Developing Microservices with built-in User Interface using Web Components

Henry Vu
Masterseminar
University of Applied Science, Wuerzburg-Schweinfurt

In the scope of this work, it was proved that Microservices can be developed with built-in User Interface using the new W3C Web Components standards. A demo ecommerce shop was implemented in this particular architecture to prove this concept.

The initial point of this paper was the introduction of Web Components by the World Wide Web Consortium (W3C) in early 2014. These technologies enable the development of independent custom frontend HTML elements which can be composed together into one application. Studies have shown that Microservices - a modular and technology-independent approach to developing distributed web application, are still mostly limited to backend functionalities. The objective of this paper is to bring the aspects of Web Components and Microservices together to build fully autonomous Microservices with frontends.

This paper is a proof of concept that Microservices can be developed with built-in UIs using Web Components. An explorative approach was used for this purpose. At first this paper covered through a short introduction to Microservices and its current frontend problem, and then through Web Components and its related technologies. After that the main focus was on building a full-stack demo e-commerce shop application. This demo application was built in a Microservices architecture style with self-contained Web Components frontend elements. The attempted approach succeeded when it comes to fulfilling all functional requirements, but this also came with some workarounds to completely imitate the behavior of a monolithic frontend. These challenges are presented in the second last chapter of this paper.

Categories and Subject Descriptors: G [**General**]: Information Systems

Additional Key Words and Phrases: Microservices, Web Components

1. INTRODUCTION

The need for flexibility in todays software development has become more crucial than ever before. Business processes have to quickly adapted to the rapidly changing global economy. Traditional ways of software development can no longer live up to these requirements. The term service oriented architecture has arisen. [Josuttis 2007, p. 1]

Microservices are an approach to distributed systems that promote the use of finely grained services with their own lifecycle, which collaborate together. Because Microservices are primarily modeled around business domains, they avoid the problems of traditional tiered architectures. Microservices also integrate new technologies and techniques that have emerged over the last decade. [Newman 2015, p. 5]

However, this approach was mostly limited to backend services. While it made a lot of sense to split the application into smaller independent pieces that can be accessed only through their APIs, same did not apply to frontend [Farcic 2015]. This will result in a set of problems as the complexity of the application grows because a monolithic frontend will not be able to catch up with changes.

Author's address: Henry Vu, University of Applied Science Wuerzburg-Schweinfurt, Germany, www.fhws.de.

Polymer is an open-source project led by a team of frontend developers working at Google. It has the ambitious goal of making it easy to build faster and modern Progressive Web Apps, taking full advantages of Web Components a set of new W3C standards. These technologies enable developing independent custom frontend HTML elements which can be composed together into one application [Polymer 2016]. This could solve the monolithic frontend problem in Microservices.

The objective of this paper is to introduce a way of developing Microservices with built-in UIs using Googles Polymer. The main idea is to build Microservices with self-contained frontend elements and to have one frontend server composing these fragments together, instead of having one big monolithic frontend accessing Microservices through their APIs.

This work is divided in five chapters. The first one is this introduction. The second chapter covers up a quick introduction of Microservices and its current frontend problem. After that we take a closer look into Web Components and Polymer to explore new possibilities that might help to solve the given problem above. The build process of the demo ecommerce shop is presented in the fourth chapter. A conclusion follows at the end

Perquisites: This paper assumes basic familiarity with web technologies, such as REST services and HTML5.

2. MICROSERVICES AND MONOLITHIC FRONTEND

Microservices architecture is a well-tested and widely adapted pattern known from operating systems like Linux and Unix. The philosophy of this architecture style is *'Do one thing and do it well'*. Around 2008 this design principal is adopted to distributed software development projects. This is a more concrete and modern interpretation of Service Oriented Architectures (SOA) [Dragoni 2016]. There are certain characteristics around organization around business capability, automated deployment, intelligence in the endpoints and decentralized control of languages and data [Fowler 2016]. To put it in a nutshell: Microservices are small, autonomous services that work together [Newman 2015, p.16].

One of the important organizational aspect in Microservices is the practice of DevOps - short for Development and Operations. This describes the collaboration and communication of developers and operational professionals in small teams to deliver full autonomous software functionalities [Loukides 2012].

However, this approach was mostly limited to backend services. While it made a lot of sense to split one big application into smaller independent pieces that can be accessed only through their APIs, same did not apply to frontend [Farcic 2015]. The frontend is still regarded as one big monolith. This also means that the practice of DevOps is still limited to backend development.

With the frontend being a monolith that accesses different backend APIs means we still have the same problems as we had before switching to Microservices. Backend teams cannot deliver business value without frontend being updated since an API without data input from a user interface does not do much. As backend business functionalities grow, autonomous backend teams can quickly react to these changes while one single frontend team is not able to catch up. One proposal to compensate this problem could be making the frontend team bigger. This could improve the development speed of the frontend to a certain degree, but the main problem remains because the frontend still has to be deployed in one go. Therefore, it is impossible for teams to work independently. The figure 1 shows a simplification of the monolithic frontend architecture with Microservices in the backend.

Microservices architecture style provides a lot of benefits to backend development as it meets the most important software development paradigm *divide et impera* (divide and conquer). These benefits are according to [Newman 2015, pp. 17 – 27]: loosely coupled, technology heterogeneity, single responsibility, small deployment, easy to maintain and scale. Wouldnt it be a step forward if we could apply those benefits to frontend development as well? The Microservices design would

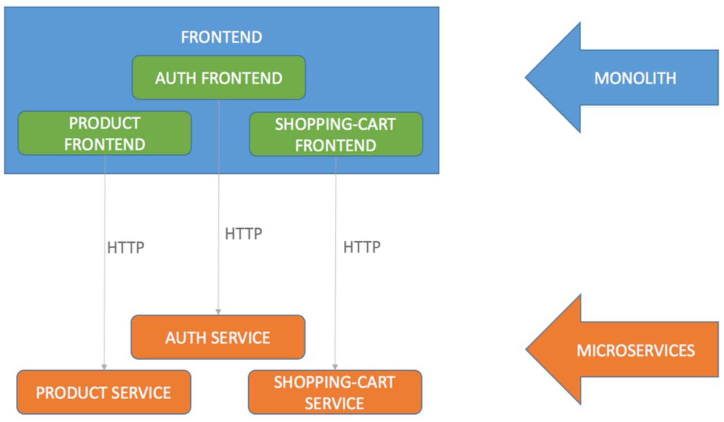


Fig. 1. Microservices with monolithic frontend, based on [Farcic 2015]

be complete. It would no longer only apply to backend business logics but also frontend visual representations of it. A fully autonomous team could develop a feature and let someone else just import it to the application (see figure 2):

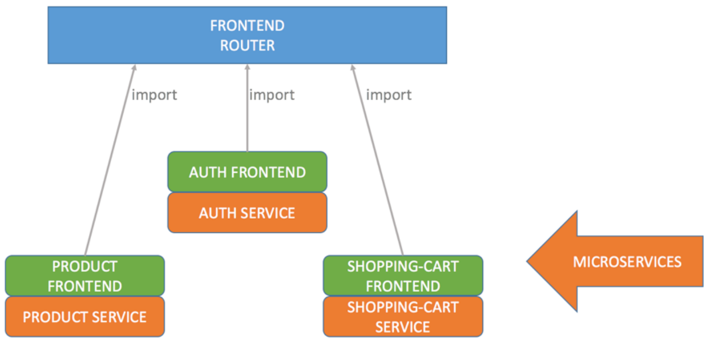


Fig. 2. Microservices with built-in frontends, based on [Farcic 2015]

The basic idea of this paper is to solve this concern by using Web Components to develop independent and self-contained frontend elements for each microservice.

3. INTRODUCTION TO WEB COMPONENTS AND POLYMER

Since there is no standard way to define a component, every framework has to invent its own, and JavaScript is a flexible enough language to allow that. Unfortunately, this results in fragmentation: components built using different frameworks do not interoperate with each other. Web Components are a set of specifications to solve this problem. [Savkin 2014]

It is a very unpleasant task to teach a website a new feature because there are many ways to achieve this, and each of these ways requires some custom tweaks. The bottom line is that it does not matter which way we choose, we always have to deal with some kind of external JavaScript-API [Kroener 2014].

Lets take a look at a simple component: the date picker component.

If we are to build this component in Backbone, the result might look as follows:

```
var datePicker = new DatePicker({el: el, format: 'yyyy-mm-dd'})
```

The same component written for Angular application is going to have a very different API:

```
<date-picker format='yyyy-mm-dd'/>
```

Through both the data pickers look and behave exactly the same, their implementations and APIs are very different. They do not extend a common interface. Therefore, they cannot interoperate with each other (see figure 3).

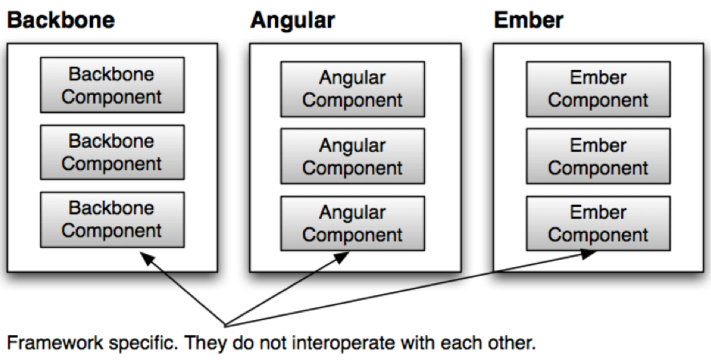


Fig. 3. Current major frontend frameworks, from [Savkin 2014]

This is where the new standards of Web Components come in handy. Just like HTML5, Web Components are not any concrete technologies, but more a set of specifications that describe new features and functionalities for the web. There are four main specs according to [Webcomponents 2016]:

(1) **Custom Elements**: This specification describes the method for enabling the author to define and use new types of DOM elements in a document
(2) **HTML Imports**: HTML Imports are a way to include and reuse HTML documents in other HTML documents
(3) **Templates**: This specification describes a method for declaring inert DOM subtrees in HTML and manipulating them in instantiate document fragments with identical contents
(4) **Shadow DOM**: This specification describes a method of establishing and maintaining functional boundaries between DOM trees and how these trees interact with each other within a document, thus enabling better functional encapsulation within the DOM

With Web Components we can easily include the date-picker example above like this:

```
<link rel="import" href="components/date-picker.html">

<date-picker></date-picker>
```

Or we can include any other element, for example a sign-in-form in the same manner:

```
<link rel="import" href="components/sign-in-form.html">

<sign-in-form></sign-in-form>
```

And it does not matter which framework these example elements use, as long as they are built on top of Web Components standards, they can be used in this conforming way. This following image (see figure 4) shows what could be possible if all existing frontend frameworks are built on Web Components standards.

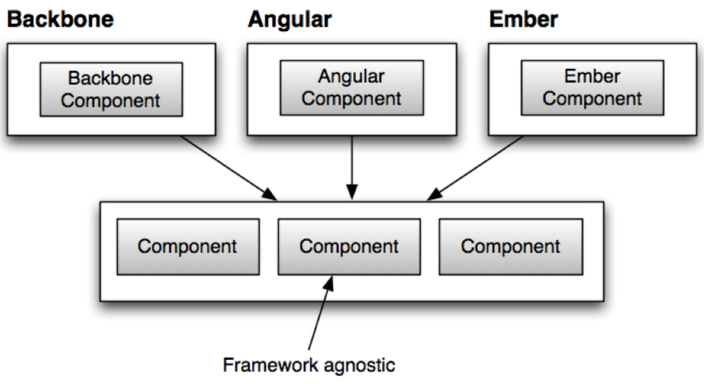


Fig. 4. How the future of the web could look like with Web Components, from [Savkin 2014]

There are a few early adapted libraries out there since Web Components is a newly released standards. Googles Polymer is currently the most advanced and popular one. It is a new type of library for the web, built on top of Web Components, and designed to leverage the evolving web platform on modern browsers [Polymer 2016].

So why might this be a perfect fit for the proposal of this paper? As mentioned in the Introduction and Microservices and monolithic frontend chapter: The Microservices architecture is still dealing with a monolithic frontend problem. And with Web Components we are able to create reusable custom frontend elements that are able to interoperate seamlessly with the browsers built-in elements.

4. BUILDING A DEMO E-COMMERCE SHOP

This chapter focuses on the building process of a demo ecommerce application. This demo application is the proof of concept for developing Microservices with built-in UIs. This proof of concept is verified when the proposed architecture can provide the same functional requirements as a monolithic frontend architecture. These functional requirements are displaying the application as one, no service API calls have to be made by the frontend router, applying CRUD functionalities to data models and establishing seamless communications between frontend components. Other nonfunctional requirements, such as development aspects or user experience are not taken into consideration.

The development proposal consists of two main services: product-service to handle all business functionalities around product domain, such as create, read, update and delete. And a shopping cart service to handle the functionalities of a shopping cart, such as save or remove product items in a cart and get the total price of a cart. Each service will be implemented with its own frontend using Polymer. A frontend router will only use HTML import to include and access the two Microservices and to display everything on the main application.

From the technology perspective of this architecture: Every single service is running on its own port to implicit its own boundary. The product and the shopping cart service are using their own database. This is conforming to the decentralized data management aspect of Microservices [Fowler 2016]. The shopping cart service database only hold the IDs of the products to guarantee data consistency, because the product data e.g. prices can always change.

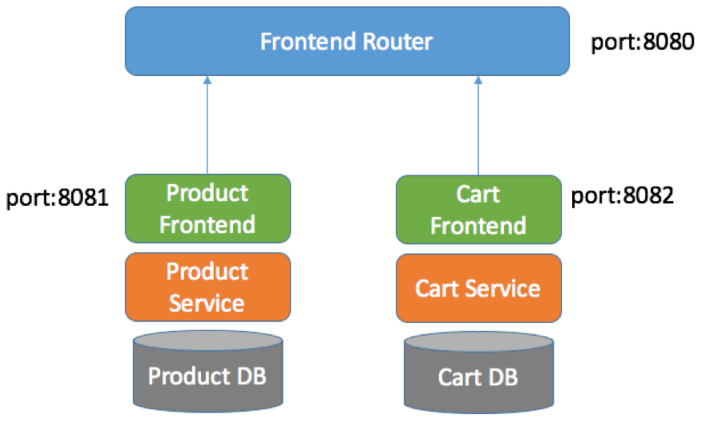


Fig. 5. Proposed architecture

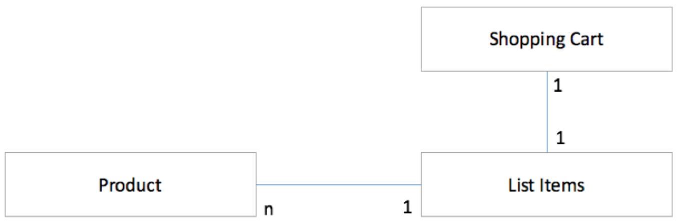


Fig. 6. Data model

To display the entire application, the frontend router merely has to import the frontend components via HTML imports and use the custom imported elements.

```
<!-- Import the product service with UI -->
<link rel="import" href="http://localhost:8081/components/product-list.html">

<!-- Import the shopping cart service with UI -->
<link rel="import" href="http://localhost:8082/components/shopping-cart.html">


<!-- Display the imported elements -->
<product-list></product-list >
<shopping-cart ></shopping-cart >
```

Since the CRUD operations on data are just standard ajax calls to the backend APIs, this task can be achieved with no further complications (see figure 7).

One of the greater challenge is to establish a communication between frontend elements since they are now no longer built by one single frontend team, but by many Microservices teams. For example, when the price of a product changes, a shopping cart that holds this product needs to notice and fetch the new price from the product service so the user can always see the updated price.

The old way of achieving this is to let the user manually refresh his shopping cart to get the latest prices or set a timer to periodically trigger a refresh which can lead to unnecessary requests if nothing changes. But since we live in a modern day of web development, these suggestions are

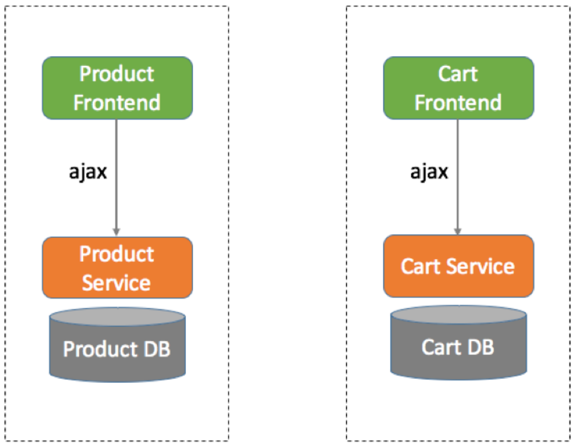


Fig. 7. CRUD Ajax

not an option. In a monolithic frontend architecture, the development team would know all of the elements and their functions. So it is easy to tell certain elements to refresh after updating some data. But in this approach we want UIs elements to be modular and encapsulated with their backend.

We used to have Microservices as backend and in order to access them, we need some kind of APIs descriptions because other developers have to know where to find the endpoints and how to use them. Now we have Microservices with a frontend layer on top. We could also provide a frontend API for anyone who would like to use our frontend element. And this is possible through Web Components, since the HTML template spec makes it possible to write HTML components in an Object Oriented way e.g. a Java Class with attributes and functions. So a possible API call for the shopping cart to refresh after the product data have been updated in the product service frontend code might look like this:

```
handleUpdateProductResponse: function() {
          document.querySelector("shopping-cart").refresh();
}
```

There are only two types of information that need to be exposed: the frontend element name *shopping-cart* and the public function *refresh()*. The name of the component is generally known, because it is the only way to import a HTML component. The refresh function could be a part of the API description for this component. Other than that the product service team does not know anything else about the shopping cart component and vice versa. The following figure illustrates the behavior of updating product information (see figure 8).

The steps are sequentially numbered:

(1) ajax request to update a product
(2) ajax response
(3) frontend API call for shopping-cart to refresh
(4) ajax request to get all updated information
(5) http request to product service to get updated information
(6) http response
(7) ajax response with updated information

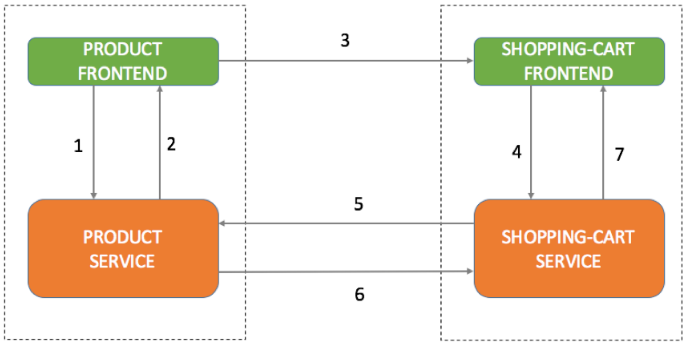
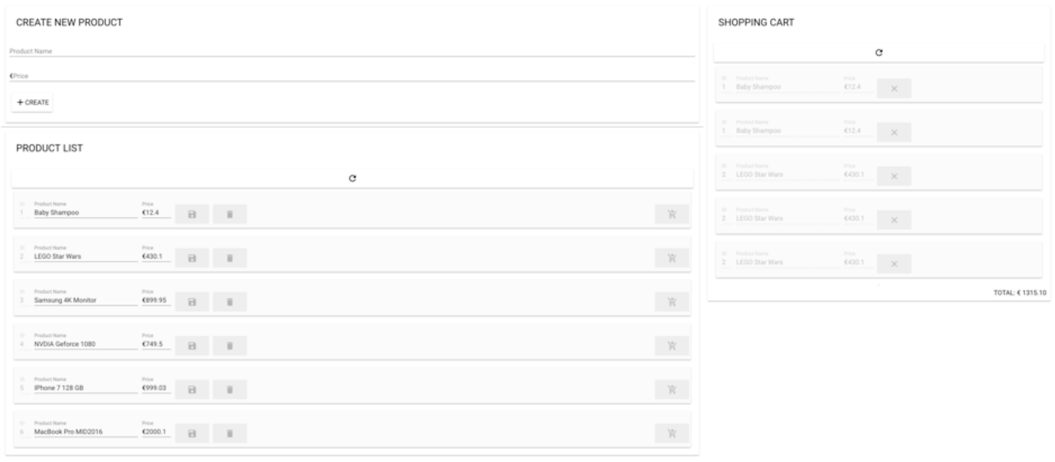


Fig. 8. Communication between services when updating product information

Figure 9 shows the result of the demo application. The left side of the web application presents the UI elements of the product service and the shopping cart is display on the right side. The entire code for this project can be found on www.github.com/henrythevu/



ID	Product Name	Price
1	Baby Shampoo	€12.4
2	LEGO Star Wars	€430.1
3	Samsung 4K Monitor	€899.95
4	NVDIA Geforce 1080	€749.5
5	IPhone 7 128 GB	€999.03
6	MacBook Pro MID2016	€2000.1

ID	Product Name	Price
1	Baby Shampoo	€12.4
1	Baby Shampoo	€12.4
2	LEGO Star Wars	€430.1
2	LEGO Star Wars	€430.1
2	LEGO Star Wars	€430.1

TOTAL: € 1315.10

Fig. 9. Demo Ecommerce shop

5. CONCLUSION

This paper has shown that it is possible to build fully autonomous Microservices with frontend by using Web Components. A sample ecommerce shop was implemented in this proposed architecture to demonstrate the proof of concept. Each Microservice is self-contained with UI. No service API calls are made from the frontend router. All functional requirements, such as displaying the application as one, implementing CRUD operations on data and establishing communication between frontend elements have been achieved. Although in order to accomplish the last mentioned requirement, the extra frontend layer needed some kind of service description, since remote DevOps teams cannot have knowledge of other frontend elements. There are many other things in this field that can be addressed to in the feature, such as service version control, standardized service descriptions for frontend components, styling options, model-driven frontend development and nonfunctional requirements.

Since Web Components are new standards of the W3C, it is a matter of time until all major browsers fully support these features. This is definitely a welcome complement to the Microservices design philosophy of modularity and platform independence.

Literatur

Dragoni, N. 2016. Microservices: yesterday, today, and tomorrow. [Online; last accessed 09-July-2016].

Farcic, V. 2015. Including front-end web components into microservices - microservices and front-end. [Online; last accessed 09-July-2016].

Fowler, M. 2016. Microservices - an in-depth description of the microservice style of architecture. applications designed as suites of independently deployable services, governed in a decentralized manner. [Online; last accessed 09-July-2016].

Josuttis, N. M. 2007. *SOA in Practice - The Art of Distributed System Design* 1. Aufl. Ed. Ö'Reilly Media, Inc.", Sebastopol.

Kroener, P. 2014. Web components erklrt – teil 1: Was sind web components? [Online; last accessed 09-July-2016].

Loukides, M. 2012. What is devops? what we mean by operations, and how it's changed over the years. [Online; last accessed 09-July-2016].

Newman, S. 2015. *Building Microservices* - 1. Aufl. Ed. O'Reilly Media, Incorporated, Sebastopol, California.

Polymer. 2016. About the polymer project. https://www.polymer-project.org/1.0/about. [Online; last accessed 09-July-2016].

Savkin, V. 2014. Why you should care about web components. [Online; last accessed 09-July-2016].

Webcomponents. 2016. Web components – a place to discuss and evolve web component best-practices. [Online; last accessed 09-July-2016].