

Best Practice in the Implementation of Restful Web Services

Salih Osman

What is REST?

Representational state transfer (REST) is a term used to describe an architectural style designed by Roy Fielding and published in his 2000 doctoral thesis “Architectural Styles and the Design of Network-based Software Architectures” in an attempt to, as he put it “represent the model for how the modern Web should work” [1]. The architectural style set out in this thesis delineated, was and delineated by, eight key principles, which contained the six architectural constraints of REST. These constraints were summarised by the terms: Client-server; stateless; cache; uniform interface; layered system and code on demand. In order for a system to be described as RESTful, it must comply with each of these constraints, excluding the final one. This was set down as optional by Fielding due to the potential negative impact on visibility, which he defined by how capable a particular component was of overseeing or arbitrating the interaction of 2 others [1].

These constraints, when applied as a whole, can increase the ability of different components of the system to communicate and interact with one another, whilst improving the flexibility of the system and increasing ease of maintenance. Conversely, however, they can lead to a decrease in the performance and reliability of the system as a whole, the adoption of REST is, as with any software style or pattern, the adoption of certain benefits at the expense of other attribute [2].

Client-Server Architecture

One of the most fundamental constraints of rest is that the client side code and the server side code must be kept independent of each other. In other words, the client has no impact on, for example, the way data is stored by the server, whilst the server has no input into the way in which the data is displayed front end [3]. The client communicates with the server by sending requests and then parsing and interpreting responses, whilst the server waits for requests before sending off responses in a set format. This allows both the client side and the server side to be scaled and even significantly modified independently of each other, without facing compatibility issues. This is because the server exposes the API, which the client then consumes, in accordance with the principle of separation of concerns, so long as the interface is not altered, the client and server sides will be unaffected by changes in the other side.

```

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer","handler"})
public class Buyer implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int buyerid;

    public Buyer() {}

    private String name;

    public int getBuyerid() {
        return buyerid;
    }

    public void setBuyerid(int buyerid) { this.buyerid = buyerid; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public List<Batch> getBuyer() { return buyer; }

    public void setBuyer(List<Batch> buyer) { this.buyer = buyer; }

    @OneToMany(fetch = FetchType.LAZY, targetEntity = Batch.class, mappedBy = "buyer")
    private List<Batch> buyer;
}

```

Figure 1: The Buyer entity, server side

```

function checkName(username) {
    $.ajax({ settings: {
        type: "GET",
        url: "/buyer",
        contentType: "application/json",
        dataType: "json"
    }}).then( doneFilter: function (buyerResponse) {
        var nameNotFound = true;

        for (let i = 0; i < buyerResponse._embedded.buyer.length; i++) {
            if (buyerResponse._embedded.buyer[i].name === username) {
                window.location.href = '/auction/' + buyerResponse._embedded.buyer[i].buyerid;
                nameNotFound = false;
            }
        }

        if(nameNotFound) {
            window.alert("That username doesn't exist");
        }
    })
}

```

Figure 2: Client side request to buver endpoint

Within the vehicle registration plate project, this REST constraint was upheld, as shown in Figures 1 and 2. The client, written in javascript, then consumes the API, making a request to the endpoint containing the buyer data. It then parses through the json objects returned from this to decide what to return on screen. No html or commands of any kind were taken from server side, rather all decisions pertaining to the UI were made client side on the basis of communication via the API.

Statelessness in REST

The REST stateless constraint basically implies that the server cannot contain any knowledge about the state of the client. Rather, every request the client sends to the server needs to include all the data required for the request to be correctly interpreted, as none can be left server side in between requests [4]. One implication of this is that there can be no such thing as server side sessions in a REST service. All information relating to a session must be kept client side.

The aim behind this is to increase the visibility of the system, as observing any individual request will reveal everything necessary for that request without the need for further examination seeking some client context. Reliability is also positively impacted, as the lack of server side input simplifies failure analysis [5].

This was one of the principles enforced in the vehicle registration plate project. As shown in Figure 2, in order to implement a pseudo-session, the buyer ID was taken from the GET request response and maintained client side by acting as a form of token as part of the URL of the auction page. Thus, if session specific information or context was required for one of the requests, for example in a possible expansion of the project it is proposed that the previous purchases of the logged in client, then the client id could be accessed from the URL and inserted into the request so as to provide a session specific response. Another possible expansion centred around the concept of securing sessions with passwords and authentication, it would be possible to persist passwords in the database, perhaps after being encrypted via some sort of HashMap. Fully authenticated login could then occur, still with no context being stored in the server, as shown in Figure 1 and with subsequent session specific requests still maintaining their standalone nature as shown in Figure 3.

```
function buyerPUT (batchid) {  
  $.ajax({  
    type: "PUT",  
    contentType: "application/json",  
    url: '/batch/'+batchid+"/buyer",  
    data: JSON.stringify( value: { _links:{self:{href:"http://localhost:8080/buyer/"  
+window.location.pathname.substr( from: 9) }}}),  
    success:location.reload(),  
    dataType: "json"  
  })  
}
```

Figure 3: Client side session specific PUT request

Cache

This constraint requires that the responses returned from server side must be able to be identified as either cacheable or non-cacheable. This allows individual clients to be “re-served” from the cache of data on their own machine, improving the speed of the web service from their perspective [6]. This was not implemented within the body of the code, rather, in order to avoid caching of any kind (essentially the equivalent of labelling every response as non-cacheable) the web browser used to access the service was used in incognito mode. This was to prevent any data being cached automatically, as is common in Chrome, and later yielding invalid results due to it no being out of date.

Uniform Interface

The presence of this constraint within REST is one of the things that differentiates it from other architectural styles[1]. The uniform interface constraint was introduced by Fielding as being further subdivided into 4 separate interface constraints. These are: “identification of resources; manipulation of resources through representations; self-descriptive messages; and hypermedia as the engine of application state” [1].

Resource identification basically implies that each discrete entity of the system, a resource, such as, for example, a license plate, batch of license plates, or a user in the system, must have its own unique resource identifier (URI), a link which points to it [7]. This enables a particular data object to be viewed and accessed according to that link as shown in Figure 4. It is possible to identify and access every

```
let batchid = $(html: '#batchNumber'+cardNumber).text();
$.ajax( settings: {

    type: "GET",

    url: "/batch/"+batchid,

    contentType: "application/json",

    dataType: "json"
```

Figure 4: Resource identification is possible via the URI

single batch through use of the “/batch/i” URI where i represents the unique ID of any given batch.

Manipulation of resources via representations can be thought of as closely related to the resource identification constraint. However, it is perhaps more abstract, as it relates to the idea of a resource being purely a purely conceptual entity of which different representations are passed around but not the resource itself [7]. The basis of the constraint is that having access to a representation of a resource is sufficient to implement a change on that resource itself. This can be seen in Figure 5, where it can

```
$.ajax({
    type: "PATCH",
    contentType: "application/json",
    url: '/batch/'+batchid,
    data: JSON.stringify( value: {price:bidAmount} ),
    success: buyerPUT(batchid),
    dataType: "json"
})
```

Figure 5: Batch resource updated via JSON representation

be seen that despite the client only having access to a JSON representation of the resource via its URI, an update can still be made which will modify the resource as a whole.

The self-descriptive messages constraint means that every request which passes from client side to server side, and every response which passes in the other direction must contain sufficient information that the message can be understood and interpreted. In other words, the “datatype”: “JSON” element of the PATCH request in Figure 5 can be thought of as fulfilling this constraint [8].

Hypermedia As The Engine Of Application State (HATEOAS) is another distinctive element of REST services in particular. The fundamental concept behind this constraint is that the client is essentially

directed, or conducted, along the application, down a series of valid paths and endpoints [9]. For example, having started at a basic URI, the client should then be given access to the URIs which represent a finer resolution view of the constituents of the initial resource (which are themselves resources). In this way it is possible to traverse each of the different endpoints related to the initial resource, with only relevant links being displayed to the client. An example of the use of HATEOAS from the license plate project, can be seen in figure 4. As shown, the PUT request is being made to the HATEOAS buyer link from the “/batch/i” endpoint. Furthermore, the data being sent in is not a variable, such as ID marker or object, but rather, the URI string corresponding to the buyer resource to which that particular batch will now be attributed. This is the essence of HATEOAS, being able to progress from a single root resource to all other elements of the system in a logical and understandable manner by navigating through the series of subsequently revealed hyperlinks [10].

Layered system

The basis of the layered system constraint is that the introduction of additional layers of abstraction between the individual components of the system will help insulate them from, for example, updates or alterations in the other components. Although there was no manually implemented layering between the client and the server, which could have been used to alter the requests and responses passing from client to server and back, the license plate project did use the repository pattern in conjunction with hibernate ORM and PostgreSQL database. Thus, requests pass from the client, through the server, then via the repository, through the ORM to the database. The use of the repository pattern can be seen in figure 6.

```
import io.swagger.annotations.Api;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@Api(tags = "Address Entity")
@RepositoryRestResource(collectionResourceRel = "numberplate", path = "numberplate")
public interface NumberplateRepository extends PagingAndSortingRepository<Numberplate, Integer> {
}
```

Figure 6: Example of repository class acting as additional layer of abstraction

Ultimately, it is possible that the license plate application cannot truly be called RESTful, at the very least because it did not implement caching in accordance with the REST constraints. However, it can certainly be said that the application benefited from the implementation of numerous REST-based techniques.

Bibliography

- [1] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000.
- [2] B. Costa, “Evaluating REST architectures—Approach, tooling and guidelines,” *J. Syst. Softw.*, vol. 112, pp. 156–181, Feb. 2016.
- [3] D. Salskyi, A. Kozhukhar, O. Olshevska, and N. Povarova, “OVERVIEW OF POPULAR APPROACHES IN CREATING CLIENT-SERVER APPLICATIONS BASED ON SCIENTOMETRICS ONAFTS’ PLATFORM,” *Avtom. Tehnol. i Biznes-Processov*, vol. 9, no. 4, 2017.

- [4] B. Mehta, *RESTful java patterns and best practices : learn best practices to efficiently build scalable, reliable, and maintainable high performance RESTful services*. .
- [5] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A Note on Distributed Computing," *IEEE MICRO*, vol. 17, pp. 49--64, 1994.
- [6] J. R. Erenkrantz, "Computational REST : a new model for decentralized, internet-scale applications," University of California, Irvine, 2009.
- [7] R. Verborgh, S. van Hooland, A. S. Cope, S. Chan, E. Mannens, and R. Van de Walle, "The fallacy of the multi-API culture," *J. Doc.*, vol. 71, no. 2, pp. 233–252, Mar. 2015.
- [8] Xinyang Feng, Jianjing Shen, and Ying Fan, "REST: An alternative to RPC for Web services architecture," in *2009 First International Conference on Future Information Networks*, 2009, pp. 7–10.
- [9] S. Kennedy, R. Stewart, P. Jacob, and O. Molloy, "StoRHm: a protocol adapter for mapping SOAP based Web Services to RESTful HTTP format," *Electron. Commer. Res.*, vol. 11, no. 3, pp. 245–269, Sep. 2011.
- [10] F. Haupt, F. Leymann, and K. Vukojevic-Haupt, "API governance support through the structural analysis of REST APIs," *Comput. Sci. - Res. Dev.*, vol. 33, no. 3–4, pp. 291–303, Aug. 2018.

Serverless Technologies and the Circuit Breaker Pattern

Salih Osman

What does Serverless mean?

The term “serverless” is relatively new, even within the field of software engineering, having appeared in academic usage around the year 2011 [1]. A common contention on the subject of serverless technologies is that the term itself is misleading, as it implies an architecture or system which completely does away with servers. Naturally, this is not the case, rather, serverless implies that the very much present and real servers are essentially hidden away from the developers. In other words, the management and maintenance of the physical servers on which the web service is hosted, is taken care of by the company providing the cloud platform. Because this all happens dynamically behind the scenes, it allows the developers to write code without having to think or worry about the complexities of provisioning or managing these servers. The term itself is somewhat difficult to pin down, as with many recent computing related buzzwords, having undergone a certain degree of evolution since its emergence. One of the less technically accurate usages would associate the term with the use of abstractions such as containers and microservices [2]. However, a more fitting definition must perforce include reference to the relevance or, as some would maintain, the equivalence of the concept of FaaS (Function as a Service) [3].

FaaS, PaaS and SaaS

The Standard Performance Evaluation Corporation (the SPEC) cloud group, seeking to provide clarification within the cloud computing lexicon, defines the term serverless (interchangeably with FaaS) on the basis of three criteria: “Granular billing”; “(Almost) no operational logic”; and “Event-Driven interactions” [4]. In other words, a serverless application only incurs cost during the time when it is actively being run. The operational logic, as mentioned, is nearly entirely taken care of by the cloud provider, meaning that it becomes their responsibility to, for example, scale the application by allocating more resources on an as and when basis. All other management of this sort is dealt with in this fashion, meaning that deployment, from the perspective of the developer, becomes about simply uploading code to the cloud provider. Lastly, the intention behind serverless applications is that their interactions are kept brief and centred around events, without having continuously running applications.

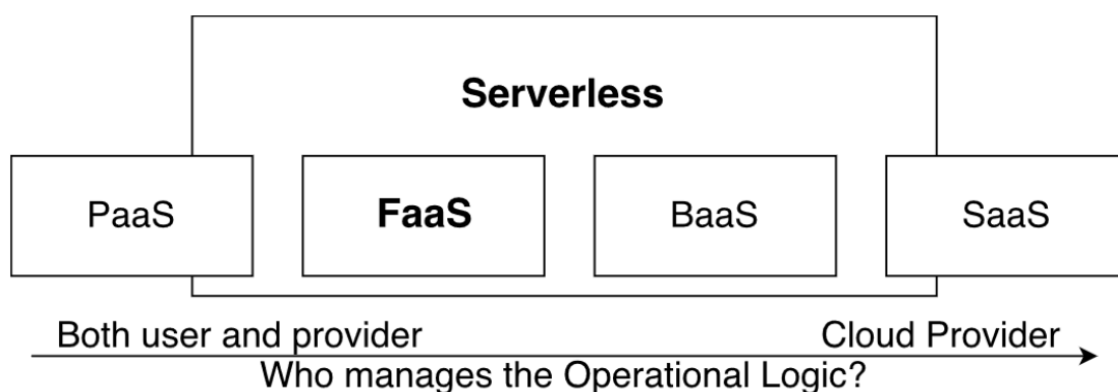


Figure 1: The orientation of PaaS and SaaS in relation to FaaS [4]

As shown in figure 1, the SPEC draws the comparison between serverless (FaaS) architecture and the related cloud computing services: PaaS (Platform as a Service) and SaaS (Software as a Service). These terms can be well described in accordance with the definitions laid out for them in the National Institute of Standards and Technology (NIST) “Definition of Cloud Computing” [5]. In short, PaaS allows users to deploy to a cloud provider, who deal with the management and provisioning etc. of the physical servers on which the code is running, in a manner similar to FaaS. However, unlike FaaS, what

is being deployed is not an individual “function” or fragment of an application but rather the application itself, for example as a JAR on something like AWS elastic beanstalk, which will be constantly running on the cloud provider’s platform, rather than being run and then terminated in an event-driven fashion. SaaS, on the other hand, means providing access to a software package over the cloud, for example Netflix or Office 365. In that instance, of course, essentially all the management of the application is performed by the cloud provider, with the user simply interacting on an end consumer basis [6]. Ultimately, while both PaaS and SaaS somewhat overlap with the sphere of serverless services, they are not suited to the type of event based, server system and long-lived application abstraction characterised by FaaS, meaning that the term “serverless” could rightly be said to inhabit the space between PaaS and SaaS [4].

Pros and Cons of Serverless (FaaS)

The advantages of serverless technologies when it comes to web hosting, can be approximately summed up under the following categories: cost effectiveness due to user only paying for the computing power used by their functions, increased ease of operation and a streamlined deployment process [7]. Increased cost effectivity can be quite clearly understood as a simple consequence of the user no longer having to pay for constant maintenance and running of their own server system, or even continuous running of their application. Instead the user need only pay when their one of their functions is actually being run, a process which can elapse over the course of a few fractions of a second [8]. Naturally, the cloud provider is able to support this significant reduction in cost because they are providing servers large-scale to multiple organisations, analogous to the way a large supermarket or wholesaler can offer more cost effective prices than the average corner shop. The increased ease of operation and deployment can be attributed to the fact that developers need only concern themselves with the writing of individual functions, with all the server, infrastructure and support concerns having essentially been abstracted away. The fact that the cloud provider is responsible for allocating extra computing power if necessary means that serverless hosting is intrinsically scalable, with developers being liberated from concerns of server capacity.

There are some drawbacks to the use of serverless technologies, which should be taken into account before the adoption of such a system. These drawbacks include factors such as security, a natural implication of the application being hosted on the cloud, with fragments of multiple application being run on the same servers, there is the potential for data to be unintentionally exposed [9]. There is also the possibility of problems or unpredictability due to aspects such as the need to start up the code for a particular function every time it is needed, as opposed to it being continuously running [10]. It should also be borne in mind that, in order to effectively design towards a FaaS system, developers may need to accept a new “mental model” centred around this new structure. This, in combination with the difficulties of moving to a different cloud provider after adopting serverless, known as “vendor lock-in”, means that much careful consideration is required before simply “jumping in” to the serverless pool [7].

What is the circuit breaker pattern?

The term “circuit breaker” refers to an electronic component which is essentially a cross between a switch and a fuse. Like a fuse, it is used to protect the rest of the circuit from high currents. However, rather than melting or breaking itself when too high of a current goes through it, a circuit breaker trips from closed to open, “breaking” the circuit. It can then be re-closed, like a switch, to re-establish the circuit once the fault has been addressed.

As may be expected, then, the circuit breaker pattern fulfils an almost exactly analogous purpose within the world of web development. Proliferated by Michael T. Nygard in 2007, this design pattern aims to prevent the possibility of cascading failure within a software system [11]. One of the inherent consequences of cloud hosting is a higher incidence of both short term and long term failures [12]. In the situation where a particular service or microservice has failed, the calls, requests or other operations enacted upon it by related services will also fail. Rather than allowing these services to continue making doomed requests until timeout, potentially failing themselves, the circuit breaker pattern prevents them from trying to make contact with the downed service after a certain number of attempts [12]. As the circuit breaker is now quick-failing any requests rather than actually allowing them to be made, the other elements of the system are no longer waiting for the unresponsive service, minimising the negative impact on the system as a whole, for the duration of the failure.

How does it work?

The circuit breaker pattern can basically be thought of as encapsulating a given function call. It might not do anything the first time that call fails but it will record that failure, along with any subsequent ones. Having been scheduled to activate after a given threshold of failures have been recorded, or after a specific fault has occurred, the circuit breaker will “trip open” as shown in figure 2 [13]. Once

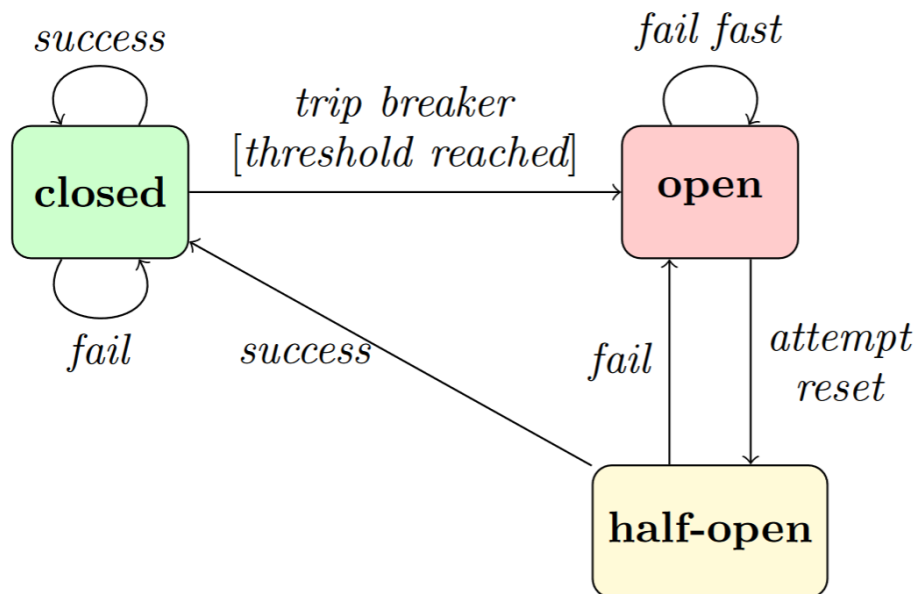


Figure 2: Representational diagram of circuit breaker pattern [13]

open, requests to the failed service are immediately responded to with a failure message, rather than being forwarded to the service. Unlike a real circuit breaker, the circuit breaker pattern has 3 states, as opposed to being a binary system of open or closed. After either a predefined length of time, or after attempting to establish contact with the failed service, the circuit breaker will enter the half open state. In this situation, the circuit breaker will pass on a few requests to the target. On the condition that each of these calls yield a successful response, the circuit breaker will once again return to the closed state, if not it will return to the open state, after which it will again go through the process of either waiting, or pinging the service until it can return to the half open state.

What is the significance of the circuit breaker pattern?

As discussed in previous sections, the diversified use of cloud services is a growing modern phenomenon, with microservice architecture and function as a service systems becoming increasingly

common and utilised. Because of this, it is even more crucial that the principle of separation of concerns is upheld. As mentioned by Balalaie, Heydarnoori and Jamshidi, the decreased fail times due to use of the circuit breaker pattern can facilitate independent service deployments [14]. Furthermore, the use of the circuit breaker pattern leads to an increase in the overall “resilience” of the system, by reducing the unnecessary load on the system and preventing failure from propagating throughout numerous different components of the system [15]. This has been the reason for the recommendation of the circuit breaker pattern in such systems as the internet of things [16] and the use of a service side, rather than simply client side circuit breaker patterns, as laid out by Montesi and Weber [13]. Ultimately, whilst it is impossible to indefinitely avoid the failure of one or other of the microservices within a system, the circuit breaker pattern provides a technique for minimising the impact of that unavoidable failure [13].

Bibliography

- [1] J. Dean, A. Harrison, R. N. Lass, J. Macker, D. Millar, and I. Taylor, “Client/server messaging protocols in serverless environments,” *J. Netw. Comput. Appl.*, vol. 34, no. 4, pp. 1366–1379, 2011.
- [2] S. Burns, “SERVERLESS COMPUTING: WHAT IT IS AND WHY IT MATTERS: Enterprises might be forgiven for thinking serverless computing is just the latest in a long line of hyped technology concepts, but there is serious substance to it.,” *Comput. Wkly.*, pp. 21–24, May 2018.
- [3] G. Hohpe, I. Ozkaya, U. Zdun, and O. Zimmermann, “The Software Architect’s Role in the Digital Age,” *IEEE Softw.*, vol. 33, no. 6, pp. 30–39, Nov. 2016.
- [4] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, “The SPEC cloud group’s research vision on FaaS and serverless architectures,” in *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC ’17*, 2017, pp. 1–4.
- [5] P. M. Mell and T. Grance, “The NIST definition of cloud computing,” Gaithersburg, MD, 2011.
- [6] R. Vidhyalakshmi and V. Kumar, “CORE framework for evaluating the reliability of SaaS products,” *Futur. Gener. Comput. Syst.*, vol. 72, pp. 23–36, 2017.
- [7] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-method empirical study of Function-as-a-Service software development in industrial practice,” *J. Syst. Softw.*, vol. 149, pp. 340–359, 2019.
- [8] D. Linthicum, “The pros and cons of serverless computing,” *InfoWorld.com*, 2017.
- [9] M. Attaran and J. Woods, “Cloud Computing Technology: A Viable Option for Small and Medium-Sized Businesses,” *J. Strateg. Innov. Sustain.*, vol. 13, no. 2, pp. 94–106, 2018.
- [10] M. Malawski, K. Figiela, A. Gajek, and A. Zima, “Benchmarking Heterogeneous Cloud Functions,” 2018, pp. 415–426.
- [11] M. T. Nygard, *Release it! : design and deploy production-ready software. .*
- [12] V. Malcher, “Design Patterns in Cloud Computing,” in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015, pp. 32–35.
- [13] F. Montesi and J. Weber, “Circuit Breakers, Discovery, and API Gateways in Microservices,” Sep. 2016.

- [14] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May 2016.
- [15] R. Ranchal, A. Mohindra, J. G. Manweiler, and B. Bhargava, "RADical Strategies for Engineering Web-Scale Cloud Solutions," *IEEE Cloud Comput.*, vol. 2, no. 5, pp. 20–29, Sep. 2015.
- [16] B. Butzin, F. Golasowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–6.