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 Architecture at Hybris

1.1 Overview

SAP YaaS provides a variety of business services to support as well as enhance the products offered as SAP hybris front office such as hybris Commerce, hybris Marketing, hybris Billing etc. Using these offered services, developers can create their own business services focussed on their customer requirements.
The figure 1.1 provides the overview of YaaS. YaaS provides various business processes as a service (bPaaS) essential to develop applications and services thus filling up the gap between SaaS and HCP. For that purpose, it consumes the application services (aPaaS) provided by HCP.

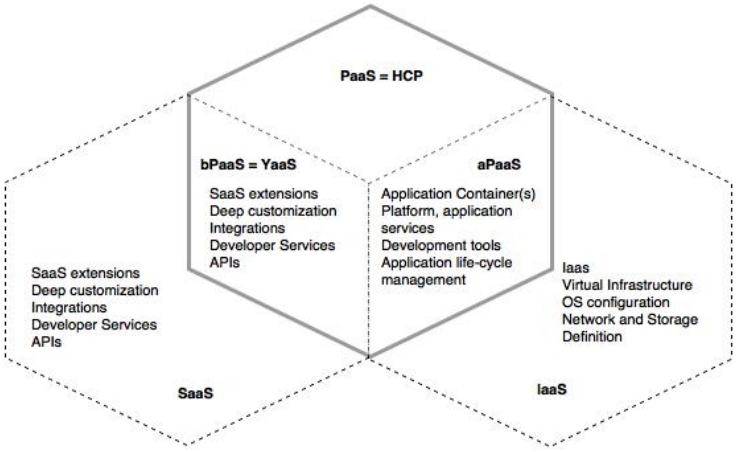


Figure 1.1: YaaS and HCP [Hir15]

1.2 Vision

The vision of YaaS can be clarified with the following statement.

"A cloud platform that allows everyone to easily develop, extend and sell services and applications."
[Stu15]

The vision can be broadly categorized into following objectives.

1. **Cloud First**
   The different parts of the application need to be scaled independently.
2. **Autonomy**
   The development teams should be able to develop their modules independent of other teams and able to freely choose the technology that fits the job.
3. **Retain Speed**
   The new features will be of value to customers if could be able to be released as fast as possible.
4. **Community**
   It should be possible for the components to be shared across internal and external developers.

The definition of microservices **??** as well as the characteristics of microservices [ref] signifies clearly that microservices architecture can be a good fit for YaaS architecture.

1.3 YaaS Architecture Principles

The Agile Manifesto [Bec+11] provides various principles to develop a software in a better way. It focus on fast response to the requirement changes with frequent continous delivery of software artifacts with close collaboration of customer and self-organizing teams.
The Reactive Manifesto [Bon+14] lists various qualities of a reactive system which includes responsiveness (acceptable consistent response time, quick detection and solution of problem), resilience (responsive during the event of failure) , elasticity (responsive during varying amount of workload) and asynchronous message passing. Loose coupling is highly focused.
Furthermore, the twelve factors from Heroku [Wig12] provides a methodology for minimizing time and cost to develop software applications as services. It emphasizes on scalability of applications, explicit declaration as well as isolation of dependencies

among components, multiple continuous deployments from a single version controlled codebase with separate pipelines for build, release and run.
Finally, the microservices architecture provides techiques of developing an application as a collection of autonomous small sized services focused on single responsibility. [section **??**] It focus on independent deployment capability of individual microservices and suggest to use lightweight mechanisms such as http for communication among services. The architecture offers various advantages, few of them being individual independent scalability of each microservice, resilience achieved by isolating failure in a component and technology heterogeneity among various development teams.[New15] Following the principles mentioned above, a list of principles are compiled to be used as guidelines for creating microservices. [Stu15] The list of principles are listed below.
the y-Factors

1. **Self-Sufficient Teams**
   The teams have independence and freedom for any decision related to the design and development of their components. This freedom is balanced by the responsibility for the team to handle the complete lifecycle of their components including smooth running as well as performance in production and troubleshooting in case of any problems.

2. **Open Technology landscape**
   The team have freedom to choose any technology that they believe fits the requirement. They are completely responsible for the quality of their product. This gives teams, ownership as well as satisfaction for their products.

3. **Release early, release often**
   The agile manifesto and twelve factors from Heroku also focus on continuous delivery of product to the clients. This will decrease time of feedback and ultimately fast response to the feedback resulting in high customer satisfaction. The teams are responsible to create build and delivery pipeline for all available environments.

4. **Responsibility**
   The teams are the only responsible groups to work directly with the customers on behalf of their products. They need to work on the feedback provided by the customers. It will increase the quality of the products and relationship with customers. All the responsibilities including scaling, maintaining, supporting and improving products are handled by respective teams.

5. **APIs first**

The API is a contract between service and consumers. The decision regarding design and development is very important as it can be one of the greatest assets if good or else can be a huge liability if done bad. [Blo16] The articles [Blo16] and [Bla08] list various characteristics of good API including simplicity, extensibility, maintainability, completeness, small and focussing on single functionality. Furthermore, a good approach to develop API is to first design iteratively before implementation in order to understand the requirement clearly. Another important aspect of a good API among many is complete and updated documentation.

6. **Predictable and easy-to-use UI**
   The user interfaces should be simple, consistent across the system and also consistent to various user friendly patterns.[Sol12] The articles [Mar13] and [Por16] specify additional principles to be considered when designing user interfaces. A few of them includes providing clarity with regard to purpose, smart organization and respecting the expectation as well as requirements of customers.

7. **Small and Simple Services** A service should be small and focused on cohesive functionalities. The concept closely relates to the single responsibility principle. [Mar16] A good approach is to explicitly create boundaries around business capabilities.[New15]

8. **Scalability of technology**
   The choice of technologies should be cloud friendly such that the products can scale cost-efficiently and without delay. It is influenced by the elasticity principle provided by the reactive manifesto.[Bon+14]

9. **Design for failure** The service should be responsive in the time of failure. It can be possible by containment and isolation of failure within each component. Similarly, the recovery should be handled gracefully without affecting the overal availability of the entire system. [Bon+14]

10. **Independent Services**
    The services should be autonomous. Each service should be able to be deployed independently. The services should be loosely coupled, they could be changed independently of eachother. The concept is highly enforced by exposing functionalities via APIs and using lightweight network calls as only way of communication among services. [New15]

11. **Understand Your System** It is crucial to have a good understanding of problem domain in order to create a good design. The concept of domain driven design

strongly motivates this approach to indentify individual autonomous components, their boundaries and the communication patterns among them. [New15] Furthermore, it is also important to understand the expectations of consumers regarding performance and then to realize them accordingly. It is possible only by installing necessary operational capabilities such as continuous delivery, monitoring, scaling and resilience.

1.4 Interviews

With the intension to anticipate the overall belief, culture and practice followed by hybris to develop YaaS, a number of interviews were conducted with a subset of key personnels who are directly involved in YaaS. The list of interviewees with their corresponsing roles is shown in the table 1.1. In order to preserve anonymity, the real names are replaced with forged ones.

Names	**Roles**
John Doe	Product Manager
Ivan Horvat	Senior Developer
Jane Doe	Product Manager
Mario Rossi	Product Manager
Nanashi No Gombe	Product Manager
Hans Meier	Architect
Otto Normalverbraucher	Product Manager
Jan Kowalski	Senior Developer

Table 1.1: Interviewee List

1.4.1 Hypothesis

During the process of solving the concerned research questions, various topics of high value have been discovered. The topics are:

1. Granularity
2. Quality Attributes
3. Process to design microservices

Similarly, based on research findings, a list of hypothesis has been made with respect to each topic.

Hypothesis 1:

The correct size of microservices is determined by:

1. **Single Responsibility Principle**
2. **Autonomy, and**
3. **Infrastructure Capability**

According S.Newman, a good microservices should be small and focus on accomplishing one thing well. Additionally, it could be independently deployed and updated. This strongly suggests the requirement of autonomy. [New15] Futhermore, M. Stine also agree that the size of a microservices should be determined by single responsibility principle. [Sti14] Also, the principle listed in **??** indicate that the advancement in the current technology and culture of the organization also defines size of microservices. [RS07]

Hypothesis 2:

Domain driven design is the optimum approach to design microservices.

The concept of domain driven design to design microservices is promoted by S. Newman and M. Fowler. [New15] [FL14] Similary, there can be found evendence of domain driven design being used in various projects to design microservices. The list of projects is shown in the table **??**.

1.4.2 Interview Compilation

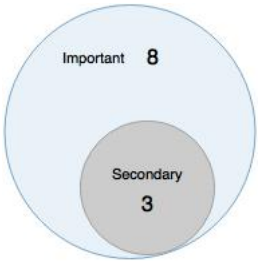
For each topic listed above in section 1.4.1, a list of questionaires is prepared. The questions were then asked to the interviewees. In the remaining part of this section, an attempt is made to compile the response from the interviews on the questionaires for each topic.

1. Granularity

Question 1.1

"Do you consider the size of microservices when desigining microservice architecture?"

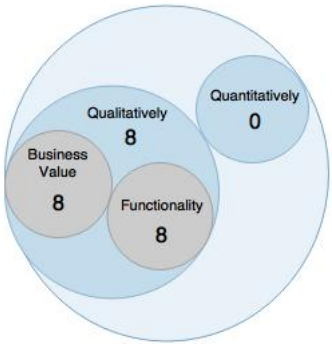
Response 1.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
Important	1	1	1	0	1	1	0	0	**5**
Secondary, Not primary	0	0	0	1	0	0	1	1	**3**
Not Important	0	0	0	0	0	0	0	0	**0**



Question 1.2

"How do you measure size of a microservice?"

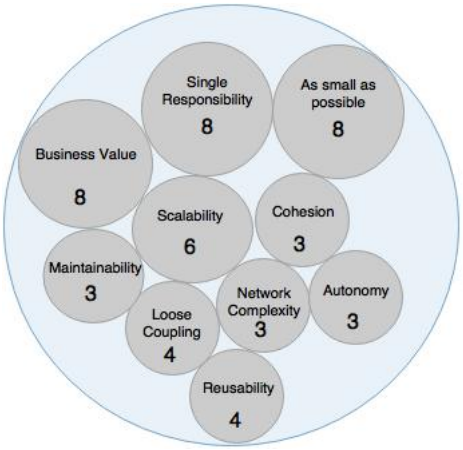
Response 1.2		John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
Qualitatively		1	1	1	1	1	1	1	1	**8**
a.	Functionality	1	1	1	1	1	1	1	1	**8**
b.	Business value	1	1	1	1	1	1	1	1	**8**
Quantitatively		0	0	0	0	0	0	0	0	**0**



Question 1.3

"What kind of size do you try to achieve for a microservice?"

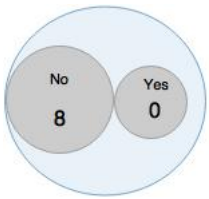
Response 1.3		John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
as small as possible in terms of functionality		1	1	1	1	1	1	1	1	**8**
influenced by other attributes										
a.	Business Value	1	1	1	1	1	1	1	1	**8**
b.	Single Responsibility	1	1	1	1	1	1	1	1	**8**
c.	Loose Coupling	0	1	0	0	1	1	0	1	**4**
d.	Cohesion	0	1	0	0	0	0	1	1	**3**
e.	Autonomy	0	1	1	0	0	1	0	0	**3**
f.	Maintainability	0	1	0	0	0	1	1	0	**3**
g.	Reusability	0	1	0	0	1	1	1	0	**4**
h.	Scalability	1	1	0	1	1	1	1	0	**6**
i.	Network Complexity	1	0	0	0	1	1	0	0	**3**



Question 1.4

"Suppose that you do not have the agile culture like continuous integration and delivery automation and also cloud infrastructure. How will it affect your decision regarding microservices and their size?"

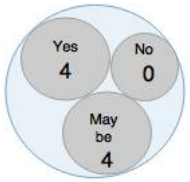
Response 1.4	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No microservices at all, start with Monolith, extract one microservice at a time as automation matures	1	1	0	1	1	1	1	1	**7**
Start with bigger sized microservices with high business value, low coupling and single Responsibility	0	0	1	0	1	0	1	0	**3**



Question 2.3

"Do you think the table of basic metrics can be helpful?"

Response 2.3	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No	0	0	0	0	0	0	0	0	**0**
Yes	1	0	1	0	1	0	1	0	**4**
Maybe	0	1	0	1	0	1	0	1	**4**

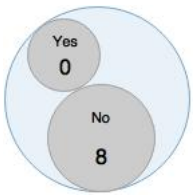


3. Process to design microservices

Question 3.1

"Do you follow specific set of consistent procedures across the teams to discover microservices?"

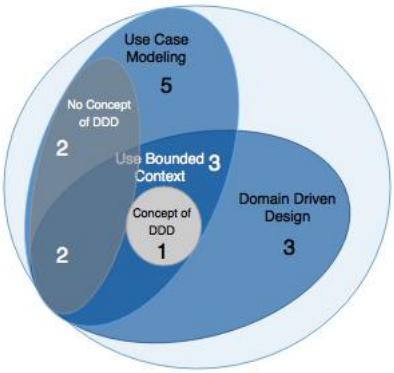
Response 3.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No	1	1	1	1	1	1	1	1	**8**
Yes	0	0	0	0	0	0	0	0	**0**



Question 3.2

"Can you define the process you follow to come up with microservices?"

Response 3.2	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	Score
By brain-storming to divide a usecase into finer functionalities considering business value, cohesion, loose coupling [Single Responsibility], autonomy, reusability and scalability.	1	0	1	1	0	0	1	1	**5**
Have you heard of Domain driven design and bounded context?									
Yes	0	0	0	0	0	0	1	0	**1**
Is the concept provided by bounded context being used?									
Yes	1	0	1	0	0	0	1	0	**3**
By identifying bounded context using domain-driven-design Understand the problem domain interacting with domain experts and studying the interaction among domain models and flow of data.	0	1	0	0	1	1	1	0	**4**



1.4.3 Interview Reflection on Hypothesis

The data gathered from the interviews which are compiled in the section 1.4.2 can be a good source to analyse the hypothesis listed on the section 1.4.1.

Hypothesis 1

The correct size of microservices is determined by:

1. **Single Responsibility Principle**
2. **Autonomy, and**
3. **Infrastructure Capability**

The response from question 1.4.2 has helped to point out some important constraints which play significant role in industry while choosing appropriate size of microservice. It is also interesting to notice how the terms in answer closely relate to the hypothesis 1.4.1. The figure attempts to clarify the relationship among the terms with the hypothesis.

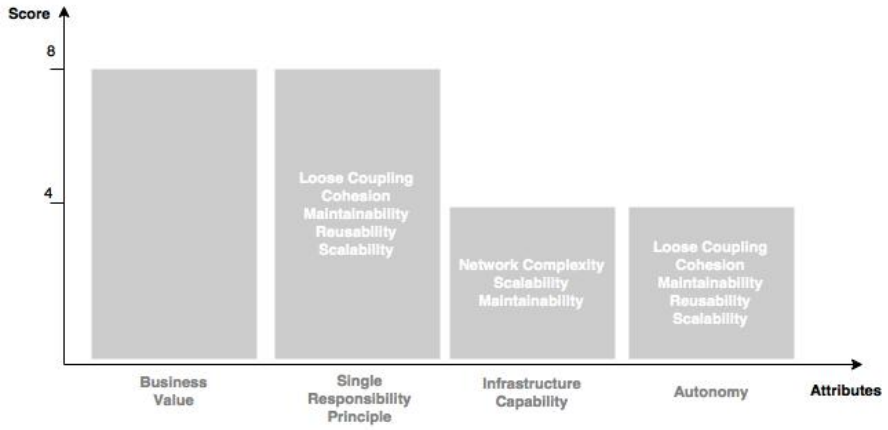


Figure 1.2: Attributes grouping from interview response

The interview unfolded various quality attributes which determine the correct size of a microservice. Moreover, these quality attributes can be classified into three major groups which are:

1. Single Responsibility Principle
2. Automony
3. Infrastructure Capability

Additionally, the response from the question 1.4.2 highly suggest the significance of availability of proper infrastructure to decide about size of microservices and the microservices architecture. This highly moves in the direction to support the hypothesis. Finally, there appears one additional constraint to affect the size of microservices, not covered by hypothesis but mentioned by all interviewees, which is the business value provided by the microservices.

Hypothesis 2

Domain driven design is the optimum approach to design microservices.
From the response to question 1.4.2, it can be implied that the organization has no consistent set of specific guidelines which are agreed and pratised across all teams. However, the various reactions to the question 1.4.2 appear to fall with two class of process. The response from 1.4.2 can be represented as in figure 1.3 to make it easy to perceive.

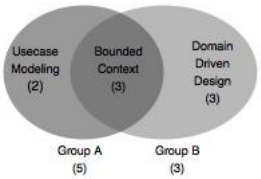


Figure 1.3: Process variations to design microservices

There is one group of reactions which follow a process to functionally divide a usecase into small functions based upon various factors such as single responsiblity and the requirements of performance. This strongly resemble to the usecase modeling technique as described in the chapter **??**.
Moreever, within the same group, a partition are also using the concept of bounded context to ensure autonomy, which also suggests towards the direction of domain driven design as mentioned in the chapter **??**. It is interesting to find out that the rest of the teams who were not using bounded context, also have no idea about the concept. It may be assumed that they will have different view once they get familiar with the concept.
Finally, the rest of the interviewees agree on domain driven design to be the process to design microservices. So, the response is of mixed nature, both usecase modeling and domain driven design are used in desigining microservices. However, it should also be stated that domain driven design are used on its own or along with usecase modeling to design autonomous microservices.

1.5 Interview Summary

Granularity of a microservice is considered as an important aspect when designing microservices architecture. The first impression in major cases is to make the size of microservice as small as possible however there are also various attributes and concepts which affects the decision regarding the size of microservices. The major aspects are single responsibility principle, autonomy and infrastructure capability.
The single responsibility gives impression to assume the size of a microservice as small as possible so that the service has minimum cohesive functionality and less number of consumers affected. Whereas, in order to make the microservice autonomous, the service should have full control over its resources and less dependencies upon other services for accomplishing its core logic. This suggests that the size of the service should be big enough to cover the transactional boundary upon its core logic and have full control on its business resources.
Moreover, the small the size of the service is, the number of required microservices increases to accomplish the functionalities of the application. This increases network complexity. Additionally, the logical complexity as well as maintainability will increase. Undoubtedly, the independent scalability of individual service will also increase. However, this increases the operational complexity for maintaining and deploying the microservices, due to the number of services and number of environment to be maintained for each services.
The research findings as well as interview response leads to the idea that the question regarding "size" of a microservices has no straight answer. The answer itself is a multiple objective optimization problem whose answer depend upon the level of abstraction of the problem domain achieved, self-governance requirement, self-containment requirement and the honest capability of the organization to handle the operational complexity.
Finally, there is an additional deciding factor called "Business Value". The chosen size of the microservice should give business and competitive advantage to the organization and closely lean towards the organizational goals. A decision for a group of cohesive functionalities to be assigned as a single microservice means that a dedicated resources in terms of development, scaling, maintainance, deployment and cloud resources have to be assigned. A rational decision would be to relate the expected business value of the microservice against the expected cost value required by it.
The teams do not follow any quantitative metrics for measuring various quality attributes such as granularity, coupling and cohesion. According to the reactions from interviews, the basic metrics to evaluate the quality attributes visualized in the table **??** can be helpful to approach the design of microservices architecture in an efficient way.
There is no single consistent process followed across all the teams. Each team has

its own steps to come up with microservices. However, the steps and decision are highly influenced by a set of YaaS standard principles 1.3. The reactions from the interviewees are already visualized in 1.3. The process followed by a portion of teams closely relate to usecase modeling approach, where a usecase is divided into several smaller abstractions guided by cohesion, reusability and single responsibility principle. Within this group, a major number of teams also use the concept of bounded concept in order to realize autonomous boundary around the service. Finally, the remaining portion of the teams closely follow domain driven design, where a problem domain is divided into sub-domains and in each sub-domain, various bounded contexts are discovered, which is then mapped into microservices.
The domain driven design can be considerd as the most suitable approach to design microservices. Following this approach, not only the problem domain is functionally divided into cohesive group of functionalities but also leads to a natural boundary around the cohesive functionalities with respect the real world, where concept as well as differences are well understood and the ownership of business resources and core logic are well preserved.

1.6 Example Scenario

YaaS provides a cloud platform where customers can buy and sell applications and services. In order to accomplish this, YaaS offers various basic business and core functionalities as microservices. The customers can either use or extend these services to create new services and applications focusing on their individual requirements and goals.
The following part of this section discuss the steps used to identify the basic microservices in YaaS.
Step 1:
The problem domain is studied thoroughly to identify core domains, supporting domains and generic domains.

#	Sub-domain	Type	Description
1	Checkout	Core	handles overall process from cart creation to selling of cart items.
2	Product	Supporting	deals with product inventory
3	Customer	Supporting	deals with customer inventory
4	Coupon	Supporting	deals with coupon management
5	Order	Supporting	handles orders management
6	Email	Generic	provides REST api for sending emails
7	Event	Generic	provides asynchronous event based notification service
8	Account	Generic	handles users and roles management
9	OAuth2	Generic	provides authentication for clients to access resource of resource-owner

Table 1.2: Sub-domains in YaaS

Step 2:
The major functionality of the commerce platform is to sell products, which makes 'Checkout' the core sub-domain. The sub-domain is responsible for a bunch of independent responsibilities. The individual independent functionalities can be rightfully represented by respective bounded contexts. The bounded contexts are realized using microservices. The other major reason for them to be made microservices, in addition to single independent responsibility, is different scalability requirements for each functionality.

#	Bounded Context	Description
1	Checkout	handles the orchestration logic for checkout
2	Cart	handles cart inventory
3	Cart Calculation	calculates net value of cart considering various constraints such as tax, discount etc.

Table 1.3: Bounded Contexts in Checkout

Step 3:
The supporting sub-domain Product deals with various functionalities related to "Product".

#	Bounded Context	Description
1	Category	manage category of products
2	Product	manage product inventory
3	Price	manage price for products

Table 1.4: Functional Bounded Contexts in Product

The bounded contexts listed in table 1.4 are based upon the independent responsibility and different performance requirement. Additionally, each functionality deals with

business entity "Product". However, the conceptual meaning and scope of "Product" is different in each bounded context. "Product" is a polyseme.
Again, there are two more functionalities identified. The first one is a technical functionality to provide indexing of products so that searching of products can be can be efficient. The functionality is generic, technical and independent. It has a different bounded context and can be realized using different microservice. Furthermore, there is a requirement of mashing up information from product, price and category in order to limit the network data from client to each individual service and improve performance. This is realized using a separate mashup service. Also, these functionalities have different performance requirement.

#	**Bounded Context**	**Description**
1	Algolia Search	product indexing to improve search performance
2	Product Detail	provide mashup of product, price and category

Table 1.5: Supporting and Generic Bounded Contexts in Product

The same process is applied for other subdomains to identify bounded contexts within.It is important to notice that the decision to realize the functionalities as microservices not only considered various technical aspects as performance, autonomy etc but also completely different aspect as "business value". The individual microservices such as Product, Category, Checkout have high business value to YaaS for the reason that they can be sold within a package.
The various microservices designed following the process as mentioned forms a layer of services based on their level of abstraction and reusability. The layers are shown in figure 1.4. The core services are technical generic services used by business services and form the bottom layer. On the top of that are the domain specific business services focused on domain specific business capability. The individual business functionalites provided by business services are then used by Mashup layer to create high level services by orchestrating them.

Figure 1.4: Microservices Layers

The figure 1.4 uncovers only microservices at the backend related to two separate subdomains. The complete architecture covering other domains such as Marketing etc and all layers including User- Interface, is shown by figure 1.5.

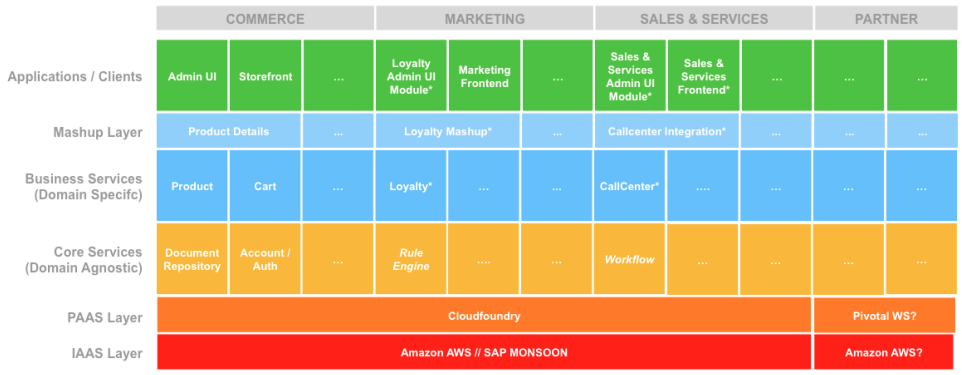


Figure 1.5: Hybris Architecture

Acronyms

API Application Programming Interface.

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.

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

[Bec+11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. *Manifesto for Agile Software Development*. 2011.

[Bla08] J. Blanchette. *The Little Manual of API Design*. June 2008.

[Blo16] J. Bloch. *How to Design a Good API and Why it Matters*. Jan. 2016.

[Bon+14] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. *The Reactive Manifesto*. Sept. 2014.

[FL14] M. Fowler and J. Lewis. *Microservices*. Mar. 2014. URL: `http://martinfowler.com/articles/microservices.html`.

[Hir15] R. Hirsch. *YaaS: Hybris' next generation cloud architecture surfaces at SAP's internal developer conferences*. Mar. 2015. URL: `http://scn.sap.com/community/cloud/blog/2015/03/03/yaas-hybris-next-generation-cloud-architecture-surfaces-at-sap-s-internal-developer-conferences`.

[Mar13] S. Martin. *Effective Visual Communication for Graphical User Interfaces*. Jan. 2013.

[Mar16] R. C. Martin. *The Single Responsibility Principle*. Jan. 2016. URL: `http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle`.

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

[Por16] J. Porter. *Principles of User Interface Design*. Jan. 2016.

[RS07] P. Reldin and P. Sundling. *Explaining SOA Service Granularity– How IT-strategy shapes services*. Tech. rep. Linköping University, 2007.

[Sol12] K. Sollenberger. *10 User Interface Design Fundamentals*. Aug. 2012. URL: `http://blog.teamtreehouse.com/10-user-interface-design-fundamentals`.

[Sti14] M. Stine. *microservices are solid*. June 2014. URL: `http://www.mattstine.com/2014/06/30/microservices-are-solid/`.

[Stu15] A. Stubbe. *Hybris-as-a-Service: A Microservices Architecture in Action*. May 2015.

[Wig12] A. Wiggins. *THE TWELVE-FACTOR APP*. Jan. 2012. URL: `http://12factor.net/`.