Guidelines for Adopting Frontend Architectures and Patterns in Microservices-Based Systems

Holger Harms
Funke Digital GmbH, Germany
h.harms@funkedigital.de

Collin Rogowski
inovex GmbH, Germany
crogowski@inovex.de

Luigi Lo Iacono
Cologne University of Applied Sciences, Germany
luigi.lo_iacono@th-koeln.de

ABSTRACT

Microservice-based systems enable the independent development, deployment, and scalability for separate system components of enterprise applications. A significant aspect during development is the microservice integration in frontends of web, mobile, and desktop applications. One challenge here is the selection of an adequate frontend architecture as well as suitable patterns that satisfy the application requirements. This paper analyses available strategies for organizing and implementing microservices frontends. These approaches are then evaluated based on a quality model and various prototypes of the same application implemented using the distinct approaches. The results of this analysis are generalized to a guideline that supports the selection of a suitable architecture.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Software and its engineering → Organizing principles for web applications**; Interoperability; Software performance; Software reliability; Software fault tolerance; • **Applied computing →** *Enterprise architecture modeling*;

KEYWORDS

Frontend architecture, Design patterns, Microservices

ACM Reference format:
Holger Harms, Collin Rogowski, and Luigi Lo Iacono. 2017. Guidelines for Adopting Frontend Architectures and Patterns in Microservices-Based Systems. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 6 pages.
https://doi.org/10.1145/3106237.3117775

1 INTRODUCTION

Microservices enforce features that impose a constant productivity level in complex systems [10]. These characteristics and advantages include technological heterogeneity, resilience, independent scalability and deployment, targeted organizational orientation, combinability, and replaceability [16]. In recent years, several major companies including Netflix and Amazon [30] have adopted microservices as an architecture style of distributed systems for their enterprise and web applications to address the current and future challenges of cloud-based applications.

In general, microservices are an *architectural style*, which, much like SOA [7] or REST [9], is designed to develop applications that consist of small, independent entities and are made available through networks [22]. Each service unit adopts a certain functionality of the application. That is, microservices decouple individual application components, which usually comprise a complete application. They serve as an alternative for monolithic applications [27]. Microservices run as stand-alone processes and communicate within the networks using lightweight mechanisms [14]. These mechanisms include, for example, a REST-based interface, which is made available via HTTP. As a result, microservices can be independently developed, installed, and scaled [14].

The integration of microservices in the frontend or the user interface (UI) is an important aspect to be considered already during the software design phase [36]. In this regard two main questions need to be answered in the context of microservices-based system development. What methods are available for integrating mircoservices in the frontend of different application types to form a holistic system and what influence does the choice of frontend architecture have on the software quality of the system? This article aims to answer these questions and to provide decision support guidance for developers in terms of a general guideline.

The following aspects are outlined in the remaining sections of this paper. Section 2 reviews and discusses relevant work related to the subject of this paper. Section 3 explains pertinent architectures and patterns that can be used for the integration of microservices in the frontend. In Section 4 the methodological approach for the evaluation of frontend architectures and patterns as well as the basic implementation in the form of prototypes is described. The test setup and its implementation are also shown. Section 5 presents the results of the tests performed, and abstracts conclusions for a guide to support the appropriate selection of a frontend architecture in the given circumstances. Finally, Section 6 summarizes all relevant findings in a conclusion and provides a brief outlook.

2 RELATED WORK

To the best of our knowledge, this is the first work that systematically examines the advantages and disadvantages of specific frontend architectures for the integration of microservices by means of explorative, prototype-based studies. The selection of the frontend architectures taken into account is based on the results and

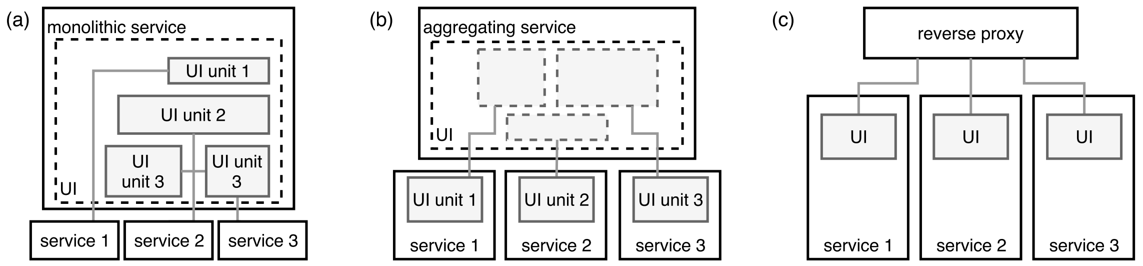


Figure 1: Integration of microservices via (a) UI monolith, (b) plugin approach, and (c) self-contained systems

conclusions of prior surveys and other researchers. Since the microservice architecture is a relatively young architectural style in the area of software engineering, the number of works, which is specifically concerned with the microservice integration in the frontend is relatively small. So far, scientific research focuses on other aspects, but to a small extent also takes the integration at the UI level into account. Versteden et al. describe a monolithic UI architecture, that uses a single page application (SPA) for the microservice integration [33]. Villamizar et al. also illustrate such a solution and extends the use case by an API gateway [34]. Marru et al. focus on a gateway architecture and describe how an access point to a microservice-based system can be constructed [15].

Further surveys focus on different frontend architecture styles for the microservice integration. The descriptions and designations of these styles are not always consistent. Nevertheless, frontend architectures that are increasingly appearing can be determined. These architectural styles are taken into account in this paper.

Newman [16] and Wolff [36] provide quite comprehensive reviews. Newman's reference book describes a *monolithic UI* approach using API composition and a *plugin approach* using UI fragment composition. Moreover the book characterizes the patterns *API gateway* and *backends for frontends* (BFF) [16]. Wolff describes these styles and patterns as well and supplements *self-contained systems* (SCS) as a further integration approach [36].

Other articles making taxonomic considerations are from Richardson and Smith [20], Zörner [38], Attermeyer [2] and Tilkov [31]. Richardson and Smith focus on monolithic architectures and the API gateway pattern. Tilkov also describes these approaches and supplements them with SCS. Attermeyer and Zörner describe the UI monolith, UI fragment composition, SCS and BFFs as integration approaches.

In addition, there are other articles that outline application examples in which microservices can be integrated by a specific solution. Steinacker [27] describes how SCS and UI fragment composition have been used in the system architecture of the e-commerce site *otto.de*. Farcic [8], Galek [11] and Wider [35] also focus on UI fragment composition in their respective application. Finally, Plotnicki [18] describes how BFFs have been applied at *SoundCloud*.

As a consequence, it can be concluded that there are basically three main architectural approaches available for integrating microservices at the frontend level: UI monolith, plugin approach, and SCS [38], [2]. Apart from these architectural styles, the patterns API gateway and BFF are also outlined in connection with the microservice integration. Still, according to the knowledge of the authors, there is no work available that compares the available approaches in terms on their impact on software quality metrics. This is required, though, in order to provide some guidance when to best choose which one of the available approaches for a certain development context is suitable. Due to these observations, the focus of this paper lays on the analysis of these styles and patterns, which are described in more detail in the following section.

3 AVAILABLE FRONTEND ARCHITECTURES AND PATTERNS

An example of monolithic frontends are rich client applications such as mobile or desktop apps [36]. The frontend is part of a coherent application which, e.g., communicates with subordinate microservices via HTTP and exchanges data in a certain format, like JSON or XML (see Figure 1 (a)). That is, API composition is applied and the microservices have an accessible interface [20]. A typical UI monolith in the context of web-based applications is implemented by a single page application (SPA), which integrates microservices on the client side [8].

Alternatively, it is possible that the microservices deliver entire segments of the UI, which are assembled via UI fragment composition by means of a higher-level application (see Figure 1 (b)). This approach can be used for web applications, but also for portal servers or desktop clients, e.g., in the style of *Spotify* [38]. In the context of web applications, the implementation can be applied on the server side via a frontend server. The server provides a framework for the UI and includes shared assets of the services, such as CSS files and JavaScript libraries [38]. The holistic UI is composed of the HTML fragments delivered directly by the microservices [36]. Technically this can be implemented via server-side includes (SSI) or edge-side includes (ESI) [27].

Self-contained systems are a further possibility to integrate microservices in the frontend of web-based applications (see Figure 1 (c)). Here, each service or rather SCS provides an autonomous web application and subdivides an application along defined technical boundaries [28]. Each professional domain resulting from the subdivision is developed as a stand-alone and interchangeable web application and is connected to other SCSs via hyperlinks [28]. For the UI the principles of the resource-oriented client architecture (ROCA) are used [2]. ROCA is a collection of simple recommendations for adequate frontends of web applications [21] and positions itself as an alternative to SPAs [36]. These recommendations should serve as a reference for implementations or as a comparison of existing applications [21]. At a technical level, SCSs can be integrated

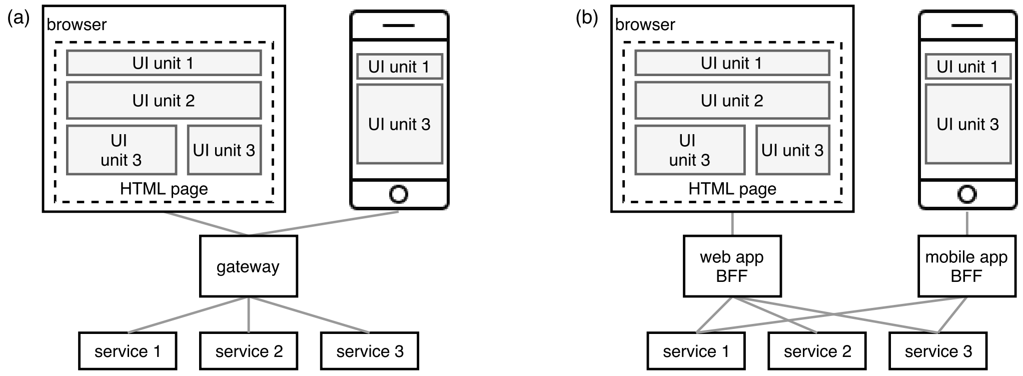


Figure 2: Integration of microservices via the patterns (a) API gateway and (b) backends for frontends

on the client side via SPAs or on the server side via applications that deliver HTML pages.

In the previously described approaches a direct communication between UI services and the microservices takes place. For UI monoliths it is possible to route this communication via a server that encapsulates the internal system architecture [17]. In this context the patterns API gateway and BFF emerged (see Figure 2).

An API gateway is a server that provides a single entry point to systems and specifies tailored APIs for various clients [20]. All requests from the clients are aggregated by the gateway and routed to the corresponding microservice. It is not the task of the gateway to integrate the microservices in the frontend, however. The main functions rather include request shaping, caching, authentication, monitoring, and load balancing [20].

To prevent the gateway from having too much logic to handle request shaping for different client types, it can be divided into multiple gateways. Each UI service or client type gets its own gateway as a collection point for service requests [16]. This describes the BFF pattern. The respective backend is closely linked to the corresponding frontend and is typically maintained by the same team, which is also responsible for the development of the frontend [17]. This approach avoids a general middleware and development teams can implement the frontend connections on this application level more independently [2].

4 METHODICAL APPROACH

To evaluate the presented frontend architectures and patterns an experimental analysis based on four prototypes was chosen, which allows studying the various architectures and patterns, both analytically and quantitatively [3]. All four prototypes implement the same basic e-commerce application. The basis for all prototypes are three microservices providing the core functionalities: a customer, a catalog, and an order microservice. The subsequent methodology was designed to evaluate these four prototypes.

4.1 Quality Model

The non-functional requirements specification of the prototypes is based on quality attributes. As the basis for the selection of quality attributes a quality model has been used, which in turn allows to evaluate the architectures with regard to non-functional requirements. During the definition of the quality model, the standard ISO/IEC 25010 [12] and the discourses of Bass et al. [4] were taken into account. Bass et al. describe seven quality attributes, which, according to their statements, are especially relevant, since they are often considered in software systems. These attributes include availability, interoperability, modifiability, performance, and testability. The quality attributes security and usability are not taken into account in the following analysis. Limiting the number of the quality attributes to be examined is necessary, since otherwise the evaluation would become too complex [19].

4.2 Test Cases

Subsequently, scenarios according to the Architecture Tradeoff Analysis Method (ATAM) [4] were defined for the specified attributes. ATAM is an approach to assess quality-related subjects of software architectures via scenarios. These scenarios make up the concrete functional requirements for the prototypes to be implemented and allow the conduct of concrete tests to evaluate the architecture styles (see Table 1 for an extract).

By means of the test case *availability-1-systemerrors*, e.g., the error handling of the UI service is evaluated. The stimulus is a request from the UI service to the Customer microservice, which queries customer data. In the test scenario, the Customer microservice is not available and it is measured whether the UI service displays evasion content. The performance test cases determine the time to interact (TTI), i.e. the time span before the user can interact with the GUI. The latency for the initial call of the homepage as well as the loading time for a page change in the shopping cart is measured. The modifiability test cases examine the effort, that is, the necessary number of services and files that needs to be changed, to meet specific new requirements. The test case *modifiability-1-featureextension* simulates an extension request for the order microservice that will allow users to access past orders.

The derivation of all other test cases is equivalent. The test results obtained are used to retain the data that is used to assess the architectures regarding the achievement of the quality attributes [26]. These evaluations make it possible to compare the different styles and develop a guideline for the selection of an architecture.

Table 1: Test case specification (extract)

Test case	Stimulus source	Stimulus	Environment	Artifact	Response	Response measure
availability-1-systemerrors	User	Request customer data	Normal operation	UI service	System runs in disturbed state	No downtime
interoperability-1-communication	order service	Total price sent	All systems during runtime	UI service	Total price correctly transferred to cart	Information transferred correctly
performance-1-latency	User	Loading of home-page and cart	All systems during runtime	UI service	Transaction is being processed	Average latency as low as possible
testability-1-dependencies	Automatic tests	Test run complete	Development	UI service interfaces	Interfaces are being validated	Amount of UI service interfaces
modifiability-1-featureextension	Extension request	Implementation of an Order history	All systems during runtime	UI and Order service	Convert with the least possible effort	Number of systems, components and/or files to be changed

4.3 Prototype Implementations

For the UI services, four prototype scenarios were created that implement the frontend architectures described above and integrate the microservices. The underlying structure of the individual microservices is very similar in all prototype scenarios. They are implemented with Spring Boot as well as Spring Cloud [25] and are characterized by a layered architecture, essentially consisting of a model, a persistence layer, and controllers that offer an appropriate interface for the respective clients. Depending on the prototype, the microservices are extended by an UI service or by a view layer, which either completely or partially delivers the UI.

The UI monolith is implemented as an SPA via AngularJS [1], following the MVC pattern. The monolith communicates with the microservices via service interfaces provided by REST-ful HTTP. In the second case, the microservices are integrated via the plugin approach using UI fragment composition. The Varnish [5] HTTP cache has been used as proxy server that uses ESI [32] for aggregating the UI fragments. Behind the proxy server, a frontend server, also implemented with Spring Boot and Spring Cloud, delivers the UI framework of the application. Here, the microservices that provide the fragments are referenced by ESI-*tags*.

In the third case, each microservice is instantiated as SCS by extending each service with its own view layer that delivers the complete UI. Correspondingly, the controller in an SCS does not implement a REST interface, but manages view paths. The SCSs are managed via an upstream Zuul [37] proxy server. Primarily, the proxy server implements the routing mechanism and acts as a gateway for users and forwards the requests to the corresponding services. All the mentioned components are created with Spring Boot and Spring Cloud.

In the fourth application, the API gateway pattern is applied by reusing the UI monolith from the first scenario and the microservices. The gateway is switched between UI monolith and microservices. In this case, no further frontend architecture is analyzed. However, the impact of a gateway as an intermediary between frontend integration and microservices can be investigated. The gateway contains a controller, which again offers REST-based interfaces. It also has clients that communicate with the microservices using REST-ful HTTP. The gateway sends requests from the monolith to the microservices, aggregates data it receives from the microservices, and sends them back to the UI monolith.

4.4 Testbed Implementation

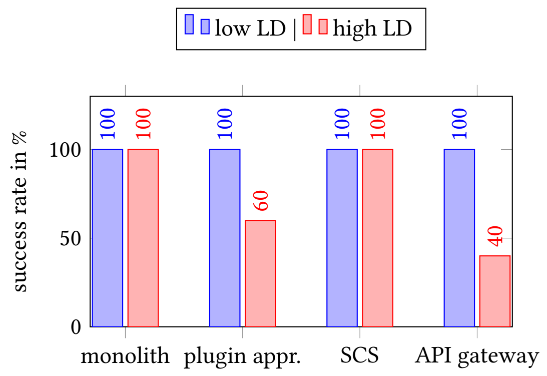
A MacBook Pro was used as workstation for the tests to be conducted. EC2 instances of the type *t2.large* (UI services) and *t2.medium* (microservices) were used in the test environment provided by Amazon Web Services (AWS) [24]. The basic operating systems differed accordingly. The MacBook Pro was based on OS 10.12.2, the EC2 instances on Amazon Linux AMI 2016.09.0. The services on AWS were operated via Docker containers [6]. As a result, automatic tests were carried out at runtime, with a simultaneous load generated by Apache JMeter [13].

On the workstation, automatic UI tests were performed addressing the AWS-powered services using Selenium [23] and TestNG [29]. Depending on the load scenario, the specified thread count varied as specified in the JMeter testplans. The ramp-up time was defined by ten seconds and the requests were repeated until the test script was terminated. For the low-load scenario 100 threads were started on each of three JMeter slaves. Theoretically, an average of 30 requests per second were sent when a query could be processed within one second. In the high-load scenario, the number of threads was increased to send up to 210 requests per second.

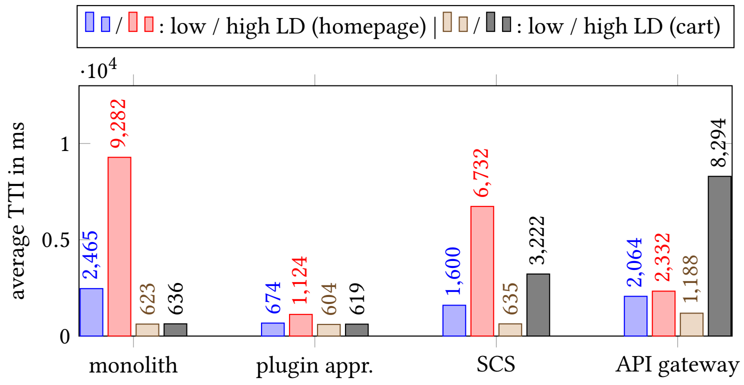
With this setup, the tests for the defined quality attributes availability, interoperability and performance were performed. Manual tests during the development period of the prototypes were also carried out in order to collect test results analytically for the attributes testability and modifiability.

5 RESULTS

The results show, that SCSs have the most advantages in the studied context. Above all, they promote the loose coupling of microservices at the frontend level. The responsible development teams are, e.g., also responsible for the UI of their microservice. This has a particularly favorable effect on availability, modifiability, and testability. In the availability test cases, only SCSs and API gateway remained 100% available in both load stages (see Table 2 (i)). Table 2 (ii) displays the results for the testability test case. Here, the plugin approach and SCSs have advantages since no interfaces between UI level and microservices have to be validated by integration tests. The results shown for the test case *modifiability-1-featureextension* in Table 2 (iii) illustrate that SCSs require the modification of fewer artifacts and files for a certain feature implementation. Figure 3 (a) shows that, in relation to interoperability, only the SCS prototypes



(a) Correct price transmission



(b) Loading time for homepage and shopping cart

Figure 3: Test results regarding (a) *interoperability* and (b) *performance*

and the prototype for the UI monolith returned the expected result without timeouts in the low load scenario as well as in the high load scenario. This was not the case for the other prototypes. Consequently, if an architecture with as loose components as possible is striven for, one should rely on SCSs.

If performance plays an important role, the plugin approach should be taken into account as it achieves very good results when used in conjunction with caching systems. Figure 3 (b) shows that the plugin approach achieved the highest loading speeds, during the initial loading of the homepage and when using the shopping cart. In addition, the corresponding UI fragment of the service is the responsibility of the respective development team [16]. Therefore, in the context of changes within a service, a similar good modifiability can be ensured with this approach as with SCS. The UI framework is implemented here by a superior service. This in turn simplifies the implementation of the global UI structure. But, on the other hand, the plugin approach entails a stronger link between the services as they are dependent on the superordinate UI framework. This means that both, SCS and the plugin approach, have disadvantages when it comes to modifications across service boundaries, since these must be coordinated for a consistent UI.

Therefore, the use of a UI monolith should be considered when UI changes occur frequently across service boundaries or if the application's domains cannot be unambiguously assigned to the microservices. The latter can especially be the case when a new

Table 2: Extracted results for (i) *availability*, (ii) *testability*, and (iii) *modifiability* test cases

Test case	monolith	plugin appr.	SCS	API gateway
(i) correct display of 404-error page (low / high load)	100% / 10%	100% / 100%	100% / 100%	100% / 100%
(i) correct display of evasion content (low / high load)	100% / 100%	100% / 40%	100% / 100%	100% / 100%
(ii) interfaces between UI and microservices	3	0	0	4
(iii) ΔServices	2	3	2	3
(iii) ΔFiles	6	5	3	8

project is launched and, in turn, leads to frequent modifications between microservices. In such scenarios, modifiability is better ensured by a UI monolith, although the development of UI and microservices is separated and the service interfaces must be adapted to the respective UI [16]. The reason for this is that the previously mentioned coordination of changes in the consistent UI is omitted. An UI monolith can also be considered if an event-oriented architecture is targeted for particularly loose coupling between the services. The reason for this is that events, in the logic layer of an application, are then coordinated through an event bus placed between UI service and microservices [36].

If other types of clients are integrated, such as mobile apps or desktop applications, it may be useful to use an API gateway or BFFs. The parallel integration of such diverse clients is difficult because they represent deployment monoliths. That is, changes in one microservice can easily lead to redeployments of all client types [36]. By using a gateway or BFF, a separation of the development of the services can be facilitated, which simplifies modifications to various releases of the client application. Deployment dependencies can therefore be avoided. However, the test results also show that a high load on a gateway can have a negative impact on the interoperability and performance of the UI service. Thus, it can become complex to reliably operate several gateway instances or BFFs.

Finally, two use cases, a television app and the online e-commerce platform of *otto.de* [27], confirmed the overall results of the tests conducted. The comparison shows, however, that in large systems, the usage of hybrid frontend architectures, combining SCS with monolithic or fragmented UIs, can bring synergy effects. The implementation of monolithic UI structures is, e.g., beneficial in the case of business processes that have not yet been unambiguously defined or are changing dynamically, since changes can occur more frequently over service boundaries. In these cases, the number of UI components to be adapted can be minimized. Fragmented UIs provide the possibility to maintain a UI framework in a centralized manner and can improve the performance in conjunction with cache systems. In this way one can benefit from the advantages of several frontend architectures simultaneously.

6 CONCLUSION

This paper analyzed methods for the integration of microservices into the frontend of client programs. Firstly, the leading frontend architectures were identified, which enable microservices to be integrated into frontends: UI monolith, plugin approach, and self-contained systems. In these approaches direct communication between the UI service and microservices takes place. For UI monoliths it is possible to perform this communication via a server, which encapsulates the internal system architecture and also takes on other tasks, such as data transformation. In this context, the API gateway and BFFs have become established.

An experimental approach based on various distinct prototypes was chosen for evaluating the different approaches since this allows to study the different architectures and patterns both analytically and quantitatively. The comparative approach made it possible to abstract a guideline for the selection of a suitable approach. It should be noted that the prototypes incorporate simplified versions of real-world applications. In order to further validate the presented results and to develop conclusions to a greater extent, more complex prototypes and testbeds are required, which also take into account further quality attributes. In this context, the test scenarios need to be expanded accordingly.

Nevertheless, the results in scope of this work show that the SCS approach brings the most benefits particularly in terms of modifiability and testability. In respect to performance the plugin approach in combination with caching systems provides very good results. The UI monolith has advantages when changes at the UI level occur often across service boundaries. Since the UI is concentrated in a single UI component, there are no coordination efforts required between the development teams to carry out the changes. The patterns API gateway and BFF are mainly used when multiple client types integrate the same microservices-based system. Here, they can take over many different functions, but above all bring advantages with regard to the dedicated data preparation for the clients and the reduction of deployment dependencies. Overall, the architectures of microservice-based systems, should primarily promote loose coupling and strong cohesion [16]. These principles foster independent development, deployment, and scalability.

REFERENCES

[1] Angular. 2017. Angular. Online. (2017). https://angular.io/ - Visited 2017-04-25.

[2] Richard Attermeyer. 2016. Frontend-Architekturen für Microservice-basierte Systeme. Online. (2016). http://www.sigs-datacom.de/uploads/tx_dmjournals/attermeyer_OTS_Microservices_Docker_16.pdf - Visited 2017-04-25.

[3] Jakob Eyvind Bardram, Henrik Bærbak Christensen, and Klaus Marius Hansen. 2004. Architectural prototyping: An approach for grounding architectural design and learning. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on.* IEEE, 15–24.

[4] Len Bass, Paul Clements, and Rick Kazman. 2013. *Software architecture in practice.* Addison-Wesley, Upper Saddle River, NJ.

[5] Varnish HTTP Cache. 2017. Varnish HTTP Cache. Online. (2017). https://varnish-cache.org/ - Visited 2017-04-25.

[6] Docker. 2017. Docker - Build, Ship, and Run Any App, Anywhere. Online. (2017). https://www.docker.com/ - Visited 2017-04-25.

[7] Thomas Erl. 2005. *Service-oriented architecture: concepts, technology, and design.* Pearson Education India.

[8] Viktor Farcic. 2015. Including Front-End Web Components Into Microservices. Online. (Aug. 2015). https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/ - Visited 2017-04-25.

[9] Roy Fielding. 2000. Representational state transfer. *Architectural Styles and the Design of Network-based Software Architecture* (2000), 76–85.

[10] Martin Fowler. 2016. Microservices Resource Guide. Online. (2016). http://martinfowler.com/microservices/ - Visited 2017-04-25.

[11] Bartosz Galek, Walacik Bartosz, and Pawel Wieladek. 2016. Managing Frontend in the Microservices Architecture. Online. (March 2016). http://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html - Visited 2017-04-25.

[12] ISO. 2011. *IEC25010: 2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation.* Standard. International Organization for Standardization, Geneva, CH. http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 - Visited 2017-04-25.

[13] Apache JMeter. 2016. Apache JMeter. Online. (2016). http://jmeter.apache.org/ - Visited 2017-04-25.

[14] James Lewis and Martin Fowler. 2014. Microservices. Online. (March 2014). http://martinfowler.com/articles/microservices.html - Visited 2017-04-25.

[15] Suresh Marru, Marlon Pierce, Sudhakar Pamidighantam, and Chathuri Wimalasena. 2015. Apache Airavata as a laboratory: architecture and case study for component-based gateway middleware. In *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models.* ACM, 19–26.

[16] Sam Newman. 2015. *Building Microservices.* O'Reilly Media, Inc.

[17] Sam Newman. 2015. Pattern: Backends For Frontends. Online. (Nov. 2015). http://samnewman.io/patterns/architectural/bff/ - Visited 2017-04-25.

[18] Likasz Plotnicki. 2015. BFF @ SoundCloud. Online. (Dec. 2015). https://www.thoughtworks.com/de/insights/blog/bff-soundcloud - Visited 2017-04-25.

[19] Torsten Posch, Klaus Birken, and Michael Gerdom. 2011. *Basiswissen Softwarearchitektur: verstehen, entwerfen, wiederverwenden.* Dpunkt Verlag GmbH, Heidelberg.

[20] Chris Richardson and Floyd Smith. 2016. Microservices: From Design to Deployment. Online. (May 2016). https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/ - Visited 2017-04-25.

[21] ROCA. 2016. ROCA – Resource-oriented Client Architecture. Online. (2016). http://roca-style.org/ - Visited 2017-04-25.

[22] D.I. Savchenko, G.I. Radchenko, and O. Taipale. 2015. Microservices validation: Mjolnirr platform case study. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on.* 235–240. DOI: https://doi.org/10.1109/MIPRO.2015.7160271

[23] SeleniumHQ. 2016. Selenium - Web Browser Automation. Online. (2016). http://www.seleniumhq.org/ - Visited 2017-04-25.

[24] Amazon Web Services. 2017. AWS – Server Hosting & Cloud Services. Online. (2017). https://aws.amazon.com/de/ - Visited 2017-04-25.

[25] Pivotal Software. 2017. Spring. Online. (2017). https://spring.io/ - Visited 2017-04-25.

[26] Gernot Starke. 2015. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden.* Carl Hanser Verlag GmbH und Co KG.

[27] Guido Steinacker. 2015. On Monoliths and Microservices. Online. (Sept. 2015). https://dev.otto.de/2015/09/30/on-monoliths-and-microservices/ - Visited 2017-04-25.

[28] Roman Stranghöner. 2015. Self-Contained Systems. Online. (March 2015). https://speakerdeck.com/rstrangh/self-contained-systems-german - Visited 2017-04-25.

[29] TestNG. 2016. TestNG. Online. (2016). http://testng.org/doc/ - Visited 2017-04-25.

[30] Johannes Thones. 2015. Microservices. *Software, IEEE* 32, 1 (Jan. 2015), 113–116. DOI: https://doi.org/10.1109/MS.2015.11

[31] Stefan Tilkov. 2014. Web-based frontend integration. Online. (Nov. 2014). https://www.innoq.com/blog/st/2014/11/web-based-frontend-integration/ - Visited 2017-04-25.

[32] Mark Tsimelzon, Bill Weihl, Joseph Chung, Dan Frantz, John Basso, Chris Newton, Mark Hale, Larry Jacobs, and Conletz O'Connell. 2001. ESI Language Specification 1.0. Online. (Aug. 2001). https://www.w3.org/TR/esi-lang - Visited 2017-04-25.

[33] Aad Versteden, Erika Pauwels, and Agis Papantoniou. 2015. An Ecosystem of User-facing Microservices Supported by Semantic Models. In *USEWOD-PROFILES@ ESWC.* 12–21.

[34] Mario Villamizar, Oscar Garces, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th.* 583–590. DOI: https://doi.org/10.1109/ColumbianCC.2015.7333476

[35] Arif Wider and Johannes Mueller. 2016. An Unexpected Solution To Microservices UI Composition. Online. (Jan. 2016). http://inside.autoscout24.com/talks/2016/01/13/microservice-ui-composition/ - Visited 2017-04-25.

[36] Eberhard Wolff. 2016. *Microservices: Grundlagen flexibler Softwarearchitekturen.* Dpunkt Verlag GmbH, Heidelberg.

[37] zuul. 2016. zuul. Online. (2016). https://github.com/Netflix/zuul - Visited 2017-04-25.

[38] Stefan Zörner. 2016. Bring your own Architecture. *Entwickler Magazin Spezial* 9 (2016), 16–19.