EVS30 Symposium
Stuttgart, Germany, October 9 - 11, 2017

Microservice-Architecture - a Practical Approach to Developing a Distributed Heterogeneous-Fleet-Management-Platform

Ilko Hoffmann[1], Julien Ostermann[2], Kavivarman Sivarasah[3]

[1]*Ilko Hoffmann, Fraunhofer IAO, Stuttgart, Germany, Ilko.Hoffmann@iao.fraunhofer.de*

[2]*Julien Ostermann, Julien.Ostermann@iao.fraunhofer.de*

[3]*Kavivarman Sivarasah, Kavivarman.Sivarasah@iao.fraunhofer.de*

Summary

In order to run heterogeneous fleets economically complex holistic management solutions are required. Developing, delivering and maintaining such ambitious software systems are challenging tasks. The microservice approach seems capable of improving productivity by increasing understatement of the system with encapsulated, cohesive and loosely coupled functionalities. This paper examines the impact on productivity by a possible reduction of complexity implementing Microservices with a practical approach to developing a distributed fleet management platform.

Keywords: car-sharing, electric drive, fleet, load management, mobility system

1 Introduction

In [1] Moseley et al. state that problems and costs in developing software are mainly caused by its presented complexity and the related lack of understanding. Other properties as conformity, changeability and invisibility identified 1986 by Brooks [2] as major issues in developing software are therefore just results to the extent of complexity occurring in the software system.

By its nature evolving monoliths often suffer by intransparent dependencies and control flows due to its lack of modularity. Any changes made on those systems require rebuilds and deployments and are therefore increasing the risk of downtimes and eventual monetary loss [3]. Adopting the software system to altering circumstances can be difficult, time consuming and expensive. Principles of agile software development can´t be realized in a proper and effective manner. We agree with the premise that many kinds of essential complexity in developing and maintaining software can be avoided by making sure to develop comprehensible stripped down and loosely coupled encapsulated functionalities. Using the Microservice-Architecture (MSA) paradigm for software development seems suitable for realizing large-scale software systems with an improvement of understanding and reduction of complexity [4].

Based on a practical approach to developing a cloud-ready heterogeneous fleet management platform this paper investigates the fulfilment of promises given by using the MSA approach in terms of flexibility, modularity and evolution in developing and maintainment of ambitious software systems [5]. Section 2 introduces related work regarding the topic of distributed systems and demonstrates the evolution of MSA

out of the Service-Oriented-Architecture (SOA) paradigm. Section 3 briefly discusses the basic requirements for the subdivision of software systems following the Domain-Driven-Design (DDD) approach. In section 4 the project eMobility-Scout (EMS) is introduced putting the DDD into practice revealing the idea of separation of concerns with namespaces. Section 5 examines the impact of Microservices on productivity in the context of EMS and the practical approach in order to reveal the relation of benefits and downsides.

2 Related Work

In literature, a lot of research has been done in the field of distributed system architecture [6]. Before the MSA paradigm evolved, distributed systems have been composed along the SOA paradigm [5]. Instead of monolithic applications, services were introduced to split up large-scale applications such as enterprise resource planning (ERP), supply chain management (SCM) or customer relationship management (SRM) systems. [7] Thus, different operation systems and programming languages could be supported to provide a better interoperability between systems. In the field of MSA, no work could be found which deals with MSA in the domain of fleet management and energy management. Only contributions in other domains have been published. In [8] Bao et al. presents a MSA for internet of things and energy in smart building, based on research challenges in the field of smart buildings. Cherradi et al. [9] investigated the necessary requirements and proposed a Microservice based architecture for a real-Time HazMat environmental information system. This shows that MSA in general and in the field of fleet management and energy management is still evolving and our contribution is a viable addition to the state-of-the-art.

In a previous work, we already applied a SOA to a distributed electric vehicle (EV) fleet management system in the project Shared E-Fleet (SEF) [10]. There, various domain specific services were connected to each other via an enterprise service bus (ESB) in order to achieve a flexible and loosely coupled system. Through a mixture of event based and fixed peer-to-peer based communication, this resulted in a very complex, tightly coupled and consequently inflexible system. As a matter of fact, services could not be simply replaced due to distributed state management and distributed service functionalities. As a result, the system was incapable of scaling and highly error-prone. These drawbacks can be solved by using MSA as stated by the authors in [3], [4], [11].

Thus, in this paper, we present a MSA of a fleet management system for heterogeneous vehicle fleets that is highly scaleable and maintainable due to independency in development, deployment and operation of Microservices [5].

3 Design Principles of Microservice Architectures

Emerging problems caused by essential complexity of ambitious software projects are compounding the difficulties of development and maintainance of those systems [12]. Reducing this kind of complexity by introducing Microservices to improve comprehensibility and diminish coherent costs is a promising approach. Being able to examine the potential advantages, requirements regarding the principle software design have to be fulfilled in the first place. Developers have to make sure that services are loosely coupled and cohesive as much as possible to avoid prospective problems. Unexpected und unwanted behaviours are often caused by unpredictable states or intransparent control flows of the system itself [1]. Keeping data and functionality encapsulated is conducive to understand the system which leads to better software quality and less errors being made by humans [13]. Processes and control sequences in modular designed software systems are potentially easier to follow and understand in comparison to those performed by big opaque monoliths. Emphasizing the importance of designing highly cohesive and loosely coupled services with clear separations of stateless and stateful functionalities leads to the assumption that those characteristics are able to reduce complexity in an effective manner [4].

3.1 Separation of Concerns

Defining the dimension of Microservices can be a challenging and time consuming task. Fowler [3] states that the accurate size of a service can vary from project to project and has still not been defined in a

satisfying manner due to differing requirements and circumstances. In spite of this fact the impact of choosing the size and resulting boundaries of a service on development and maintainance of software must not be underestimated. Following the approach of DDD introduced by Evans [14] creating systems on domain level is supposed to bring various advantages in terms of transparency and understatement. According to DDD such domains consist of a set of bounded contexts with encapsulated functionalities and well defined boundaries. The communication processes of the domain realized by interacting bounded contexts are therefore corresponding to the communication processes on business level. This approach also includes the awareness of which informations are supposed to be shared with other contexts and which are supposed to stay hidden [4]. Understanding the communication and control flow of the business process enhances comprehensibility not only to developers and is capable of reducing complexity and thus the quality of the software.

3.2 Contract First

Looking at software design on business level supports the idea of focussing on capabilities of single services rather than on thinking of how they are actually implemented. Although the microservice approach facilitates modifications of services it´s essential to be aware of the impact of changes of communication interfaces on the software system and dependent service chains. In order to be able to provide complex and elaborate functionalities Microservices have to cooperate with each other [5]. The approach of choreographic communication has no need for a centralized conductive controlling instance and synchronized communication pattern and achieves therefore a high degree of decoupling. A contract in this context is defined as an agreement between service provider and service consumer on which data and functionality a certain service is offering [15]. Thus, it´s even more important to develop web interfaces which ideally minimize the need of modification and are able to evolve without causing changes in other components. The concept for design, documentation, publication and evolution of those interfaces is particularly important in this kind of environment [11]. Consumer driven contracts can help to reduce the risk of availability issues caused by inappropriate changes made by the service provider [16]. Designing smart endpoints in a sustainable way and keeping associated logic inside of service boundaries eases the maintainment of high cohesion and loose coupling of services [4].

4 The eMobility-Scout Project

The eMobility-Scout project is a federal funded project, which aims to build a highly scalable electric vehicle management system for different kind of electric vehicles in corporate fleets to lower the bar for adaption, decrease complexity in usage as well as provide economically viable usage of the vehicles in the everyday life. Therefore, a system is designed and implemented to manage, monitor and control all associated aspects of such a mobility system. This includes an adequate disposition of vehicles to the user demand to tackle range anxiety, monitoring of vehicle statuses, energy consumption and operation as well as directly controlling charging processes and energy management. Because of the expected exponentially increasing adaption of electric vehicles and the subsequent need for an intelligent, scalable and adaptable management system arises. Furthermore, in order to reduce total cost of ownership and related financial risks regarding the investment into electric vehicles in corporate fleets, holistic management and mobility solutions are required [17]. Together with adaptive and new business models this management solutions are capable of reducing the break-even-point of initial costs and benefits for the company [18].

In detail, the resulting software solution developed in the project EMS is supposed to unite optimization procedures for energy and disposition management to facilitate the economical operation of heterogeneous fleets.

4.1 Design of Namespaces

Due to the collaboration between three partners and development teams involved in the EMS project an early separation of responsibilities on an abstract level was required. Respectively to the expertise of each partner the domain got separated in various namespaces like fleet-management, hardware-integration and resource optimization for instance. This coarse-grained approach of division allows the grouping of related functionality and thus reduces the burden of change management later on [15]. Furthermore, it facilitates

the design of consumer driven service contracts between collaborative companies on business level. The responsibility of defining service boundaries inside those namespaces is therefore left to the expertise of the specific owner. In Figure 1 the boundaries of those namespaces and services in the context of EMS are depicted which are explained in further detail in the following.

4.2 Service Specialization

In order to be able to run and maintain the distributed EMS software system a differentiation between three different kinds of service types were required. Hence, we differentiated between basic, global and functional services (see Figure 1). Services considered as basic services are responsible for tasks, which are required by all other services. Through the distributed nature of the system, we introduced centralized services responsible for configuration, logging and asynchronous event-driven messaging. Global services are specialized services, which manage the cloud-nature of the application in order to enable fundamental management layers. Furthermore, they establish the capabilities for user authentication, tenant management, messaging functionality and menu navigation for all functional services. Functional services implement the business logic of each namespace related services. Functional services can use basic and global services to enable collaboration between other services, monitoring and troubleshooting. The service namespaces are defined as followed.

4.2.1 Fleet Management

The holistic digital management of corporate car fleets in EMS provides various functionalities like the management for bookings, vehicles, pooling, damages and locations similar to SEF and EcoGuru [10], [17]. Furthermore, the possibility of merging and transparent tracking costs in a global operational context is implemented by a separate Microservice preparing the possibility of external billing. Exposing functionalities to involve end users like fleet-manager, disposition-manager and drivers via the user interface of the management platform are supported. In order to facilitate the process of driving the implementation and connection of a proprietary key box is planned to automate the car key exchange. The monitoring service in EMS merges the operational key performance indicators of the mobility system in runtime. Depending on the individual configuration of specific thresholds stakeholders can be informed about current or prospective operational misfunctions in order to be able to take appropriate actions and prevent potential problems.

4.2.2 Live-Datapool

Providing real time data is one of the core concepts of EMS and therefore essential for the realization of important use cases like disposition or energy optimization and thus the economical integration of electric vehicles. The Live-Datapool is collecting and processing data of the charging infrastructure and corporate fleet. The charging points are organized in clusters with centralized data access points. In contrast each vehicle is equipped with an on-board unit (OBU) connected to the CAN-Bus of the car collecting sensor data. The datapool is receiving the data of the charging infrastructure and vehicles sent via the mobile network in preconfigured intervals.

4.2.3 Charging-Infrastructure

Communicating to embedded systems like charging stations require the implementation of middleware responsible for the translation of requests in proprietary communication protocols. This middleware in the context of EMS can be used bidirectionally and enables the remote control of charge point clusters or modification of their energy flows in realtime through a user interface or predefined charging protocols.

4.2.4 Optimization

As already stated in SEF the scheduling of electric cars require more intelligent disposition and energy consumption algorithms than conventional combustion cars [10]. These optimization algorithms are aiming to find the best possible match of vehicle and booking considering energy consumption goals at the same time avoiding factionary payment by the energy provider due to exceeding a contractually set limit of energy consumption.

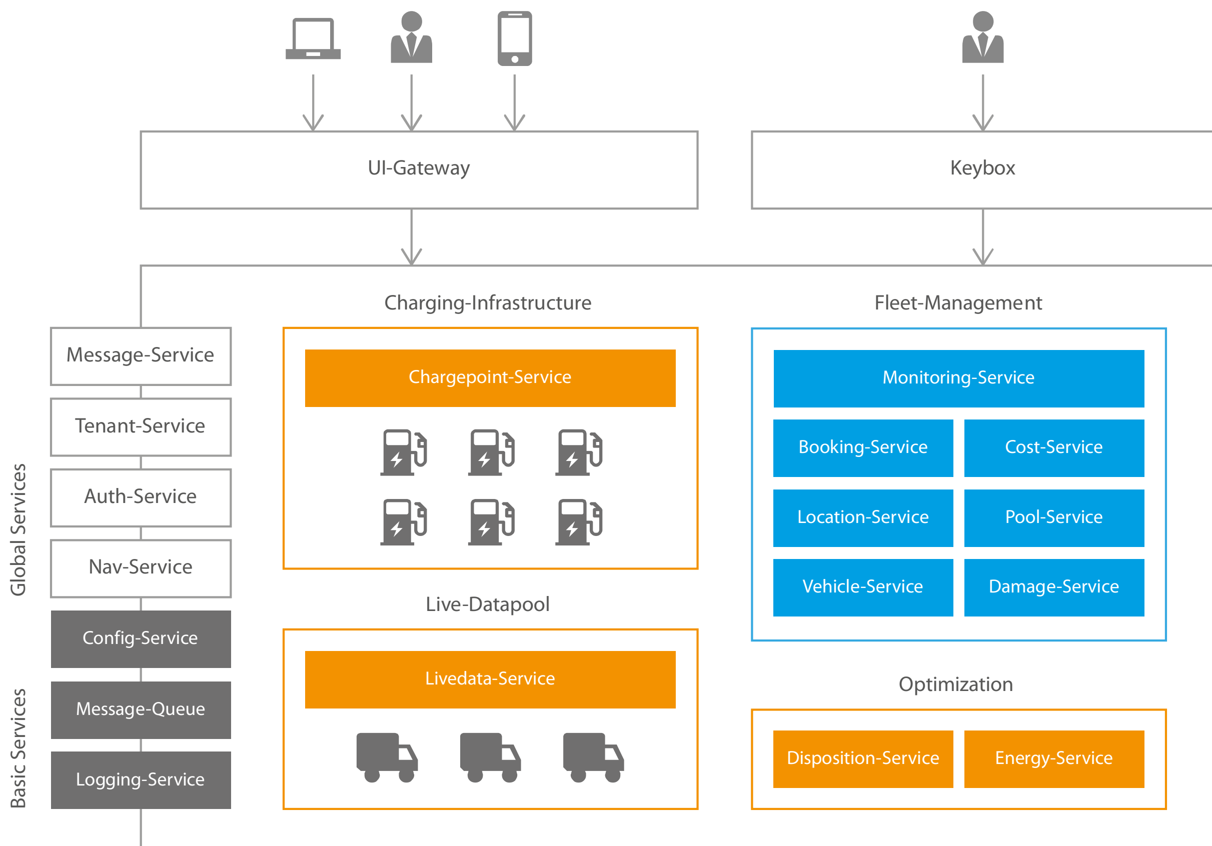

Figure 1: Coarse-grained namespaces in EMS

5 Impact of Microservices on Productivity

Choosing Microservices over monoliths isn´t necessarily suited to all application scenarios. When developing a greenfield project for instance Microservices can come along with a lot of potential benefits. Although companies like Netflix and Amazon already established the proof of concept by implementing big enterprise microservice systems it´s important to state that application scenarios vary and those benefits do not come for free. Set up and manage a bundle of basic services first in order to run the architecture is one of the downsides which can produce overhead and additional complexity before even starting to write the first line of business logic. Fowler [19] for instance recommends not to begin a project from scratch with a microservice approach. Developers should rather start off with a monolith in order to be productive as fast as possible. Breaking services out later as the knowledge of the system and boundaries increases will accelerate the productivity. Being aware of the fact that setting up a microservice ecosystem is causing overhead the following fields are supposed to be evaluated by the practical approach to development of the distributed system in EMS.

5.1 Agility and Deployment

Microservices were considered as a best practice model by our development team in order to achieve quick progess in evolving the services. Soon we realized that an agile team organisation considering code prior to documentation and a 100% automation of integration, delivery and deployment of the developed code and software artefacts were some of the major requirements to get succeeded with our approach. Furthermore, several frameworks and libraries are needed for the development and operation of Microservices which demands a high level of technical understanding and knowhow from the developers. Considering the collaboration of distributed teams belonging to three different companies and the required coordination and

agreements on the basic architecture exploiting the advantages of Microservices took some time and several reconciliations.

After reaching a collective agreement the free choice of technology stacks depending on the skillsets of the teams was starting to pay off. As Microservices are meant to be independently deployable each team used its most preferred programming languages, libraries and data storages for the development which led to a high technological diversity. Additionally, services could be developed independently and software artefacts were delivered in various iterations just by taking the service contract agreements in consideration. In conclusion we were able to deliver new features more frequently and faster in a loosely coupled way. Using a continuous delivery pipeline on an automation server like Jenkins [20] helped with testing, building and deploying the services to the staging environment in a fully automated way.

5.2 Security and Confidentiality

In order to prevent abuse by unauthorized third parties it's important to implement security concepts which cover confidentiality and integrity aspects in distributed systems. Confidential data frequently transferred over network is always exposed to the danger of being attacked by others. Therefore it´s necessary to assure that essential information is always available and unauthorized access is avoided simultaneously [5].

Regarding APIs as doors to the outside world securing them with authentication- and authorization mechanisms seem to be inevitable. In EMS the entrypoint for requests is the gateway of each namespace which secures the communication with Transport Layer Security (TLS) and proxies incoming requests to the appropriate services. Since our Microservices provide various endpoints requiring different access rights and following the loosely coupling approach the responsibility of securing proprietary resources were shifted to the providing services themselves.

Requests sent to the service APIs are expected to have an authorization header containing a JSON Web Token (JWT), which is issued by the authentication-service (see Figure 1, Auth-Service). The authenticity of the JWT is verified first to ensure that the token is actually issued by the Auth-Service and has not been modified since then. Our tokens have a limited validity period of eight hours in accordance with the length of a common working day. The custom payload of the token includes all the essential information about the user or consuming service like tenant details, unique id and assigned authorities. Besides decoding and validating the token itself, the authority information determines if the consumer is allowed to access a certain API resource. In case a request triggers further services to be called, the JWT provided in the origin request is allowed to be passed throughout multiple Microservices. Therefore, it is necessary to implement specific security layer on each Microservice in order to secure the whole system properly.

5.3 Reliability and Availability

Management systems need to be reliable and stable. Because of their limited amount of encapsulated functionalities reaching higher test coverages in Microservice-Architectures should be possible [5]. For this reason developers have to emphasize the integration of automated test pipelines in order to avoid system errors and downtimes. Implementing procedures such as continuous integration and deployment into the system lead to benefits improving cycle-times between developing and integration of services [21], [22].

With the number of implemented services, the number of APIs between the services increases. Being aware of direct and indirect dependencies between the services not only minimizes the overhead of the http communications but also reduces the error-proneness. Although in terms of graceful degradation the services are conceived to be stable and able to provide a minimum set of functionalities even if dependent services are currently not reachable. Even though the dependent services could be developed and delivered in various iterations, the service interfaces should still be compatible to stay stable and reachable. If an API is released once and a newer API is to be introduced, the older one should stay up until the last consumer has switched to the newer version of the API. This is achieved by implementing co-existing endpoints with an incremented version number. Although we should be aware of the effort that comes with the maintenance of multiple API-versions as they could implement various business logics.

To ensure a high reliability of our services 100% test automation is needed. The continuous delivery pipeline mentioned in section 5.1 already encloses a continuous integration of the code committed to a

version control system. It includes an automated testing especially unit testing and integration testing which ensure the code quality of the implementation and that the Rest-APIs are still fulfilling the consumer-driven-contracts. To enable encapsulated testing of services that depends on other services mocking frameworks like Mockito [23] are used.

5.4 Maintainment and Operation

Although small services can be developed and maintained by small teams it also calls for experience due to bigger operational and coordinative complexity. Decentralized data management and related consistency issues are leading to further challenges [22]. Transactions made in distributed systems require multiple resources on different Microservices to be updated. Therefore it's needed to implement functionality into the system capable of handling and establishing eventual consistency concepts [21]. Taking decisions based on inconsistent data can lead to serious mistakes which in worst case will become expensive for operators.

Issues related to decentralized data management and potential inconsistency are handled by asynchronous communication patterns using publish-subscribe mechanisms. A centralized message broker is responsible to handle topics, receives messages and broadcasts those to subscribed services. The payload of the messages contains patches in JSON data structure which only consists of attributes which have been since the last exchange between the owner and consumer. The deactivation of a vehicle for instance triggers the schedule plan to be optimized by the disposition-service (cf. Figure 1, Disposition-Service), the charging schedule to be changed and by implication notifications to be sent out. Using asynchronous messaging we avoid problems caused by direct service-to-service communication and are introducing the approach of decentralized choreographic communication pattern to EMS.

Reducing complexity of software aims for reaching the degree of transparency to a certain amount. Keeping track of system states by following the control flow sequentially supports developers and administrators to understand. For practical reasons it´s necessary to provide centralized logging solutions in order to enable the possibility to do so. In EMS we took advantage of the ecosystem provided by the Elastic Stack [24] to implement such centralized logging systems. Requiring all services to implement the additional communication and therefore increasing the individual complexity first developers are able to examine the operation of the system and improve transparency and understatement.

5.5 Flexibility and Evolution

As illustrated in Figure 1, our system architecture consists of a several number of microservices as single purpose applications and additional services that help managing, monitoring and operating the infrastructure. A subset of the Microservices are running in lightweighted, virtualized Docker containers that are meant to be highly scalable, whereas another part is directly hosted on the Amazon's cloud service platform AWS (Amazon Web Services). AWS allows scaling the services in a convenient way, as scaling the self-hosted docker containers should be set up additionally. Load balancing mechanisms enable the ability of selective scaling with the reproduction of Microservices delegating the occurring load to various endpoints automatically.

After the successful implementation of basic services required by a MSA in order to be able to operate, the system is prepared to be extended in functionality and size. Loosely coupled services come along with the possibility to exchange poor implementations in order to improve performance of certain functionalities. Considering the service contracts Microservices can be exchanged on runtime if necessary due to the capabilities of continuous deployment as stated in section 5.1. Implementing the MSA in EMS the fleet management platform can now be extended effectively minimizing the risk of downtimes and is capable to grow flexible and rapidly.

6 Conclusion and Future Work

While there are many potential benefits coming along with MSA there are also many challenges to resolve in the first place. Those challenges include the setup of basic services responsible for communication, monitoring and configuration building the base to run such architecture before being able to deliver productive code. There are many open source tools on the market allowing developers to implement such

services by themselves saving a lot of work. In spite of this a deep technical understatement and experience of the developer is required. Big service provider like Amazon, Google and Microsoft are offering a cloud service ecosystem facilitating the implementation of MSA a lot. However reducing complexity there are still many decisions to make. Issues like security, reliability and delivery have to be resolved in an effective manner in order to be able to benefit from advantages claimed by Microservices. Getting deeper into this topic during the implementation of EMS setting up the architecture from scratch it has to be emphasized that the overhead was slowing the development down particularly in the beginning.

Considering the impact on productivity it´s important to state that implementing a monolith might be the better choice in some cases. In spite of this it´s hard to tell if a project is suited to the characteristics of a MSA approach or not and has to be decided form case to case. Gaining experience with those kinds of loosely coupled systems the break-even-point will certainly be reached early. Even so the size of the planned project seems like an accurate indicator and base for the decision if the implementation of Microservices is appropriate or a monolith is preferable. Reaching the break-even-point however will result in an increasement of productivity, scalability and understatement of big complex software systems and hence, improve the development process in terms of agility and cycle times.

In case of EMS project the development of a mobility platform the microservice approach seems appropriate considering the possibilities of further development and achieved potential for evolution. In future work, we will break down the mentioned services into building block like functionalities. By this, services can be fully reduced to their main functionality achieving building block-like composability, to enable reusability of their business logic and quasi-standardized RESTful interface definition. Especially in the field of software-as-a-service applications, this would increase efficiency in the development of new applications when the software architect knows which services can be substituted with already existing service componentsn

Acknowledgments

This research has been supported by the IKT III program in the eMobility-Scout project. They are funded by the German Federal Ministry of Economics and Technology under the grant number 01ME15006D. The responsibility for this publication lies with the authors.

References

[1] B. Moseley and P. Marks, “Out of the tar pit,” *Software Practice Advancement (SPA)*, pp. 1–66, 2006.

[2] F. P. J. Brooks, “No silver bullet-essence and accidents of software engineering,” *Proceedings of the IFIP Tenth World Computing Conference*, pp. 1069–1076, 1986.

[3] L. James and M. Fowler, “Microservices.” [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 19-Jan-2017].

[4] S. Newman, *Building Microservices*. 2015.

[5] N. Dragoni *et al.*, “Microservices: yesterday, today, and tomorrow,” 2016.

[6] K. Channabasavaiah, K. Holley, and E. M. Tuggle, “Migrating to a service-oriented architecture,” *IBM DeveloperWorks*, 2004.

[7] Z. Mahmood, “Service Oriented Architecture: Tools and technologies,” *Proceeding of the 11th Wseas International Conference on Computers: Computer Science and Technology, Vol 4*, 2007.

[8] K. Bao, I. Mauser, S. Kochanneck, H. Xu, and H. Schmeck, “A microservice architecture for the Intranet of Things and energy in smart buildings,” in *Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA 2016*, 2016.

[9] G. Cherradi, A. El Bouziri, A. Boulmakoul, and K. Zeitouni, “Real-Time HazMat Environmental Information System: A micro-service based architecture,” *Procedia Computer Science*, vol. 109, pp. 982–987, 2017.

[10] J. Ostermann, T. Renner, F. Koetter, and S. Hudert, “Leveraging electric cross-company car fleets through cloud service chains: The shared E-fleet architecture,” in *Annual SRII Global Conference, SRII*, 2014, pp. 290–297.

[11] M. Villamizar *et al.*, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Colombian Computing Conference, 10CCC 2015*, 2015.

[12] F. J. Corbató, “On building systems that will fail,” *Communications of the ACM*, 1991.

[13] E. W. Dijkstra, “The humble programmer,” *Communications of the ACM*, 1972.

[14] E. Evans, “Tackling Complexity in the Heart of Business Software,” vol. 7873, no. 415, p. 529, 2002.

[15] Y. Zhong and J. Yang, “Contract-first design techniques for building enterprise web services,” in *2009 IEEE International Conference on Web Services, ICWS 2009*, 2009.

[16] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Migrating to Cloud-Native architectures using microservices: An experience report,” in *Communications in Computer and Information Science*, 2016, vol. 567.

[17] D. Satikidis, K. Sivarasah, K. Lehmann, I. Hoffmann, and G. Scheffler, “EcoGuru – A system for the integrated management of electrified mobility systems,” in *15. Internationales Stuttgarter Symposium*, Springer Fachmedien Wiesbaden, 2015, pp. 217–232.

[18] C. Lerch, “New business models for electric cars: A holistic approach,” *Energy Policy*, vol. 39, no. 6, pp. 3392–3403, 2010.

[19] M. Fowler, “MonolithFirst,” *2015-06-03*. [Online].
Available: https://martinfowler.com/bliki/MonolithFirst.html. [Accessed: 29-Jun-2017].

[20] “Jenkins - Build great things at any scale.” [Online].
Available: https://jenkins.io/. [Accessed: 30-Jun-2017].

[21] M. Fowler, “Microservice Trade-Offs,” 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html. [Accessed: 19-Jan-2017].

[22] A. Krylovskiy, M. Jahn, and E. Patti, “Designing a Smart City Internet of Things Platform with Microservice Architecture,” in *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, 2015, pp. 25–30.

[23] “Mockito framework site.” [Online]. Available: http://site.mockito.org/. [Accessed: 30-Jun-2017].

[24] “An Introduction to the ELK Stack (Now the Elastic Stack).”

Authors

Ilko Hoffmann (B.Sc. B.A.) is research associate at the Fraunhofer Institute for Industrial Engineering IAO in Stuttgart since 2016. Currently working at the University of Applied Science in Esslingen in the Application Center KEIM he is doing research in areas around New Mobility Concepts, Internet of Things and Smart Applications with focus on Distributed-Systems, Agile Development and User-Experience.

Julien Ostermann (M.Sc.) is research associate at the Fraunhofer Institute for Industrial Engineering IAO in Stuttgart since 2013. He is doing research in areas around Smart Energy Solutions and Electric Mobility with focus on Energy Optimization Procedures, Holistic Fleet-Management Solutions, Agile Development and Distributed-System-Engineering.

Kavivarman Sivarasah (M.Sc.) is research associate at the Fraunhofer Institute for Industrial Engineering IAO in Stuttgart since 2012. Currently working at the University of Applied Science in Esslingen in the Application Center KEIM he is doing research in areas around New Mobility Concepts, Distributed Data Management and Distributed System Engineering.