2017 at 10:56 am

I would do exactly the same as you Robert. The thing is: how did you deal with CORS with Backend / FrontEnd, assuming that you were on a Cloud environnement ?

Karl Andresen
April 20, 2017 at 6:01 pm

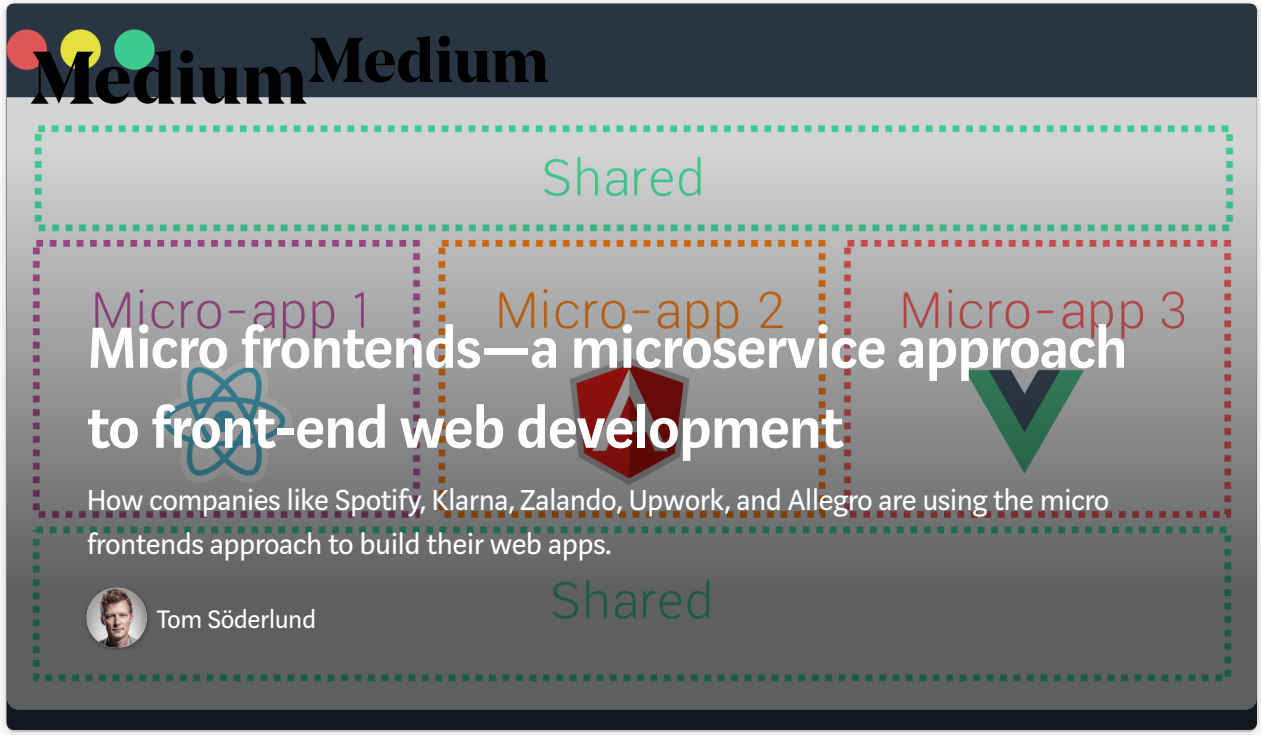
You probably already know this by now, but for anyone else still wondering... Use access-tokens instead of session based cookies in regards of authorization. (Hint: OAuth1/2 authorization delegation protocol...)

Pingback: Blog de Jose Cuéllar | Algunos comentarios de los libros que he leído los últimos meses

tomsoderlund
July 6, 2017 at 12:29 pm

Thanks for a great post, I linked it from here:

"Micro frontends — a microservice approach to front-end web development"



Gonzalo Gmlp
August 1, 2017 at 2:18 am

Good article

Pingback: Micro Frontends approach

Pingback: #Podcast – NTN 37 – Despues de #Angular y #React, es tiempo de #VueJS! Javascript everywhere! – El Bruno

Pingback: #Podcast – NTN 37 – After #Angular and #React, now it's time for #VueJS! Javascript everywhere! – El Bruno

sandeepjainblr
October 13, 2017 at 6:03 pm

Great post! I am linking it with
http://www.agilechamps.com/microservices-to-micro-frontends/