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

The microservices architecture provides various advantages such as agility, independent scalability etc., as compared to the monolithic architecture and thus has gained a lot of attention. However, implementing microservices architecture is still a challenge as many concepts within the microservices architecture including granularity and modeling process are not yet clearly defined and documented. This research attempts to provide a clear understanding of these concepts and finally create a comprehensive guidelines for implementing microservices.
Various keywords from definitions provided by different authors are taken and cateogorized into various conceptual areas. These concepts along with the keywords are researched thoroughly to understand the microservices architecture. Additionally, the three important drivers: quality attributes, constraints and principles, are also focussed for creating the guidelines.
The findings of this research indicate that eventhough microservices emphasize on the concept of creating small services, the notion of appropriate granularity is more important and depends upon four basic concepts which are : single responsibility, autonomy, infrastructure capability and business value. Additionally, the quality attributes such as coupling, cohesion etc should also be considered for identification of microservices. Furthermore, in order to identify microservices, either the domain driven design approach or the use case refactoring approach can be used. Both these approaches can be effective in identifying microservices, but the concept of bounded context in domain driven design approach identifies autonomous services with single responsibility. Apart from literature, a detail study of the architectural approach used in industry named SAP Hybris is conducted. The interviews conducted with their key personnels has given important insight into the process of modeling as well as operating microservices. Moreover, the challenges for implementing microservices as well as the approach to tackle them are done based on the literature review and the interviews done at SAP Hybris.
Finally, the findings are used to create a detail guidelines for implementing microservices. The guidelines captures how to approach modeling microservices architecture and how to tackle its operational complexities.

Contents

1 Challanges of Microservices Architecture

This chapter focuses on various challenges faced while implementing microservices. In Section 1.1, various advantages of the microservices architecural approach along with the challenges are listed. Then each challenge is broken down into several sub-challenges and discussed further after Section 1.2. For each challenge, various techniques which can be used to handle them are also presented. Additionally, in order to give the insight about approach in industries, the techniques used in SAP Hybris are also presented.

1.1 Introduction

The Section **??** lists some drawbacks of monolithic architecture. These disadvantages have become the motivation for adopting the microservices architectural approach. The microservices architecture offers opportunities in various aspects however it can also be challenging to utilize them properly. The responses from Interview Question **??** also highlight some prominent challanges. In this chapter, the challanges and advantages of microservices architectural approach are discussed. **[Fowler:2015aa]**

Advantages

Strong Modular Boundaries
It is not completely true that the monoliths have weaker modular structure than microservices. But, as the system gets bigger, it is very easy for a monolith to turn into a big ball of mud. However, it is very difficult to do the same with the microservices. Each microservice is a cohesive unit with full control upon its business entities. The only way to access its data and functionalities is through its API.

Challanges

Distributed System Complexity
The infrastructure of the microservices architecture is distributed, which introduces many complications, as listed by 8 fallacies **[Factor:2014aa]**. The calls are remote and are imminent to accomplish business goals. The remote calls are slower than local calls and affect the performance to a great deal. Additionally, network is not completely reliable which makes it necessary to handle failures.

Independent Deployment
Due to the automony characteristic of the microservices, 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.[**Newman:2015aa**]

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

Agile
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, decreasing the release cycle time.

Operational Complexity
As the number of microservices increases, it becomes difficult to deploy in an acceptable speed. Additionally, realising new version of the microservices 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 difficult to monitor and debug microservices.

In the following sections of this chapter, various ways to tackle the challenges listed in Section 1.1 are discussed in detail.

1.2 Integration

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

1.2.1 Sharing Data

An easiest way for collaboration among the microservices is to allow the microservices to access and update a common datasource. However, using this kind of integration creates various problems.[**Newman:2015aa**]
Problems

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 to tackle the problems listed above.[**Richardson:2015aa**]

1. **private tables per service** - multiple microservices share the same database underneath but each microservice owns a set of tables.
2. **schema per service** - multiple microservices share same database however each microservice owns its own database schema.
3. **database server per service** - each microservice has a dedicated database server underneath.

Techniques

The logical separation of data among services discussed in the Section 1.2.1 increases autonomy but also makes it difficult to access and create consistent view of data. However, the problems can be compensated using following approaches.[**Richardson:2016aa**] [**Richardson:2015aa**]

1. The implementation of business transaction which spans multiple microservices is difficult and also not recommended because of Theorem **??**. The solution is to apply eventual consistency **??** focusing 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 microservice can be used to handle the logic to join data from multiple microservices by accessing respective APIs.
   b) CQRS pattern **??** can be used by maintaining separate a) model, and b) logic for updating as well as 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 data can be duplicated into another microservice which accesses the data. The duplicate data can be made consistent with the owner's data using event driven approach.

SAP Hybris uses the technique of maintaining private schema per microservice as listed in the Section 1.2.1 in order to share data. For this purpose, there is a generic microservice called "Document" which handles the task of managing data for all microservices per each tenant and application. Furthermore, SAP Hybris follows eventual consistency and creates various mashup services for collective view of data from multiple services. Additionally, data are duplicated to decrease the network overload and when fresh data are required, resource owner's API is accessed directly.

1.2.2 Inter-Service Communication

A microservice is an autonomous component but the necessity of interaction among microservices cannot be denied. The various possible interactions among microservices is shown in Figure 1.1.[**Richardson:2015ab**]

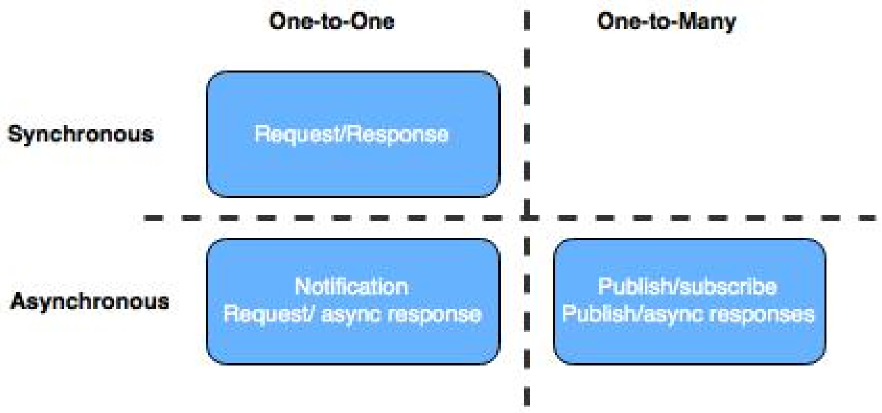

Figure 1.1: Inteaction Styles among microservices [[**Richardson:2015ab**]]

One to One interactions

1. **Request/Response**: It is synchronous interaction where client sends request and expects for response from the server.

2. **Notification**: It is asynchronous one way interaction where client sends request but no reply is sent from the server.
3. **Request/async Response**: It is asynchronous interaction where client requests server but does not get blocked waiting for the response. The server send the response asynchronously.

One to Many interactions

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

Techniques

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 contain a mixture of these interactions. However, a clear idea about the requirement as well as thorough knowledge regarding drawbacks associated with each interaction is necessary before choosing any type of interaction for a specific case.**[Newman:2015aa][Richardson:2014aa][Morris:2015aa]**
Synchronous Interaction
In synchronous interaction, it is simple to understand the flow, The synchronous approach is a natural way of communication. However, as the volume of interactions get higher and distributes to various levels or branches, it increases the overall blocking period of client and thus increases latency. The synchronous nature of interaction increases temporal coupling between microservices because it demands both consumer and producer microservices to be active at the same time. Since, the synchronous client needs to know the address and port of the server to communicate, it also inducess location coupling. With the cloud deployment and auto-scaling, this is not simple. Finally, to tackle the location coupling, service discovery mechanishm is recommended to be used.

Asynchronous Interaction
The asynchronous interaction decouples the microservices. It also improves latency

as the client is not blocked waiting for the response. However, there is one additional component called message broker to be managed. Additionally, the conceptual understanding of the flow is not natural and not easy to reason about.

SAP Hybris uses both asynchronous as well as synchronous communication where appropriate. For, synchronous, it uses REST calls where as for asynchronous messaging it uses a generic microservice called "PubSub", which again uses Apache Kafka for managing publication and subscription of messages. An instance of the various interaction is shown in the Example 1.2.2.2.

1.2.2.2 Example

The Figure 1.2 shows various interactions during creation of an order. At first, the client, ’checkout microservice’ in this case, sends Restful Post order request to the ’Order microservice’. Next, the ’order microservice’ publishes ’Order Created’ event and then sends response to the client. The ’OrderDetails microservice’, which is subscribed to the event ’Order Created’ acts on the event by first fetching necessary product details from the ’Product microservice’ using Restful Get request. Finally, the ’OrderDetails service’ sends request to the ’Email microservice’ for sending email to the customer regarding order details but does not wait for response. The ’Email service’ sends response to ’OrderDetails service’ when email is sent successfully.

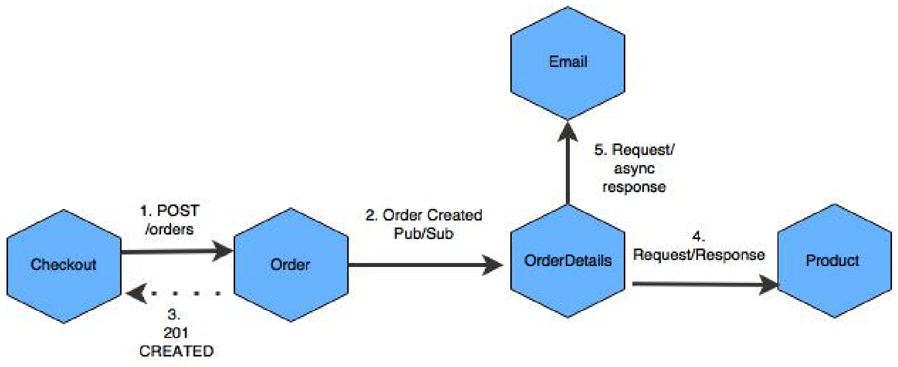

Figure 1.2: Interaction during Order creation

1.3 Distributed System Complexity

1.3.1 Breaking Change

Although microservices are autonomous components, they still need to communicate. Moreover, they also undergo changes frequently. However, some changes can be backward incompatible and break their consumers. The breaking changes are inevitable but the impact of the breaking changes can be somehow reduced by applying various techniques. [**Newman:2015aa**]

Techniques

1. **Avoid as much as possible**
   A good approach to reduce impact of incompatible changes of producer microservices 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 the former approach encapsulates the underlying implementation and consumers are only tied to the interfaces.
   Another approach is to apply TolerantReader pattern when accessing provider's API [**Fowler:2011aa**]. The consumer can get liberal while reading data from the provider. Without blindly accepting everything from the server, the consumer can filter the required data only without caring about the payload structure and the sequence of data. So, if the consumer is tolerant while accessing the provider, the consumer service can still work even if the provider adds new fields or change the structure of the response.

2. **Catch breaking changes early**
   It is also crucial to identify the breaking change as soon as it happens. It can be achieved by using consumer-driven contracts. Consumers will write the contract to define what they expect from the producer's API in the form of various integration tests. These integration tests can become a part of the build process for the producer microservice. In this way, any 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 may breaks 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, the new deployment should be carefully managed not to fail all the consumers immediately. One way is to support old version of end points as well as new ones. This gives some time for consumers to react on their side. Once all the consumers are upgraded to use new version of the provider, the old versioned end point can be removed. However, using this approach means that additional tests are required to verify both versions of the end points.

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

At SAP Hybris, breaking change is avoided as much as possible using REST and also Tolerant Reader pattern. When data is read from another microservice, the data is first handled by internal mapper library which maps the input data into the internal representation to be used by microservice. Additionally, for providing detailed information regarding state of current version of the microservice, semantic versioning is used. If there is breaking change in few endpoints, then multiple version of them is also maintained for short period of time giving the consumer fair amount of time for migrattion. Similarly, if there are breaking changes in most of the endpoints then the entire microservice is maintained with different versions.

1.3.2 Handling Failures

In the microservices architectural approach, communication among the microservices is achieved along quite unreliable network, failures are inevitable. So, it becomes challenging as well as highly necessary to handle the failures. Many strategies can be applied for the purpose. **[Newman:2015aa][Richardson:2015ab][Nygard:2007aa]**

Techniques

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

2. **Circuit Breaker**
   If a request to a service keeps failing, there is no value to keep sending request to the same server. It can be a better approach to identify when the request fails and stop sending further request to the provider microservice 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 through the circuit breaker 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.**[Fowler:2014ac] [Newman:2015aa] [Nygard:2007aa]**

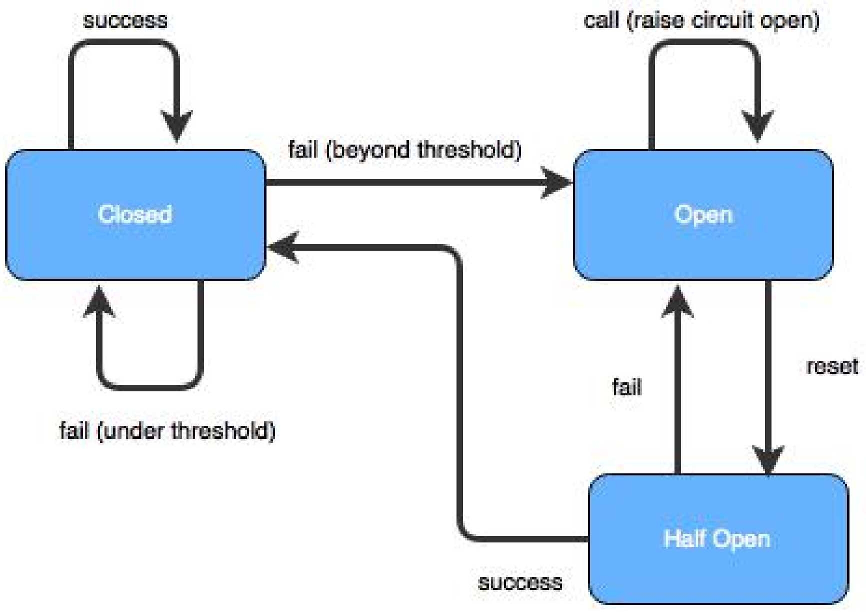

Figure 1.3: States of a circuit breaker [[**Fowler:2014ac**]]

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 available resources will save any resource from being constrained whenever there is problem in any part. One example of such bulkhead is the assignment of separate connection pool for each downstream resources. In this way, if there is a problem in the request from one connection pool, it will not affect another connection pool and also not consume all available resources. [**Newman:2015aa**] [**Nygard:2007aa**]

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 as a fallback mechanism.

At SAP Hybris, synchronous requests are made recursively for certain number of times, each attempt waits for the response only for certain fixed amount of time depending upon the use case. In order to save network resources and provide fault tolerance, circuit breaker pattern is used.

1.4 Operational Complexity

1.4.1 Monitoring

With monolithic deployment, monitoring can be achieved by sifting through few logs on the server and looking various server metrics. The complexity can increase to some extent if the application is scaled along multiple servers. In that case, monitoring can be performed by accumulating various server metrics and going through each log file on each host. The complexity goes to whole new level when monitoring microservices deployment because of large number of individual log files produced by the microservices. In addtion to the difficulty of going through each log file, it is more challenging to trace any request contextually along all the log files.

Techniques

There are various ways to work around these complexities [**Newman:2015aa**] [**Simone:2014aa**].

1. **Log Aggregation and Visualition**
   A large number of logs in different servers 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 solutions can be achieved using ELK stack as shown in the Figure 1.4. [**Anicas:2014aa**] [**Newman:2015aa**]

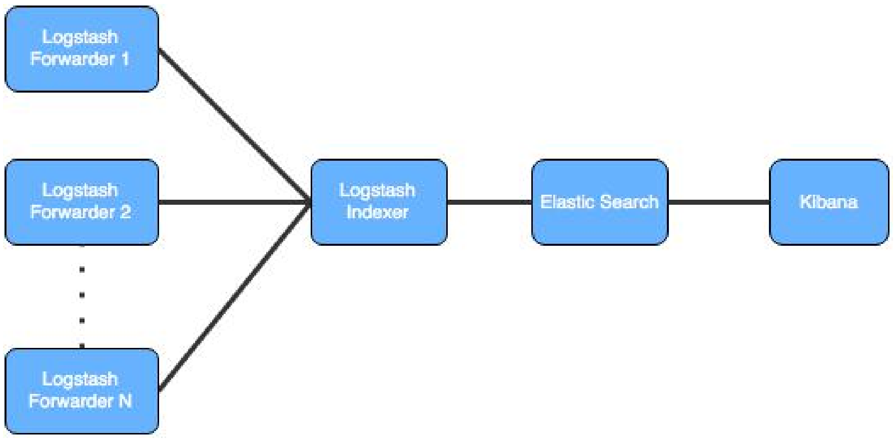

Figure 1.4: ELK stack

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 a web interface to search and visualize the log data.
The stack makes it easy to trace logs and visualize graphs along various metrics.

2. **Synthetic Monitoring**
Another useful aspect of monitoring 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 set of important business logic can be tested at regular interval. The result can be fed into notification subsystem, which will trigger notification to the responsible parties. This ensures confidence that any problem can be detected as soon as possible and can be worked on sooner. **[Simone:2014aa] [Newman:2015aa]**

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 assigned to the request by the microservice at the user end and passes the 'Correlation ID' towards the downstream microservices. Each downstream microservice does the same and pass along the ID. In this way, a complete picture of any request is easier to visualize.

SAP Hybris uses ELK stack for monitoring and log visualization. In order to check the health status of the microservices, 'uptime' tool is used. The webhook from the tool updates the central status page showing the current status of all the existing microservices. Any problem triggered by 'uptime' is also reflected in this page. Similary, a set of smoke tests are triggered from continuous integration tool such as TeamCity at regular interval. In order to track the requests at each microservices, a unique code is assigned to each request. The code is assigned by API-Proxy server which is the central gateway for any incoming request into YaaS.

1.4.2 Deployment

A major advantage of the microservices is that the features can updated and delivered quickly due to the autonomity and independent deployability of each microservice. 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 applications does not work perfectly on microservices. This section discusses how continuous deployment can be achieved in the microservices.
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.

Techiniques

1. **Continuous Integration**
   It is a software development practice in which newly checked in code are followed by the automated build and verification phase 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 improve quality.
   To ensure autonomy and agility in microservices, a better approach is to assign separate sourcecode repository and separate build for each microservice as shown in the Figure 1.5. Each repository is mapped to a separate build in Continuous Integration server. Any changes in the microservice sourcecode repository 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 the repository ownership and build responsibility to the teams who owns the respective microservices. **[Newman:2015aa] [Fowler:2006ab]**

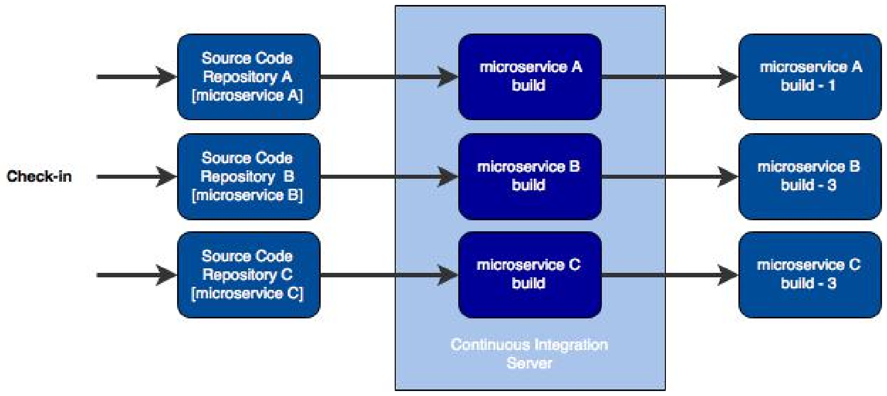
Check-in
Source Code Repository A [microservice A]
Source Code Repository B [microservice B]
Source Code Repository C [microservice C]
microservice A build
microservice B build
microservice C build
Continuous Integration Server
microservice A build - 1
microservice B build - 3
microservice C build -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

Figure 1.5: Continuous Integration [**Newman:2015aa**]

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 deployment. First one is continuous integration of the sourcecode and creation of artifact. Next one is continuous delivery in which the artifact is fed into a build pipeline. A build pipeline is a series of build stages, each stage responsible for the verification of certain aspects of the artifact. A successful verification at each stage ensures more readiness of the artifact to release.

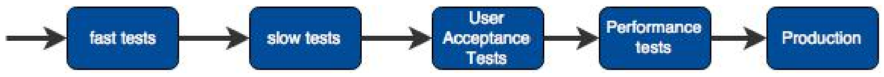

Figure 1.6: Continuous Delivery Pipeline [**Newman:2015aa**]

A version of build pipeline is shown in the Figure 1.6. Here, faster tests are performed ahead and then a) slow tests and b) manual testing are performed down the line. This kind of successive verifications make it faster to identify problems, create clear visibility of the 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 then fed into corresponding build pipeline for the microservice. [**Newman:2015aa**] [**Fowler:2013aa**]

The variation of continuous integration and deployment used at SAP Hybris is already discussed in Section **??**.

1.5 Conclusion

In this chapter, various challenges needed to handle when implementing microservices architectural approach are listed. Each challenge is further broken down into fine constraints. Secondly, various ways of handling those constraints are discussed. Finally, the approaches used in SAP Hybris for handling the challenges are also presented. Microservices architecture provides efficient way to develop agile software. However, without a good grasp in the techniques to handle the challenges, it may not be a good idea to start with microservice in the first place.

List of Figures

List of Tables