Efficient Architecture Design for Software as a Service in Cloud Environments

Pavel Seda[1], Pavel Masek[1], Jindriska Sedova[3], Milos Seda[2], Jan Krejci[1], and Jiri Hosek[1]
[1]Department of Telecommunications, Brno University of Technology, Brno, Czech Republic
[2]Institute of Automation and Computer Science, Brno University of Technology, Brno, Czech Republic
[3]Department of Law, Masaryk University, Czech Republic

***Abstract*—With the increasing popularity of Software as a Service (SaaS) clouds, software architects face new challenges in software architecture design, e.g. efficiently integrating those chosen web services into clouds. In this respect, early quality evaluation of the designed SaaS cloud application is crucial to mitigate the risk of later architectural changes due to a violation of quality requirements (such as response time, network throughput and scalability). Architects need to navigate in a rich set of offered services, a variability of the SaaS cloud environments, which makes it difficult to predict the general approach without time-consuming implementation of application prototypes (using GraphQL, Representational State Transfer (REST), Simple Object Access Protocol (SOAP), and Application Programming Interfaces (APIs) approaches). In this paper, we present an abstract application model designed to select the details with a high level of scalability and efficient interactions between service components. This model is reusable and provides new way of thinking about cloud architecture.**

***Index Terms*—cloud computing, GraphQL, SaaS, sofware architecture, REST, web services**

I. Introduction

No doubt that clouds have been developed significantly in the last decades. Starting from a wide extension of internet services, providers need to integrate applications and services into infrastructures and efficiently set the communication channels between service components and also with their clients, e.g. with using web services. The clouds are based on cloud computing which is clarified in Def. 1 [1].

Definition 1. Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

The evolution of developing cloud computing started in the 1950s with mainframe computers. That time users communicated using terminal with a central computer. Unfortunately, this terminal was able to perform only connection with a mainframe computer and also the cost of this mainframe computer was not economically feasible for companies to buy them. But this idea was the root of cloud computing since companies did not needed to manage these computers, but they need to access them. In later 1960s and 1970s, this concept was supported using virtualisation which refers to the act of creating a virtual version of something, including virtual computer hardware platforms, storage devices, and computer network resources. That allowed to use shared resources with simultaneous operations of more than one Operating System (OS) in the same time. The further significant event was a creation of virtualised private networks by telecommunication companies [1], [2]. This offer to users shared access to a single physical infrastructure through a non-trusted network so that this communication appears to be as if the computers were in one private trusted network. That is usually secured using Internet Protocol Security (IPsec) protocol in tunnel mode (in this mode the entire packet is encrypted and validated. This is then encapsulated in a new Internet Protocol (IP) packet with a new IP header) [3], [2].

A. Cloud composition

The cloud is basically composed by service models, deployment models and key characteristics.

The three prominent service models of cloud computing are (see Def. 1): (i) Software as a Service (SaaS), (ii) Platform as a Service (PaaS) and (iii) Infrastructure as a Service (IaaS). The difference between these types is shown in Fig. 1. The diversity lies in options which are supported within the cloud. For each service model the options are highlighted by light blue color in Fig. 1.

The five essential characteristics of these service models are [5] (see Def. 1):

- On-demand self-service: (i) focuses on delivering IT services driven by user requests, (ii) no human interaction with the cloud provider, (iii) cloud computing provides a mean of delivering computing services that make the underlying technology, beyond the user device, almost invisible).
- Ubiquitous network access: (i) focuses on delivering IT services anytime, anywhere, and through user-chosen devices, (ii) users accessing services via Internet technologies expect a secure, "always-on" computing infrastructure that is delivered as easily and reliably as electricity from a wall outlet.
- Pool of virtualized resources: (i) focuses on delivering IT services through resource pools that can expand and contract based on the requirements of the underlying workload and the usage characteristics.

978-1-5386-9361-2/18/$31.00 ©2018 IEEE

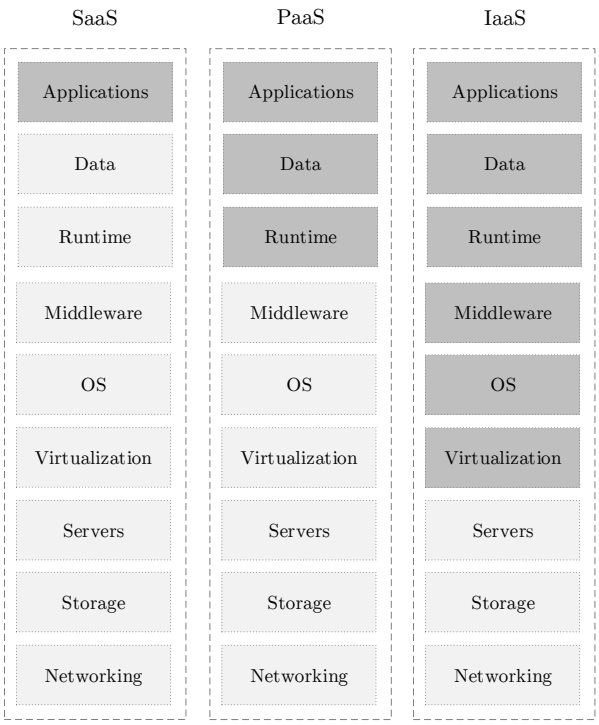

Fig. 1. Service models of cloud computing [4].

- Utility-based pricing: (i) focuses on delivering IT services that can be metered for usage and charged for (if needed) through pricing models including subscription, usage pricing – Service Level Agreements (SLAs).

These clouds represent four deployment types (as was mentioned in Def. 1): (i) public, (ii) private, (iii) hybrid and (iv) community.

The Gartner defines these deployment types as follows [6], [7]:

Definition 2. A public cloud is a publicly accessible cloud environment owned by a third-party cloud provider. The IT resources on public clouds are usually provisioned via the previously described cloud delivery models and are generally offered to cloud consumers at a cost or are commercialized via other avenues (such as an advertisement).

Definition 3. Private cloud computing is a form of cloud computing that is used by only one organization, or that ensures that an organization is completely isolated from others.

Definition 4. A hybrid cloud is a cloud environment comprised of two or more different cloud deployment models. For example, a cloud consumer may choose to deploy cloud services processing sensitive data to a private cloud and other, less sensitive cloud services to a public cloud.

Definition 5. A community cloud is similar to a public cloud except that its access is limited to a specific community of cloud consumers. The community cloud may be jointly owned by the community members or by a third-party cloud provider that provisions a public cloud with limited access. The members of the cloud consumers community typically share the responsibility for defining and evolving the community cloud

In this work, we provide the general solution for all these deployment types.

B. Main contribution

In this paper, we focused on setting the most effective architecture on the server side (e.g., returning only requested data, understandability and maintainability of SaaS solution) of the clouds and also design the appropriate communication between providers, clients and components on the server concerning efficiency, scalability, reusability and security. The prototypes of proposed and discussed architectures were implemented and tested. Following the results obtained from created prototypes, both advantages and disadvantages are highlighted in this paper.

Our proposed architecture is currently working solution, based on modified microservice architecture wrapped into one gateway, which was created based on our experiences from previous projects which were focused on working with web services, communication with Internet of Things (IoT) devices, especially the Narrowband IoT (NB-IoT) one etc. In our previous projects, we were in case of architectures like microservices disappointed that in case of getting project bigger we were not able to create meaningful Representational State Transfer (REST) endpoints and usability of SaaS solution was quite not user-friendly. For these reasons, we tried to figure out how to design our SaaS solution in the most effective way, because we need to communicate with different types of clients. In order to design the above-mentioned goal, we first answer the key questions: how the cloud is supposed to be used? Is the client always able to communicate using Hyper Text Transfer Protocol (HTTP) protocol? What are the given restrictions (budget, the required skills for developers and at the side of system administrators)? At that moment we can proceed and sum-up the main contributions of this work as follows:

- Design of the effective communication between client side and server side concerning security, data validation, expandability (adding new functionality/modules/interaction with other services) and especially for the clear usability of our SaaS solution.
- Trial implementation and verification of proposed architecture.

The remainder of this paper is structured as follows. In Section II we discuss different types of architectures which were used for SaaS cloud solutions, Section III details the importance of using suitable web services architecture or protocol for designing SaaS solutions. Section IV discusses our proposed and created architecture for our SaaS solution with respect to aspects mentioned above. Finally, concluding remarks together with lessons learned are summarized in Section V.

II. The State of the Art

In these days it is common to move all applications or services to clouds, e.g. Amazon Web Service, Microsoft Azure, IBM Cloud and Google Cloud. These clouds were created following different architectural approaches. In the 1980's and earlier the ecosystem of clouds was mostly vertical and in 1990's it moved to horizontal business-process-focused structures due to evolution in business organizations. They need to integrate/sell or continuously deliver different services. This need leads to migration from monolithic architecture to Service Oriented Architecture (SOA) and Microservices [8], [9], which are described in following sections.

A. Monolithic Architecture

The monolithic architecture combines (i) user interface, (ii) application logic, and (iii) data at once. In architecture such this, it is difficult for team members especially the new ones to understand and modify the code, once the application becomes large and the team grows in size. The additional drawbacks of this approach are [10], [11], [12]:

- Difficult continuous deployment: a large monolithic application is an obstacle to frequent deployments. In order to update one component or module, it is necessary to redeploy the entire application. This will interrupt background tasks, regardless of whether they are impacted by the change, and possibly cause problems. There is also a chance that components that have not been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers since they usually need to iterative rapidly and redeploy frequently.
- Scalability issues: this architecture is scalable only in one dimension (adding resources – CPU, RAM etc.), at some moment it becomes expensive to manage increasing data volume because powerful CPU and RAM is more expensive than just adding another computer [11], [12]. Also, different application components have different resource requirements – one might be CPU intensive while another might be memory intensive. With a monolithic architecture it is not possible to scale each component independently.
- Slows down development process: if the code base becomes too large the Integrated Development Environment (IDE) is slowed down, and developers are less productive.
- Impossible to sell application modules in separate: it is not possible to separate functionality in modules and sell to third-party only modules with requested functionality. In that case, users must pay for the whole application and cannot buy just a few modules which usually results in fact that this application is too expensive [10], [11], [12].

This leads to fact that there is no possible modularity, low level of code reusability, it is hard to test, and also there is no horizontal scalability in such architecture.

B. Layered Architecture

Is an architecture used in designing computer software or hardware where system or network is designed into separate layers so that changes could be done in a separate layer and do not affect others This architecture leads to separate modules that behave like they are in the same layer. The layered system is ideal in case that system has different levels of functionality. Particular function which is called by a user is actually performed in the lowest level or layer of a system. Usually, a user does not want to call this function in the lowest level, because they are too complex to use, they have several implementations etc. Common layers in this layered architectures are (from the bottom): (i) data access layer, (ii) business logic layer sometimes called service layer, (iii) facade layer, (iv) web service layer, these days usually implemented using REST architecture, and (v) graphical user interface layer [13], [14]. The benefit of this is that different layers could be distributed, implemented independently and easily replaceable with a different approach, e.g. developers decide to use REST instead of SOAP so they just replace the particular module.

Basically the layered architecture is (i) *One-tier architecture*, (ii) *Two-tier architecture* or *Three* (*and* +)-tier architecture: One-tier is the simplest one, in that case, all the necessary layers are located on the same machine. It is easy to design, but the least scalable. Pros of this architecture are: (i) little development maintenance required for client level, (ii) it is easy to update as cons we see server side is expensive and there is single-point-of-failure.

Two-tier architecture supplies basically network communication between client and server, e.g. browser makes a request to a server, the server processes the request and returns desired response (web page). These approaches improved scalability in the way of dividing the user view from data layers. Pros of this architecture are: (i) better load sharing, (ii) some work is done at a client, (iii) presentation can be customized per client and as cons we see tight coupling and client needs to be developed/deployed. And the last one is three+ tier architecture, which is commonly used in web apps development. This architecture separates server layers basically into three or more layers as: (i) presentation, (ii) application logic, (iii) data layer (resource management). Pros of this architecture is that server layers can be split and also easily replaced with another approach, e.g. replacing Object Relation Mapping (ORM) with lighter solution as cons we see: (i) it is complex, (ii) sometimes problematic interconnectivity between layers on different nodes (serialization and deserialization of data messages) [13], [14]. These tiered architectures are shown in Fig. 2.

C. Service Oriented Architecture and Microservices

SOA is architecture proposed by International Business Machines (IBM) company. This architecture is oriented on services, it introduces concept of accessing information systems. In SOA, the terms like information system, software or application are connected with a set of services that are accessible as single instance. In general implementation of

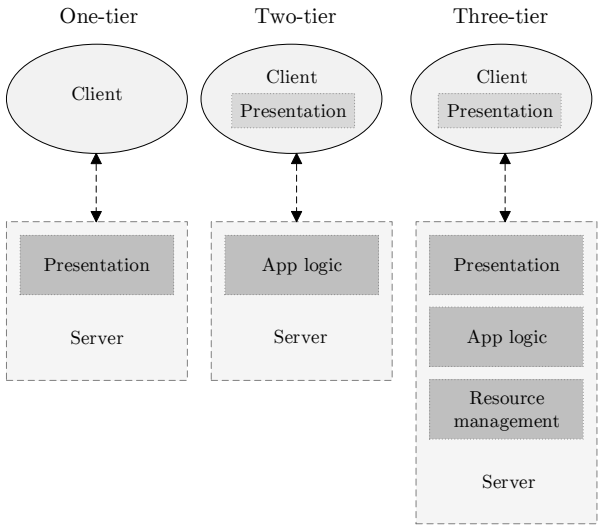

Fig. 2. Structure of One/Two and Three+ tiered Architectures [15].

SOA it is usually done using web services based on Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP) protocol [16], [8], [9], [17].

Microservices is sometimes called as one of possible implementations of SOA architecture. Instead of standard SOA implementation using WSDL and SOAP, it uses REST architecture for designing particular services [18], [19].

III. The Role of Web Services in SaaS Clouds

Every cloud solution must be accessible for clients. The most common ones use SOAP protocol, REST architecture, GraphQL or in case of low level devices protocols as MQ Telemetry Transport (MQTT) and Constrained Application Protocol (CoAP). The choice of particular communication architecture or protocol significantly affects designing the whole architecture in terms of security, performance and scalability. This is the reason why they are briefly described in this section as the combination of module design and communication protocols creates the architecture itself.

A. REST API

Definition 6. REST is an architectural style that defines a set of constraints and properties based on HTTP protocol [20].

The important one is that Representational State Transfer Application Programming Interface (REST API) is architecture not protocol as, e.g. SOAP. Because that architecture is nowdays well-known, we depict only few suggestions (i) prefer communication using JavaScript Object Notation (JSON) over Extensible Markup Language (XML), (ii) use plural instead of singular in terms of design Application Programming Interface (API) endpoints, (iii) use nouns, not verbs, (iv) provide ability to filter, sort, retrieve, and paging for data collections, (v) set versioning for each API, (vi) HTTP methods PUT, POST and PATCH should return resource representation, (vii) use sub-resources for relations [21], [22], [23], [24], [25].

B. SOAP

SOAP is protocol for exchange messages based on XML format, mostly over HTTP protocol, but allows also other protocols as Simple Mail Transfer Protocol (SMTP). The main advantage of this protocol over REST architecture is that allows encryption on particular messages and possibility to generate client-side based on WSDL document. The main disadvantage of this protocol is that it is too complex for developers in comparison with REST, because users usually need Create Read Update Delete (CRUD) operations only without difficult figuring out how to use this. Also, the support for programming languages is not so wide as in case of REST [26], [27].

C. GraphQL

GraphQL is open source data query language proposed by Facebook in 2015 to a public audience. It provides the efficient way of querying data and seems to be like a great alternative to REST. The main benefit comparing to the REST architecture is that GraphQL has a single endpoint and returns flexible data structures while REST has a set of endpoints that each return fixed data structures. With GraphQL, it is natural that we define what fields should be returned by the API while using REST this practice is not very common. From the facts listed above, we could currently depict only one issue, that GraphQL is still not supported by many tools, but that should be a matter of time [28].

D. IoT Protocols for Low-Level Communication

Currently, standard IoT protocols are MQTT and CoAP. These protocols are suitable in low-level communication in terms of sending and receiving raw bytes [29], [30].

IV. Proposed SaaS Architecture

The previous sections discussed in detail selected software architectures in terms of efficiency, scalability, reusability, testability, isolation and abstraction. Based on these aspects we propose the new way of design software architecture. This architecture is using benefits from different kind of architectures. Quite a long description of different architectures and protocols in this paper was necessary to fill the whole picture about the advantages and disadvantages of each approach and to fully depict proposed solution.

The proposed solution is using the further modification of microservice architecture in terms of wrapping it to one Gateway using GraphQL. This architecture is also n-tier, because each module in the SaaS solution is n-tier into several layers: (i) data layer is for managing data in terms of querying database, (ii) service layer is used for business logic (computing requested service) and also for calling remote REST APIs for using additional services if necessary or another modules in the system, e.g. if module 1 in the system needs something from module 2 in the system then a communication

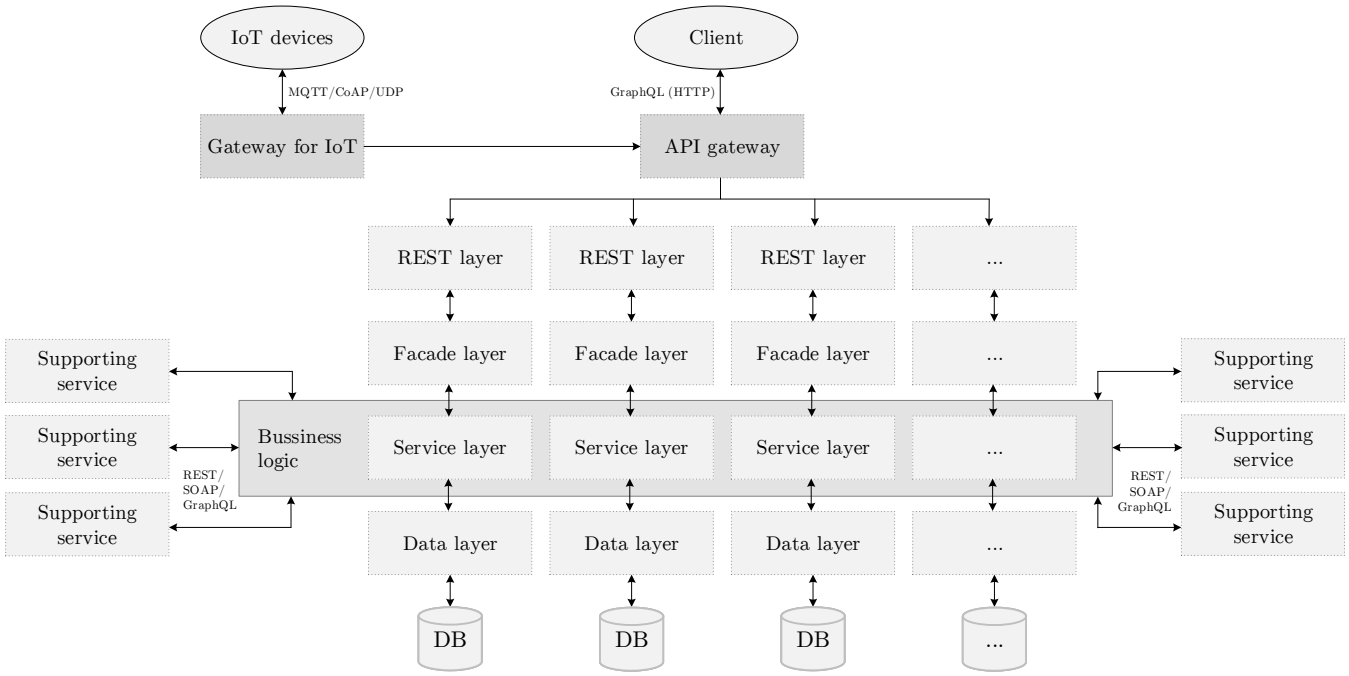

Fig. 3. Proposed Architecture using API Gateway with GraphQL query language.

is provided using REST layer. This approach has several benefits and several drawbacks. As the benefit, we could see it is language independent and each action on API is documented so it is easy to use it. For scalability of the intended application and also easy to use approach, it is suitable to prefer calling REST API instead of directly using service. The reason is easy if we call another service layer directly we need to parse results to another language and in some cases we could not fully use benefits of that service and also we could achieve many circular dependencies between two components of the system. As a drawback, we could see that it needs additional serialization and deserialization.

Schema of the proposed architecture is shown in Fig. 3. In this schema, we see that API Gateway access point is using GraphQL query language instead of REST, but each service has its own REST layer, that in general means that services could be easily developed separately. In the phase of deploying this service, it will just need to be connected to the API Gateway.

In general main benefits of this architecture are:

- Scalability: adding functionality to the system is quite easy, just with adding a new module and connecting it to the API Gateway calls.
- Testability: increase testability since each module is designed into several layers (it is easier to test a specific module than to test the whole system at once).
- Reusability: leads to an increase of Return on Investment (ROI) (pay off in the future).
- Isolation: allows isolating technology upgrades to individual layers in order to reduce risk and minimize the impact on the overall system.
- Abstraction: allows to make changes in implementation without changing the abstraction layer
- Language independence: system could be easily developed using any number of programming languages, cooperation between them is straightforward and well documented.
- Reduce costs for development and integration: modules could be developed in parallel, the maintenance is easier, because each team is responsible for a particular module or even particular layer in a specific module.
- Reduce number of endpoints (compared to microservices).

Disadvantages of proposed architecture are (i) architecture is complex, developers need to learn different technologies, (ii) performance is decreased during the call of web services and (iii) additional common issues on distributed systems computing as solving node failures.

The most common implementation approach in SOA architecture supposing for communication using SOAP protocol which is these days for most scenarios quite overkill, e.g. IoT devices prefer lighter communication protocols than heavy protocols as is SOAP. Also SOA does not provide to communicate with one central node as our solution provides (API Gateway communication using GraphQL), additional drawback of using SOA is that it proposes XML as communication format. In contrast our proposed architecture supposes using as main format JSON and optionaly XML.

In comparison with microservices, our proposed architecture provides greater user experience from this application by using only one endpoint (many endpoints result in the API being unclear), which is managing the whole server-side processing.

A user is just defining in that endpoint what he would like to get (using GraphQL) from the system and system will manage it. Additional advantages are based on benefits of GraphQL over REST API, e.g. versioning: GraphQL takes a strong opinion on avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema. In REST architecture when API developers need to perform versioning the API, because there is limited control over the data that's returned from API endpoint, any change can be considered a breaking change, and breaking changes require a new version. If adding new features to an API requires a new version, then a trade-off emerges between releasing often and having many incremental versions versus the understandability and maintainability of the API. In contrast, GraphQL only returns the data that is explicitly requested, so new capabilities can be added using new types and new fields on those types without creating a breaking change. This has lead to a common practice of always avoiding breaking changes and serving a version less API [31].

V. Conclusion

In the paper, we focused on finding suitable architecture for current SaaS cloud solutions. While current modern SaaS solutions are usually following n-tiered, SOA or microservices architecture, we propose another way of designing SaaS solutions using API Gateway with GraphQL query language. In the perspective of this architecture, it is not focused too much on performance, but especially on scalability, testability, abstraction, isolation, language independence, development cost reduction, effectiveness, and reusability.

However, we shown that each architecture of cloud solution is not perfect, some of them have high performance of operations, but are not scalable (Monolithic architecture), some of them are much more scalable (microservices), but performance is downgraded by serialization and deserialization of data messages. This fact of decreasing performance over improved scalability is popular, because it allows continuous delivery of software solutions.

Acknowledgment

The described research was supported by the National Sustainability Program under grant LO1401. For the research, infrastructure of the SIX Center was used.

References

[1] R. Namboori, "The Evolution of Cloud Computing." https://dzone.com/articles/cloud-computing-1. Accessed: 2018.

[2] A. Agarwal, S. Siddharth, and P. Bansal, "Evolution of cloud computing and related security concerns," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–9, March 2016.

[3] D. G. Kogias, M. G. Xevgenis, and C. Z. Patrikakis, "Cloud federation and the evolution of cloud computing," *Computer*, vol. 49, pp. 96–99, Nov 2016.

[4] "Cloud computing - saas, paas and iaas models." http://saphanatutorial.com/cloud-computing-saas-paas-and-iaas-models/. Accessed: 2018.

[5] E. Schouden, "Cloud computing defined: Characteristics and service levels." https://www.ibm.com/blogs/cloud-computing/2014/01/31/cloud-computing-defined-characteristics-service-levels/. Accessed: 2018.

[6] "IT Glossary." https://www.gartner.com/it-glossary/. Accessed: 2018.

[7] A. Sridhar, *Big Data Analytics with Hadoop 3*. Packt Publishing, 2018.

[8] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: Service Oriented Architecture And Web Services*. IBM, 2004.

[9] IBM, "IBM Service-Oriented Architecture Services." https://www.ibm.com/services/us/igs/pdf/g565-1226-soa-services.pdf. Accessed: 2018.

[10] R. Machado, *Monolithic Architecture (Architecture and Design)*. Prestel Pub., 1995.

[11] C. Richardson, "Pattern: Monolithic Architecture." http://microservices.io/patterns/monolithic.html. Accessed: 2017.

[12] A. Gaurav, *Building Microservices with .NET Core 2.0 – Second Edition: Transitioning monolithic architectures using microservices with .NET Core 2.0 using C-sharp 7.0*. Packt Publishing, 2017.

[13] P. Sheriff, *Fundamentals of N-Tier Architecture*. PDSA, Inc., 2006.

[14] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley., 2003.

[15] S. Rajmukar, "Software architecture: One-tier, two-tier, three tier, n tier." https://www.softwaretestingmaterial.com/software-architecture/. Accessed: 2018.

[16] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design (Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall., 2005.

[17] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices)*. Prentice Hall., 2004.

[18] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O Reilly Media., 2015.

[19] I. Nadareishvili, R. Mitra, M. Mclarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O Reilly Media, 2017.

[20] D. Linthicum, *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*. Addison-Wesley Professional, 2017.

[21] P. Sturgeon and L. Bohill, *Build APIs You Won't Hate: Everyone and their dog wants an API, so you should probably learn how to build them*. Philip J. Sturgeon, 2015.

[22] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013.

[23] S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. Yahoo Press, 2010.

[24] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[25] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 2011.

[26] C. Jinfu, W. Huanhuan, and D. Towey, "Worst-input mutation approach to web services vulnerability testing based on soap messages," *Tsinghua Science and Technology*, vol. 19, no. 5, pp. 429–441, 2017.

[27] S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamara, P. Fremantle, D. Koenig, and C. Zentner, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Sams Publishing, 2004.

[28] B. Samer, *Learning GraphQL and Relay*. Packt Publishing, 2016.

[29] P. Masek, J. Hosek, K. Zeman, M. Stusek, D. Kovac, P. Cika, J. Masek, S. Andreev, and F. Kropfl, "Implementation of true iot vision: Survey on enabling protocols and hands-on experience," *International Journal of Distributed Sensor Networks*, vol. 4, no. 3, pp. 1–18, 2016.

[30] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar, "Secure mqtt for internet of things (iot)," in *2015 Fifth International Conference on Communication Systems and Network Technologies*, pp. 746–751, April 2015.

[31] "GraphQL Best Practices." https://graphql.org/learn/best-practices/. Accessed: 2018.