FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing a business platform using microservices.

Rajendra Kharbuja

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing a business platform using microservices.

Entwerfen einer Business-Plattform mit microservices.

Author:	Rajendra Kharbuja
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Manoj Mahabaleshwar
Submission Date:	

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Rajendra Kharbuja

Acknowledgments

Abstract

Contents

1 Challanges of Microservices Architecture

1.1 Introduction

The section **??** lists some drawbacks of monolithic architecture and these have become the motivation for adopting microservices architecture. Microservices offer opportunities in various aspects however it can also be tricky to utilize them properly. It do come with few challanges. The response from interview **??** also highlights some prominent challanges. In this chapter, the challanges of microservices alongside its advantages will be discussed. [Fow15]

Advantages

Strong Modular Boundaries

It is not totally true that monolith have weaker modular structure than microservices but it is also not false to say that as the system gets bigger, it is very easy for monolith to turn in to a big ball of mud. However, it is very difficult to do the same with microservices. Each microservice is a cohesive unit with full control upon its business entities. The only way to access its data is through its API.

Independent Deployment

Due the nature of microservices being autonomous components, each microservice can be deployed independently. Deployment of microservices is thus easy compared to monolith application where a small change needs the whole system to be deployed.[New15]

Challanges

Distributed System

The infrastructure of microservices is distributed, which brings many complications alongside, as listed by 8 fallacies.[Fac14] The calls are remote which are imminent to accomplish business goals. The remote calls are slower than local and affect performance in a great deal. Additionally, network is not reliable which makes it challanging to accept and handle the failures.

Integration

It is challanging to prevent breaking other microservices when deploying a service. Similarly, as each microservice has its own data, the collaboration among microservices and sharing of data can be complex.

Agile	Operational Complexity
Each microservice is focused to single responsibility, changes are easy to implement. At the same time, as the microservices are autonomous, they can be deployed independently, making the release cycle time short.	As the number of microservices increases, it becomes difficult to deploy in an acceptable speed and becomes more complicated as the frequency of changes increase. Similary, as the granularity of microservices decreases, the number of microservices increases which shifts the complexity towards the interconnections. Ultimately, it becomes complex to monitor and debug microservices.

1.2 Integration

The collaboration among various microservices whilst maintaining deployment autonomy is challenging. In this section, various challanges associated with integration and their potential remedies are discussed.

1.2.1 Shared Database

An easiest way to collaborate, is to allow services to access and update a common data source. However, using this kind of integration creates various problems.[New15]

1. The shared database acts as a point of coupling among the collaborating microservices. If a microservice make any changes to its data schema, there is high probability that other services need to be changed as well. Loose Coupling is compromised.
2. The business logic related to the shared data may be spread across multiple services. Changing the business logic is difficult. Cohesion is compromised.
3. Multiple services are tied to a single database technology. Migrating to a different technology at any point is hard.

Alternatives
There are three distinct alternatives.[Ric15b]

1. **private tables per service** - multiple services share same database underneath but each service owns a set of tables.
2. **schema per service** - multiple services share same database however each service owns its own database schema.

3. **database server per service** - each service has a dedicated database server underneath.

Consequences
The logical separation of data among services increases autonomy but also present some downsides.[Ric16] [Ric15b]

1. The implementation of business transaction which spans multiple services is difficult and not recommended because of theorem 2.1. The solution is to apply eventual consistency 2.2 focussing more on availability.
2. The implementation of queries to join data from multiple databases can be complicated. There are two alternatives to achieve this.
   a) A separate mashup service can be used to handle the logic to join data from multiple services by accessing respective APIs.
   b) CQRS pattern 2.3 can be used by maintaining separate views as well as logic for updating and querying data.
3. The need for sharing data among various autonomous services cannot be avoided completely. It can be achieved by one of the following approaches.
   a) The data can be directly accessed by using the resource owner's API.
   b) The required data can be duplicated into another service and made consistent with the owner's data using event driven approach.

1.2.2 Inter-Service Communication

As much as it is correct to say that a microservice is an autonomous component, for the same reason it is also reasonable to accecpt the necessity of interaction among microservices. The various possible interactions among microservices is shown in figure 1.1.[Ric15a]

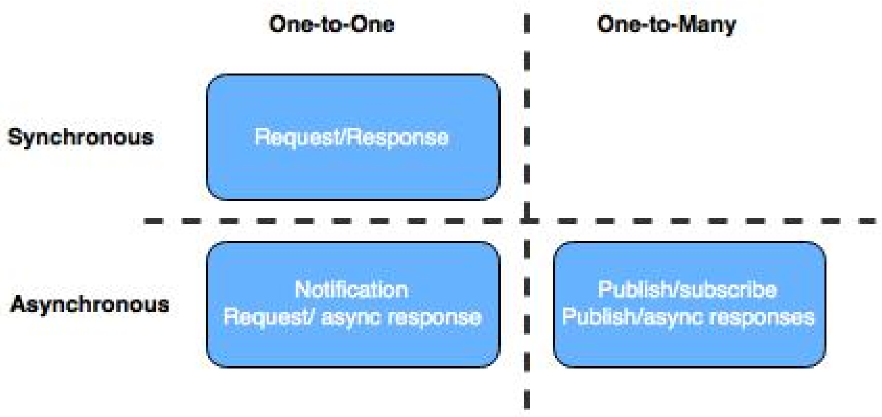

Figure 1.1: Inteaction Styles among microservices [[Ric15a]]

One to One interactions

1. **Request/Response**: It is synchronous interaction where client sends request and expects for response from server.
2. **Notification**: It is asynchronous one way interaction where cliend sends request but no reply is sent from server.
3. **Request/async Response**: It is asynchronous interaction where client requests server but does not block waiting for response. The server send response asynchronously.

One to Many interactions

1. **Publish/subscribe**: A service publishes notification which is consumed by other interested services.
2. **Publish/async responses**: A service publishes request which is consumed by interested services. The services then send asynchronous responses.

1.2.2.1 Synchronous and Asynchronous

Each of the interactions listed in section 1.2.2 has its own place in an application. An application can contained a mixture of these interactions. However, a clear idea

about the requirement as well as drawbacks associated with the interaction is necessary.[New15][Ric14][Mor15]
Synchronous Interaction
It is simple to understand to flow and natural easy way of implenting an interaction. However, as the interactions get higher, it increases the overal blocking period of client and thus increasing latency. The synchronous nature of interaction increases temporal coupling between services which demands both to be active at the same time. Since, the synchronous client needs to know the address and port of server to communicate, it also induces location coupling. With the cloud deployment and auto-scaling, this is not simple. Finally, to mitigate the location coupling, service discovery mechanishm is recommended to be used.

Asynchronous Interaction
The asynchronous interaction decouples services. It also improves latency as the client is not blocked waiting for the response. However, there is one additional component which is message broker to be managed. Additionally, the conceptual understanding of the flow is not natural, so it is not easy to reason about.

1.2.2.2 Example

The figure shows various interactions during creation of order. At first, the client, 'checkout service' in this case sends Restful Post order request to 'Order service'. Next, the 'order service' publishes 'Order Created' event and then sends response to the client. The 'OrderDetails service', which is subscribed to the event 'Order Created' reponds by first fetching necessary prodcut details from 'Product service' using Restful Get request. Finally, the 'OrderDetails service' sends request to 'Email service' for sending email to customer regarding order details but does not wait for response. The 'Email service' sends response to 'OrderDetails service' when email is sent successfully.

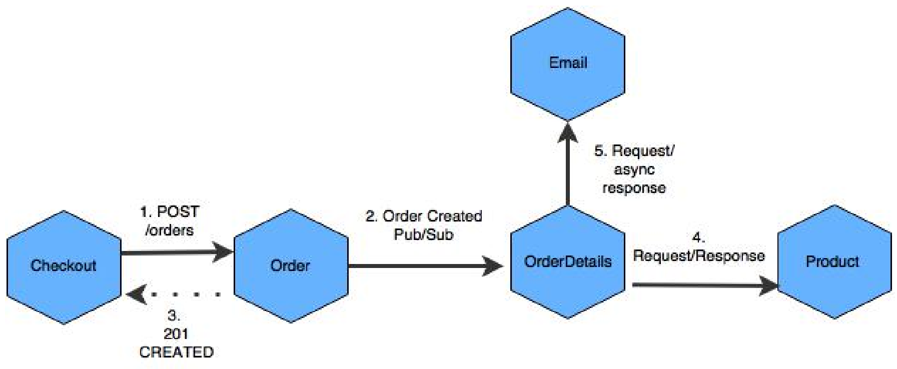

Figure 1.2: Inteaction during Order creation

1.2.3 Breaking Change

Although microservices are autonomous components, they still need to communicate. Also, it is obvious that they undergo changes frequently. However, some changes can be backward incompatible and break their consumers. The breaking changes are inevitable but however the impact of the breaking changes can be somehow reduced by applying various techniques. [New15]

1. Avoid as much as possible
   A good approach to reduce impact of incompatible changes to the consumers is to defer it as long as it is possible. One way is by choosing the correct integration technology such that loose coupling is maintained. For example, using REST as an integration technology is better approach than using shared database because it encaptulates the underlying implementation and consumers are only tied to the interfaces.
   Another approach is to apply TolerantReader pattern while accessing provider's API. [Fow11b] Consumer can get liberal while reading data from provider. Without blindly accepting everything from server, it can only filter the required data without carrying about the payload datastructure and order. So, if the reader is tolerant, the consumer service can still work if the provider adds new fields or change the structure of the response.

2. Catch breaking changes early
   It is also crucial to identify breaking change, if it happens, as soon as possible. It can be achieved by using consumer-driven contracts. Consumers will write the contract to define the expectations from the producer's API in the form of various integration tests. These integration tests can become a part of build process of the

producer so the breaking change can be detected in the build process before the API is accessed by the real consumers.

3. Use Semantic Versioning
   Semantic Versioning is a way to identify the state of current artifact using compact information. It has the form MAJOR.MINOR.PATCH. MAJOR number is increased when backward incompatible changes are made. Similarly, MINOR number is increased when backward compatible functionalities are added. Finally, PATCH number represents that bug fixes are made which are backward compatible.
   With semantic versioning, the consumers can clearly know if the provider's current update will break their API or not. For example, if consumer is using 1.1.3 version of provider's API and the new updated version is 2.1.3, then the consumer should expect some breaking changes and react accordingly.

4. Coexist Different Endpoints
   Whenever a breaking change on a service is deployed, it should be carefully considered not to fail all the consumers immediately. One way is to support old version of end points as well as new ones. This will give some time for consumers to react on their side. Once all the consumers are upgraded to use new version of the provider, the old version end point can be removed. However, using this approach means that additional tests to verify both versions.

5. Coexist concurrent service versions
   Another approach is to support both version of service at once. The requests from old consumers need to be routed to old versioned service and requests from new consumers to the new versioned service. It is mostly used when the cost of updated old consumer is high. However, this also means that two different versions have to be maintained and operated smoothly.

1.3 Handling Failures

With a large number of microservices where communication is happening along not so reliable network, failures are inevitable. At the same time, it becomes challenging as well as highly necessary to handle the failures. Many strategies can be employed for the purpose. [New15][Ric15a][Nyg07]

1. Timeouts
   A service can get blocked indefinitely waiting for response from provider. To prevent this, a threshold value can be set for waiting. However, care should be taken not to choose very low or high value for waiting.

2. Circuit Breaker
   If request to a service keep failing, there is less sense to keep sending request to the same server. It can be a better approach to track the request failures and stop sending further request to the service assuming that there is problem with the connection. By failing fast in such a way will not only save the waiting time for the client but also reduces unnecessary network load. Circuit Breaker pattern helps to accomplish the same. A circuit breaker has three states as shown in the figure 1.3. In normal cases when the connection to the provider is working fine, it is in 'closed' state and all the connections to the provider goes through it. Once the connection starts failing and meets the threshold (can be number of failures or frequency of failures), the circuit breaker switch to 'open' state. At this state, any more connections fail straight away. After certain time interval, the state changes to 'half open' to check the state of provider again. Any connection request at this point passes through. If the connection is succesful, the state switches back to 'closed' state, else to 'open' state.[Fow14] [New15] [Nyg07]

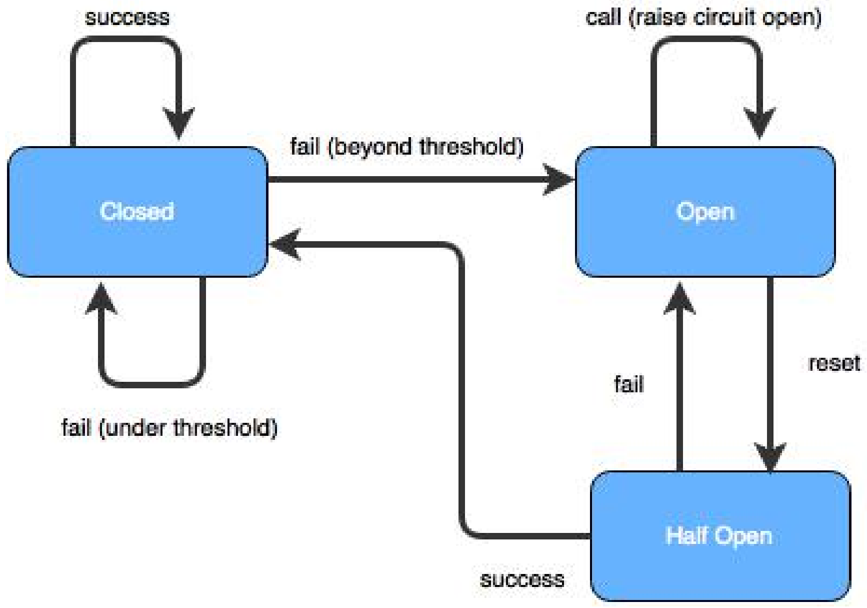

Figure 1.3: States of a circuit breaker [[Fow14]]

3. Bulk Head
   Bulk Head is the approach of segregating various resources as per requirement and assigning them to respective purposes. Maintaining such threshold in avail-

able resources will save any resource from being constrained whenever there is problem in any section. One example of such bulkhead is the assignment of separate connection pool for each downstream resources. In this way, if problem in the request from one connection pool will not affect another connection pool. [New15] [Nyg07]

4. Provide fallbacks
   Various fallback logic can be applied along with timeout or circuit breaker to provide alternative mechanism to respond. For example, cached data or default value can be returned.

1.4 Monitoring

With monolithic deployment, monitoring can be achieved by sifting through few logs on the server and various server metrices. The complexity can get added to some extent if the application is scaled along multiple server, in which case, monitoring can be done by accumulating various server metrices and going through each log file on each host. The complexity goes to whole new level when monitoring microservice deployment because of large number of individual log files produced by microservices. As difficult it is to go through each log files, it is even more challenging to trace any request contextually along all the log files.
There are various ways to work around these complexities. [New15] [Sim14]

1. Log Aggregation and Visualition
   A large number of logs in different locations can be intimidating. The tracing of logs can be easier if the logs could be aggregated in a centralized location and then visualized in a better way such as graphs. One of such variations can be achieved using ELK stack as shown in the figure 1.4. [Ani14] [New15]

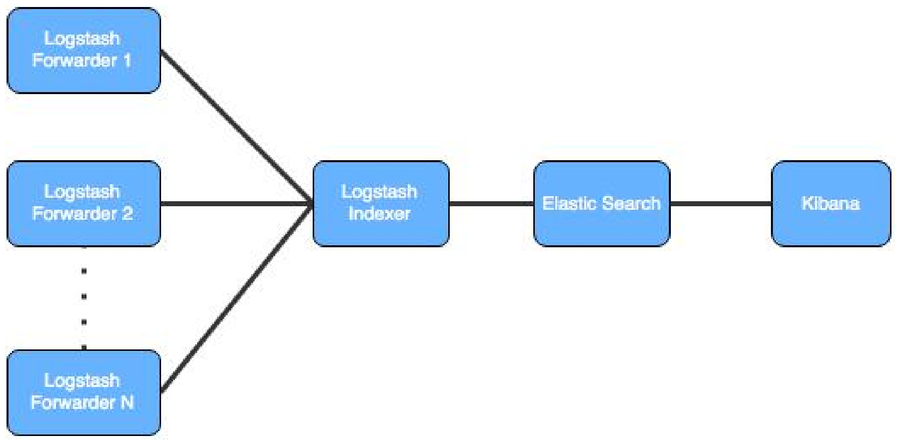

Figure 1.4: ELK stack

The ELK stack consists of following major components.

a) Logstash Forwarder
It is the component installed in each node which forwards the selected log files to the central Logstash server.

b) Logstash Indexer
It is the central logstash component which gathers all the log files and process them.

c) Elastic Search
The Logstash Indexer forwards the log data and stores into Elastic Search.

d) Kibana
It provides web interface to search and visualize the log data.
The stack makes it easy to trace log and visualize graphs along various metrices.

2. Synthetic Monitoring
Another aspect of monitoring which can be helpful, is to collect various health status such as availability, response time etc. There are many tools to make it easier. One such tool is 'uptime', which is a remote monitoring application and provides email notification as well as features such as webhook to notify a url using http POST.
Rather than waiting for something to get wrong, a selected sets of important business logic can be tested in regular interval. The result can be fed into notification subsystem, which will trigger notification to responsible parties. This

will ensure confidence that any problem will be detected as soon as possible and can be worked on sooner. [Sim14] [New15]

3. Correlation ID
   A request from a client is fulfilled by a number of microservices, each microservice being responsible for certain part of the whole functionality. The request passes through various microservices until it succeeds. For this reason, it is difficult to track the request as well as visualize the complete logical path. Correlation ID is an unique identification code given to the request by the microservices at the user end and passes it towards the downstream microservice. Each downstream microservice does the same and pass along the ID. It gets easier to capture a complete picture of any request in that way.

1.5 Deployment

A major advantage of microservices is that the features can updated and delivered quickly due to the autonomity and independent deployability of each microservices. However, with the increase in the number of microservices and the rate of changes that can happen in each microservice, fast deployment is not straight forward. The culture of doing deployment for monolithic application will not work perfectly on microservices. This section discusses how it can be achieved.
There are two major parts in deployment. In order to speed up deployment in microservices, approach to accomplish each part has to be changed slightly.

1. Continuous Integration
   It is a software development practice in which newly checked in code are followed by automated build and verification to ensure that the integration was successful. Following continuous integration does not only automate the artifact creation process but also reduces the feedback cycle of the code providing opportunity to improving quality.
   To ensure autonomy and agility in microservices, a better approach is to assign separate source code repository and separate build for each microservice as shown in figure 1.5. Each repository is mapped to separate build in Continuous Integration server. Any changes in a microservice triggers corresponding build in Continuous Integration server and creates a single artifact for that microservice. An additional advantage of this approach is clear assignment of ownership of the repository and build responsibility to the teams, owning the microservices. [New15] [Fow06]

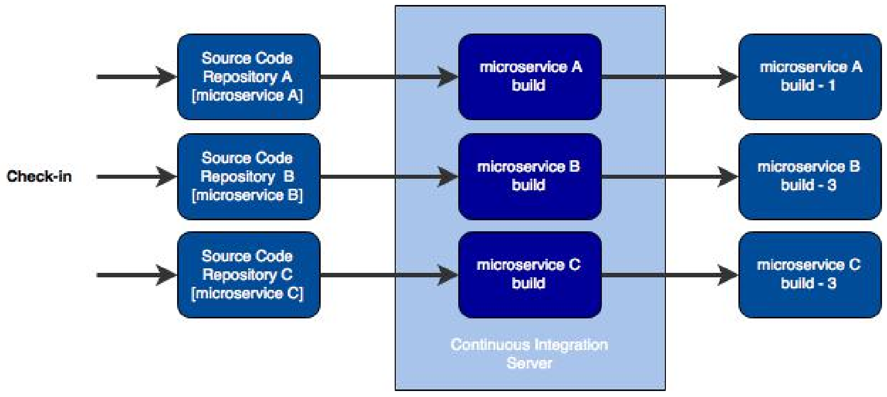
Check-in
Source Code Repository A [microservice A]
Source Code Repository B [microservice B]
Source Code Repository C [microservice C]
microservice A build
microservice B build
microservice C build
Continuous Integration Server
microservice A build - 1
microservice B build - 3
microservice C build - 3

Figure 1.5: Continuous Integration [New15]

2. Continuous Delivery
   It is a discipline in which every small change is verified immediately to check deployability of the application. There are two major concepts associated with continuous delivery. First one is continuous integration of source code and creation of artifact. The artifact is then fed into a build pipeline. A build pipeline is a series of stages, each stage responsible for the verification of certain aspects of the artifact. A successful verification at each stage ensures the readiness of the artifact further close to release.

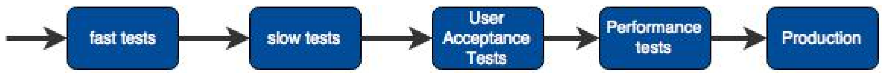

Figure 1.6: Continuous Delivery Pipeline [New15]

A version of build pipeline is shown in figure 1.6. Here, faster tests are performed ahead and slow tests, manual testing are performed down the line. This kind of successive verifications make it faster to identify problems, clear visibility of problems and improve the quality as well as confidence in the quality of the application.
Each microservice has an independent build pipeline. The artifact build during continuous integration triggered by any change in the microservices is fed into corresponding build pipeline for the microservice. [New15] [Fow13]

2 Appendices

2.1 CAP Theorem

CAP Theorem was published by scientist Eric Brewer and also called as Brewer's Theorem. There are three major requirements for deploying an application in a distributed environment.

1. **Consistency**
   A functionality of a service is accomplished as a whole or not at all. All the nodes accessing any data see the same version at any given time.

2. **Availability**
   The functionalities provided by a service are available and working.

3. **Partition Tolerance**
   The partitions which occurs due to problems in the network does not affect the operations of the system unless the whole network fails.

According to the theorem, as the system scales across a number of nodes and the volume of requests increase, it becomes difficult to achieve all the three qualities but to compromise any one of them. [Bro09] Network partitions cannot be avoided completely due to the very nature of distributed system and so is the quality of partition tolerance has to be maintained. The choice is the trade off between availability and consistency. It is completely dependent upon business requirement which one to choose. [BG13]

2.2 Eventual Consistency

It is a weaker form of consistency which guarantees all read to a data item return the same value eventually, if no additional updates are performend to the same data item. For any update, only the nodes which are viable and reachable at the moment are updated and the remaining nodes are updated when they are back. [BG13]

2.3 Command Query Responsibility Segregation(CQRS)

In this pattern as shown in the figure 2.1, the conceptual model is divided into two separate models, each one for update and read. It refers to creating different object model handled by different logical processes. The database can be shared, where it acts as integration point but can have different database as well.

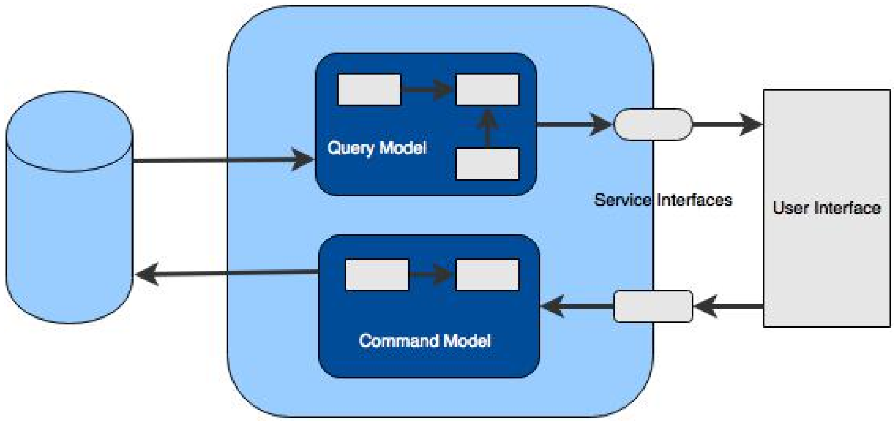

Figure 2.1: CQRS [[Fow11a]]

Acronyms

API Application Programming Interface.

CQRS Command Query Responsibility Segregation.

CRUD create, read, update, delete.

DEP Dependency.

HCP Hana Cloud Platform.

IDE Integrated Development Environment.

IFBS International Financial and Brokerage Services.

ISCI Inter Service Coupling Index.

ODC Operation Data Granularity.

ODG Operation Data Granularity.

OFG Operation Functionality Granularity.

RCS Relative Coupling of Services.

REST Representational State Transfer.

RIS Relative Importance of Services.

SCG Service Capability Granularity.

SDG Service Data Granularity.

SDLC Software Development Life Cycle.

SFCI Service Functional Cohesion Index.

SIDC	Service Interface Data Cohesion.
SIUC	Service Interface Usage Cohesion.
SIUC	Service Sequential Usage Cohesion.
SLC	Self Containment.
SMCI	Service Message Coupling Index.
SOAF	Service Oriented Architecture Framework.
SOCI	Service Operational Coupling Index.
SOG	Service Operations Granularity.
SRI	Service Reuse Index.
SWIFT	Society for Worldwide Interbank Financial Telecommunication.
UML	Unified Modeling Language.
YaaS	Hybris as a Service.

List of Figures

List of Tables

Bibliography

[Ani14] M. Anicas. *How To Use Logstash and kibana To Centralize Logs On Ubuntu 14.04*. June 2014.

[BG13] P. Bailis and A. Ghodsi. “Eventual Consistency Today: Limitations, Extensions, and Beyond How can applications be built on eventually consistent infrastructure given no guarantee of safety?” In: 11 (Apr. 2013).

[Bro09] J. Browne. *Brewer's CAP Theorem*. Jan. 2009.

[Fac14] P. Factor. *The Eight Fallacies of Distributed Computing*. Dec. 2014.

[Fow06] M. Fowler. *Continuous Integration*. May 2006.

[Fow11a] M. Fowler. *CQRS*. July 2011.

[Fow11b] M. Fowler. *TolerantReader*. May 2011.

[Fow13] M. Fowler. *ContinuousDelivery*. May 2013.

[Fow14] M. Fowler. *CircuitBreaker*. Mar. 2014.

[Fow15] M. Fowler. *Microservice Trade-Offs*. July 2015.

[Mor15] B. Morris. *Why REST is not a silver bullet for service integration*. Jan. 2015.

[New15] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[Nyg07] M. T. Nygard. *Release It!* Pragmatic Bookshelf, Mar. 2007.

[Ric14] C. Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. May 2014.

[Ric15a] C. Richardson. *Building Microservices: Inter-Process Communication in a Microservices Architecture*. July 2015.

[Ric15b] C. Richardson. *Does each microservice really need its own database?* Sept. 2015.

[Ric16] C. Richardson. *Pattern: Database per service*. 2016.

[Sim14] S. D. Simone. *Sam Newman: Practical Implications of Microservices in 14 Tips*. Oct. 2014.