Domain Driven Design Notes

Contents

List of Figures

Chapter 1

Microservices Architecture

1.1 Pre-Introduction

At present these notes are taken from a Udemy course named Domain Driven Design for Architects.

1.2 Business and Digital Transformation

Business transformation is an umbrella term that is used for referring to fundamental changes in how an organisation conducts its business. See examples.

1.2.1 Why do businesses need to transform

> Environmental changes - new regulations may force the organisation to change how they do business. Competitive pressure - think of an organisation that is dealing with a competitor that is rolling out innovative products at a very rapid pace.

What is the choice for these organisations? They must transform. They have to think of new products. They have to think about the speed at which they can roll out these new products.

- New opportunities - Organisations may have to transform themselves to integrate with new technology. This often requires serious transformation initiatives. - Customer demands - expectations are continuously changing in order to maintain and expand their customer base. Organisations need to adjust their business to meet their customers demands and expectations. Businesses that ignore their customers expectations tend to loose out to the competitors.

1.2.2 Digital Transformation

The process of using digital technologies to meet the needs of transformed business processes and to create innovative customer engagement mechanisms. The relationship between digital transformation and business transformation is that the digital transformation supports the business transformation initiatives (ahem) The value in this is that it allows for organisations to change at a very rapid pace. Businesses that fail to transform fail to survive. Netflix vs Blockbuster

Transformation is not a one time initiative. Businesses need to change on a continuous basis and this requires rapid changes to their systems and applications. Organisations must keep up with the pace of new and evolving technologies.

1.2.3 Common problems

A common challenge that businesses face in their transformation journey is that the old ways of building software hinders or makes it difficult for organisations to transform. It is slow to build software using the older technologies and architectural paradigms. Older technologies and older ways of building applications to integrate with newer digital technologies This is where microservices architecture can help. Microservices architecture addresses these challenges and helps the organisations move at a faster pace to achieve their transformational objectives. Transformation is about rapid changes, and in the case of microservices, changes are isolated to a set of microservices. Changes to a small microservices will be much faster than changes to a monolith where there are a lot of interdependencies between multiple modules realising the different business funtionalities.

1.2.4 Summary

Organisations need to continuously transform. The requires IT systems to change at a rapid pace. There is a need for rapid adoption of new digital technologies and speed to market is key. Microservices architecture helps the organisations meet these requirements from IT perspective.

1.3 A Business Perspective of Microservices

Once the microservices have been identified in a microservice application, each of the microservices is assigned to a small team. These small teams build and operate that microservice. The members in these teams bring different skills to the table, and these teams build different skills to the table and these teams are supported by the domain experts.

Common question: what should the size of a microservices team be? Two pizza team. (some people need to get some salt with their pizza lol) The idea is that there is better collaboration among smaller teams, which leads to frequent software releases. This in turn helps the organisation respond faster to changes in business. Overall, this will lead to the technology becoming a competitive edge for an organisation. (Worked for amazon apparently)

1.3.1 Benefits of microservices

The first benefit of organizing microservices around business capabilities is that each service can evolve independently.

1.3.2 Business perspectives of changes in a monolith

Means that this coordination between the various teams will slow down the process of making changes to different parts of the application from the business perspective. Slow to release new products in the market. This problem is addressed with microservices. This monolithic architecture, when replaced with microservices architecture will look something like this, wherein each of the capability will be realised in an independent microservice. So what this means is that changes can be performed each of these services independently and that translate into a business benefit wherein faster responses can be achieved to changes in business environment.

Microservices architecture allows the business to make radical changes to how it operates. As long as new microservices maintaining the same contracts as the old microservice, there will be no change required in other microservices.

The next benefit is that it makes it easier for the IT teams to understand the business. Overall IT teams don't need to dive deep into all business capabilities. They can focus on the business capability that they are building in their microservices with microservices built around the business capability. The IT teams are able to achieve higher alignment with business priorities. What this boils down to is that since each of the microservices team is operating independently, they do not spend time on managing the conflicting business priorities. This will lead to faster speed to value for the business.

1.3.3 Summary

Summarise discussion. Businesses need to stay competitive by way of rapid transformations and this rapid transformations require support from the IT teams to deliver value to the market at a faster pace. Microservices architecture is an enabler or a catalyst for continuous business transformation as it helps the IT teams to move at the same speed as the business. One important thing to keep in mind is that to get the most benefit from the microservices architecture, it is critical for the microservices teams to carve out of an appropriate business scope for each of the microservices. If not done correctly, it will lead to teams being interdependent and that will lead to loss of advantages of microservices architecture. This is where domain driven design comes into picture. The domain driven design bounded context is a representation of the business scope for the microservice.

Key points

> **Note:**
>
> Smaller teams translate to faster speed to market
>
> Microservices are organised around business capabilities and the benefit of this approach is that enables the IT teams to operate independently.

1.4 A Technical Perspective of Microservices

Microservices architecture suggests creation of loosely coupled set of services for building applications. These services interact with each other over a network by way of a lightweight protocol such as HTTP. Each of these services have an independent code base, i.e that means that they can be independently deployed. Teams owning these services are empowered to make decisions and what that means is that there is no centralised governance. Teams can make their own decisions on what works best for the services they own. Each of the services have a very defined scope from the business perspective.

Loose coupling in the context of microservices. Loose coupling means that there is minimal dependency between the microservices. A consumer microservice invokes an interface on the provider microservice, the consumer microservice has knowledge only about the external interfaces exposed by the provider microservice. Microservice has no knowledge of the internal implementation of the provider. Microservices are over the network protocol. As a result, there is code level dependencies between microservices. Microservices expose there external APIs.

These APIs are commonly implemented as RESTful Services or GraphQL APIs which are made available to other microservices over the HTTP protocol. Apart from HTTP, asynchronous messaging mechanism is also used for building the interactions between microservices. It is common for the microservices to interact with other microservices using request reply pattern as well as publish subscribe messaging pattern.

Kafka, RabbitMQ and ActiveMQ are some of the commonly used technologies for building these messaging based interactions.

1.4.1 Advantages of microservices architecture compare to monolithic applications

- It is easier to manage changes in a microservice when there is a change in one services. - There is no impact on other services within the application since the codebases for each of the microservices is independent, little or no coordination is needed between the teams and the refression. - Testing needs to be carried out only for the microservice case that is getting changed. - Other microservices need not be tested. - Deployment for each of the microservices may be carried out independently. - Each team decides on the frequency of deployments, depending on their requirements and other constraints. I.e teams do not follow a common deployment plan, which is very common in the case of monolithic applications. - This independent deployments lead to higher productivity and faster delivery of software.

- Talk about polyglot microservices. — The idea behind polyglot microservices is that the team owning the microservice may decide on the technology stack for their service.

Experts caution against the use of multiple languages, as this may lead to challenges. One of the biggest differentiators of microservice architecture is the fact that there is failure isolation. – What this means is that failure in one service will not bring down the entire system, which a common scenario in the case of a failure of a component in a monolithic application.

Another benefit of microservices is that each of the services may be scaled independently.

1.4.2 Disadvantages of microservices.

Since microservices interact with each other over a network protocol, an application built with microservices architecture may exhibit poor performance compare to the same application implemented with a monolithic architecture. In a microservices application, each microservice manages its own database. This leads to complexity in managing the data integrity. - The reason for that is that in the case of monolithic application, you may use a common database and you may use local transactions to manage the data integrity. - In the case of a distributed architecture like a microservices architecture, traditional transactional mechanisms may not work and this leads to higher complexity. - At runtime, microservices are launched as independent processes. - These independent processes need to be monitored. - If you have an architecture where you will need tens of hundreds of instances of the same microservices, it may become challenging to monitor these microservices and to debug the microservices in case of issues.

Another common concern for microservices is that since microservices exposes interfaces in the form of APIs that lead to an expanded attack surface for the microservices based application (hmmmmmmmm hide behind auth/load balancer) To address these disadvantages organisations planning to adopt microservices need to invest in new technologies in terms of infrastructure tools, and then they also need to invest in skills development. - This means that the organisations may need to make upfront investments for an application that will be built with microservices architecture.

Summary - Microservices from the technology perspective Pros - Architecture change management becomes easier and deployments can be carried out independently, this means that features can be released much faster. – Speed to market is increased, failures are isolated and services may be scaled independently, what this means is that it leads to a better quality of experience for the consumers of the application.

Cons - From the cons perspective, poor network performance is a concern. (containers vs pods) - There are challenges related to monitoring data management and security. – Some of these challenges may be addressed by way of investment in tools, infrastructure and skill development.

1.5 Adoption of Microservices Architecture

Two terms green field microservices and brownfield microservices.

Adoption of Microservices architecture requires an organisation to acquire resources with new technology skills. They need to invest in technologies such as cloud containers and a number of tools for building and operating microservices. Organisations need to change their processes.

Example: They need to build their DevOps practices. - Older ways of managing IT resources and applications does not work for microservices applications. - An organisation may need to change its culture, for example faster decision making. —- This won't happen overnight.

Successful adoption of microservices architecture requires commitment from the business and IT leaders. The role of an architect is to guide and educate the business and IT teams in terms of cost and benefit of adoption of microservices architecture.

It is important for the architect to not just talk about the technical benefits of microservices but spend time building a business case for microservices adoptions for their specific organisation. The idea is that it will be easier for the architect to get the support from the IT and business leaders if there are benefits of adopting microservices.

The business case To bulid a business case one has to think about the business impact and the technology. As an architect thing about the specific business for your organisation. — It is going to improve the customer experience at a rapid pace. — Is it going to lower the cost of IT operations. — Is it going to generate new revenue streams. — Is it going to give your organisation a competitive advantage

Specific business advantage will depend on your organisations business. As an architect you must understand your organisations business to be able to map the benefits of microservices architecture to your organisations.

Tech focused architect – Can release software every 6 weeks vs every 3 months – Monolithic applications are difficult to change hence adopting new technologies will be slow — Invest in microservices in order to move faster Business focused architect – Can help the business cut down product development process to six weeks which is 50% faster than out competitor. – Adoption of new digital technologies can help the business achieve the goal of increasing the lifetime value of our customer. ——- As microservices architecture provides a foundation for faster adoptions of these new digital technologies.

The second one will appeal more to the business stakeholders. The idea is to resonate with the business and IT leaders by projecting the business benefits of microservices rather than just describing the technical benefits of microservices.

5 to 7 slides vs 50 pages long formal document..for business stakeholders.

Author example of building a business case. - Make clear message about business value - If possible it needs to be quantified —- Need to use the right numbers to be able to create an impactful statement from the business value perspective. - Put together a roadmap on how you expect the organisation to adopt microservices. — Important to indicate time to value. — Would it take you three months to deliver value or would it take 18 months to get there? — It is important for it to be conveyed clearly, to set the right expectations. Need to describe what you need to be successful. —- Looking to get commitment from your IT and business leadership. —- They can't commit to you unless what they are getting into and depending on where you

are in your microservices journey. Do a POC. — Demonstrating the value with something live is always helpful.

Consider what works for an organisation and the put together your business case becasue that will have a higher change of success that using a generic format.

There are two types of microservices projects. Brownfield microservices projects are where the existing monolithic application needs to be converted into microservices. Greenfield microservices projects whereas a new application needs to be built from the ground up.

A brownfield project team has to deal with legacy technologies and tech debt. They have two options. - First options is to refactor the application. That is to convert the monolithich application into microservices. – This can be carried out by way of a big bang approach where all the microservices are built in parallel or the organisation can take an evolutionary approach where they build the microservices by peeling off parts of the existing monolith application. – The other options is to completely rebuild the brownfield application from the ground up.

The greenfield project team has two options for building their application. The first one is they can build their application as a microservice application from the get go. A couple of considerations that the greenfield project team needs to keep in mind.

First one is that they need to ensure that there are tools and technologies available to the for building these microservices adn the organisation. Readiness also play an important role. – Does the organisation have mature DevOps practices and processes. – This options is suggested for teams tha tare experienced with microservices and are working for organisations that have already adopted microservices.

Second approach is to use the monolith first approach. The monolith first approach suggests that hte greenfield application team create a well-designed monolith application. - Gain some experience with the application an then peel off parts of the Monolith application to create appropriate microservices.

Key points - As an architect working on microservices applications you must think about the specific business benefit of microservices to your organisations. - There are two types of microservices projects. – In a brownfield microservices project an existing monolithic application needs to be converted to microservices. — Two options for this kind of project. — First one is to refactor the existing Monolith application to microservices application — The second type is the greenfield project in which the project team does not have any kind of legacy debt to deal with.

There are two options for greenfield projects. - Greenfield projects may be implemented with microservices architecture from the ground up - Or the project team may take the monolith first approach, wherein they first create a monolith application and then convert it to microservices.

Chapter 2

Introduction to Domain Driven Design

2.1 Domain, Sub-Domain and Domain Experts

A domain is defined as a sphere of knowledge, influence or activity. The business perspective - a domain represents the field or industry in which the business operates. The software perspective - domain may be though of as the representation of the problem space for that software. The domain is made up of multiple sub domains.

For large and complex domains, it is virtually impossible for a domain expert of know everything about the domain (challenge accepted :D). As a result, there are multiple subject matter experts or domain expert within a domain, mostly aligned with the sub domains in the bigger picture.

Useful questions: - What domain do you operate in? Think business domain not technology domain. - What are the sub domains within the domain that you operate in? - List out the domain experts that you work with.

Summary - Domain is defined as a sphere of knowledge, influence or activity. - Domains are made up of multiple sub domains. - It is not possible for a single expert to have a through knowledge of all the sub domains in a complex large domains (MVK?) - As a result multiple domain experts are needed to support business functions. - Mostly, these domain experts are aligned with the sub domains within the larger domain

2.2 Conceptual Models, Architural Styles

Conceptual models are defined as a representation of a system made from a composition of concepts. Key idea here is concepts. I.e that focus is not on the physical parameters of the system.

The focus here is on the critical components that make up the system. This information is not sufficient for the engineer in the factories to be able to build the car.

Whats the purpose? Multiple benefits to of starting with conceptual models. The first one is that it enhances the understanding of the designers. As the designers put together the concept and think through it, they may find flaws in the design or the may find opportunities for improvement. —¿ A good conceptual model will lead to a better product. Second one is that it makes it easy to convey the ideas behind the concept to the stakeholders. Third one is that conceptual models provide a point of reference to create detailed specifications. This model contains some information about the physical aspects of the end product. This information, combined with the models layout can act as the starting point for the designers.

The fourth one is that conceptual models provide documentation for future reference. The first thing they do is identify the core concepts that need to be put together to design the final

product. They come up with the common terminology for the domain concepts used in the model. They identify the different parts of the system.

Also they will identify the different parts of the overall product of the system. Next, the designer identifies and documents the relationship between the various domain concepts, and they also identify the critical or the foundational parameters for these concepts as well as the relationships in the banking domain.

Architectural model - the visualisation of the system represented by the model.

A more formal definition of the architectural model or architecture is that is is a structured representation of a solution that meets the requirements in the problem space. It is a high level abstraction of parts of the end solutions that presents a view or a perspective on how the required payments will be met and it assists in answering the questions posed by different stakeholders.

What is the difference between architectural model or architecture and design?

The difference is in the level of details and focus.

Architecture is mostly high level. Provides a skeleton for the end product and the focus is more long term concepts. Components that may not change as freqeuently or never in the lifetime of the product. The design is relatively detailed and the focus is on implementation. This is relative. Design can also be high level but they are relatively detailed compared ot the convetional or traditional architecture models.

Summary 1 - Conceptual models - a set of concepts and their relationships 2 - Architectural models are a structured representation of solution that meets the requirements from the problem space. Does not have any implementation details 3 - Whereas design is a structured representation of a solution that has some level of implementation details in it.

2.3 Modelling Techniques and Architectural Styles

Author opinion - Three guiding principles - Think about the purpose of the model and the audience for whom the model is being created for. Think about the prespective and the viewpoints of the audience. - The idea is to think about their interests and their concerns. - Will the model that I'm creating address their concerns and answer their questions? - Third one is level of details. How detailed should my model be so that it provides most value to the target audience.

Many techniques for creating these models (hmmmmm)

Four plus one architectural view model describes the architecture from the viewpoints of multiple stakeholders. Central to this technique is the idea of scenarios of use cases. Think of these as requirements. These scenarios are used as guide for creating multiple views and these scenarios are also used for validating architecture.

Logical view focuses on the functionality or the capabilities exposed by the system to the end user. The process view, as the name suggests explains the processes in the system and how these processes interact with each other. The development view illustrates a system from a programmers perspective and is concerned with software management. This view is also known as the implementation view. The last one is the physical view that depicts the system form the point of view of the engineers. It is concerned with the topology of software components on the physical layer as well as the phyiscal connetion between these components.

Why call it four plus one? The reasons is there are four views and plus on is for the scenarios. A large software project involves multiple stakeholders. They may be tens or even hundreds of stakeholders involved in the project. These stakeholders have different set of interests and concerns.

Business Exec - Their interest is in understanding the value that the system will provide to the end user and how that end user value will translate into value for the business. Business experts - important that the business implements the right set of processes and these processes are accurate. Their interests will be in the process. Developers - they are concerned

with the implementation of the system. They are concerned about the deployments and the management of the software. As a result their interest is in the development view. Network engineer - are not concerned with he development view or the logical view or the process view. They would like to understand how the various servers that will host the applications or the components will talk to each other over the network. Architecture view - finally, the architect is responsible for creating these views and ensuring that these views rae providing the most value to each of these stakeholders.

This is just a small set of stakeholders. There may be many more stakeholders playing different roles in the software initiative.

Practice - List out the stakeholders in your organisation and think about which views will be most appropriate for them.

UML Created by Object Management Group. Provides a standard set of diagrams for architectural modelling and these diagrams are created by using a standard set of notations. Vast subject. See other resources if required. Latest version of UML consists of 14 diagrams which can be used to created the 4 + 1 architectural view model.

Author preference. - Use case diagram for depicting the scenarios for logical views - Use the state diagram and the class diagram for process view. - Use sequence diagram and activity diagram for development view - For development view use the component diagram and package diagram. - For physical view deployment diagram works out the best.

Software architectural style may be through of as a reusable architectural pattern which may be used as a solution to a commonly occurring problem. There are multiple architectural styles and these architectural styles are categorised based on the key focus area. The service orientated architecture and the message bus architecture falls in the category of architectural style that focuses on the communication between components. The layered architecture and the object orientated architecture and design are the common styles use when the architect is focusing on the structure of the system. The client server and the three tier architectural styles fall in the category of deployment, wherein the focus is on the deployment of the various components that make up the system. The database centric design and the data flow diagrams are styles used where the architect is focusing on the core data within the business domain.

The last architectural design is DDD where the focus on the business domain rather than the technolgoy.

Summary - Architects create models using different modelling techniques. - Two such techniques. The four plus one architectural view model and unified modelling langauge. - The second may be combined to create very effective architectural models. - There are multiple architectural styles. - Each of these follow a set of common principles and they focus on specific aspects of the system. - Architects may adopt any of these styles depending on their needs and preferences. - Domain driven design is an architectural style in which the focus in on the business domain.

2.4 Domain Models

Reasons businesses invest in software is to solve business problems. Business problems may be defined as current or long term challenges and issues that may prevent the business from achieving its goals. These goals may be short term or they may be long term. It is important for the architects to understand the business problem. The architect must understand the domain first and for that the architects create domain models.

A domain model is defined as organised structured knowledge of the domain that is relevant for solving a business problem Important: Organised and structured knowledge.

The domain model consists of multiple parts. Key concepts are the foundational concepts related ot the domain. Domain vocabulary - consists of common terms and their definitions

used by the stakeholders when they are discussing the problem space wihin that domain. This ensures that all stakeholders have a common understanding of all the terms used in that domain.

Think of domain entities as domain objects that have a unique identity. The attributed in the domain objects may change over the lifetime of the object.

In the real world, domain entities have a relationship with other domain entities and the model captures these relationships.

Businesses use defined processes for carrying out the operations and these processes are documented within the domain model by way of workflows and activities.

Important thing to keep in mind is that the domain model captures structured knowledge that is used for solving a business problem. In fact the domain model may contain additional knowledge by they way of visual depictions or diagrams and textual documentation.

As the creator of the domain model, you are in control of what should go in the domain model to make it as effective as possible.

There are no special tools for creating domain model of visualisations. May use any typical tool that supports UML modelling for textual documentations.

Stakeholders working on the domain models may decide on the tools that will work out best for them.

Key points Domain model is organised and structured knowledge about the domain. The purpose of the domain model is to help with creating a sutions to business problems within that domain.

The five elements. - Domain vocabulary - Domain entities - Relationship between the entities - Workflows and activities - Key concepts

2.5 Modelling Techniques and Architectural Styles

Enterprise domain models - also known as aggregate or unified domain models.

In a complex industry it is hard to find one expert who knows everything.

To get the domain knowledge, the teams must work with these domain experts as this knowledge is mostly not documented anywhere but is in the heads of these domain experts. Domain knowledge is organised and structured knowledge about the domain and these domain models are created by way of a process referred to as knowledge crunching.

The team receiving the knowledge form the domain expert or experts analyses the received information and knowledge and creates the domain models.

Typically the knowledge crunching process is spearheaded by the technology team that works very closely with the domain experts to create that structured domain knowledge or the domain models.

it is common for the technology team to be led by an experienced technologist. (Some kind of IT lead)

Ideally the IT lead should have some prior experience with domain modelling exercise and it is not required that they are an expert in specific technolgies.

Opinion - Successful leads have a breadth of knowledge in multiple technologies and they are also open to learning busienss related topics. Rest of the technology team may be composed of team members with different skill sets and roles.

Idea is that these models could be used by software development teams to build robust well documented systems. These enterprise modesl were referred to as the unified models or aggregate models. The idea behind all these are the saem.

Intended software development process looks something like this. The software development teams will focus on specific areas within the enterprise domain model and then carry out the technical design. Code and then build the final product. In theory this sounds good but in reality this process is marred with multiple challenges.

First challenge is the creation of the model itself. Enterprise models are inherently complex due to the scope and szie and the fact that multiple experts need to be engaged in order to create such models. The second one is that its hard to keep the models up to data as there is not single owner for the mode and it falls on the IT team to manage the model as well as the software product..hmmmmmm

Unfortunately after a while due to product delivery priorities, models start to fall behind the actual implementations and loses its value.

The third one is that there are linguistic challenges when you try to merge together knowledge about multiple domains into one single model.

Lol - It is very common to see the same business term have different meanings in different sub domains within the organisation. These linguistic challenges can cause big confusion for the software development team as well as the domain experts. An important point to note is that these challenges are no applicable to enterprise level domain models but to any domain model that has to deal with complex domains.

Pros of DDD - Domain driven design approach provides principles and patterns to address the challenges faced with developing complex domain models.

Summary - Knowledge crunching refers to the process of creating the domain model from the knowledge gathered from the domain experts. - There are multiple challenges with creating models for complex domains and these challenges are addressed by the domain driven design approach.

Chapter 3

Understanding the Business Domain

3.1 Why understand the business

Every modern business is a technology business. For some it is the core model. For others, technology supports the core model. Irrespective of the core business model technology is an essential part of business. The goal of IT teams is to help the business achieve the business goals, hence IT teams must understand the business model. Understanding of the business model by IT teams will lead to faster delivery of value to the business. It will lead to more active participation of the IT teams in the business decisions and it will align the IT objectives with the business objectives. The most important outcome of having teams of having teams understand the business is that business start to think of it as a trusted partner rather than just a service provider. What this essentially means is more value to the business, move value to the customers of the business.

important Architects and developers etc who understand the business earn the trust of their peers. I.e don't just learn new techologies, spend time understanding the business.

3.2 Introduction to Business Model Canvas

Business model canvas is a tool that helps discuss, communicate, design and understand the organisations business model. The benefit of the business model canvas is that the entire business model may be depicted in one image.

To create the business model canvas, one has to focus on nine core areas of the business and these are organised in the form of a canvas.

Do own research

3.2.1 Building blocks

- Nine building blocks - The first building block is customer segments. Customers are the reason the business exists, so one has to think carefully about who are the customers. - The next building block is value proposition. What kind of value are we providing to each of these customers. - The next one is key resources. Emphasis on key, there are many resources that our business requires but you need to think about those resources which are essential for value proposition. - There may be multiple resources that the business needs, but think about the key ones which without the business cannot exist. - Next one is the key partners who are suppliers of the key resources to the business. - Next one is technology providers. Consider procuring technology from other suppliers or partners. - Also need to consider appropriate permissions to operate. Without them a business will not be able to operate. - Businesses need to carry out

multiple activities. Underkey activities, you need to think about the activities that the business carries out to create the value for the customer.

- Customer retention is one of the most important things for any business, and to retain a customer you need to ensure that the customers are happy with the services you are providing and the relationship you have with them. - Under the customer relationships, one has to think about the type of relationship that is offered to each of the customer segments. - The cashflow revenue stream depicts the inflow of revenue for the business for what does. - The cost structure has the depiction of the cash outflow. These are the expenses incurred by the business on carrying out the key activities. - Next one is the channels by which the customers would like to be reached.

Chapter 4

Domain Driven Design: Strategic Patterns

4.1 Introduction to DDD and Patterns

Domain driven design is an architectural approach that provides principles and patterns to address challenges faced with developing complex domain models. Term originates from from Eric Evans, Domain Driven Design book.

Domain driven design emphasises on the fact that the IT project's primary focus should be on the business domain and business logic rather than technology. Promotes the idea of breaking the unified model into smaller, more manageable models and it suggests the use of iterative process for improving the model to solve a domain problem.

Domain driven design offers two types of patterns.

Strategic patters are used for dividing a complex and large business problem into smaller chunks with well defined boundaries.

Tactical patterns are used for translating the conceptual models to software application and service designs.

The strategic pattern applies across multiple bounded contexts, whereas tactical patterns are applicable only within a bounded context.

The idea is that domain driven design approach leads to smaller, independent domain models that can be built can be built as highly decoupled and independent set of microservices.

4.2 Business Subdomain Types

A business domain is composed of multiple subdomains. The type depends on the value a subdomain generates for the business. Having an understanding of the type of subdomains is in a position to assist the business in making the build versus buy decisions for the technology solutions. Businesses need to perform multiple different functions to operate. These functions are performed in the purview of subdomains. Different organisations within the same industry may look at their enterprises in a different way. From the subdomains perspective, one of the most common reasons is that businesses may not operate in all sub domains in that overarching domain or industry. Sub domains may be further divided into smaller sub domains and can be seen as sub domain granularity, and it depends on the focus of the business. Each organisation within an industry, depending on their focus may view their enterprise as composed of different subdomains under the purview fo which htey performn their business operations. Each subdomain has a different level of complexity associated with it. There are multiple factors that may be contributing toward the complexity of the sub domain.

4.2.1 Reasons for complexity

The most common reason it the complexity of the business rules itself. Another reasons is the compliance aspect. If the sub domain is operating in a highly regulared environment, then that will add complexity to the sub domain. Complex calculations or complex algorithms may require specialised skills or specialised knowledge to understand the sub domain and that adds to the complexity. Next one is the processes and the handovers required between the sub domain and other sub domains or even external entities; will also contribute towards the complexity of the sub domains.

Dynamic - idea here is that if the sub domain requires changes to processes, rules, structures or any other aspect, then it will be difficult to manage the knowledge and understanding of the business domain as it is changing with the time . Depending on the industry, there may be other factors that will contribute toward the complexity of the sub domain.

4.2.2 Sub-Domain Categories

- Sub-domains are categorized into three types based on their complexity and business value it adds. - Generic, core and supporting.

Generic Subdomain

- Known solutions exist for such subdomains Nothing special about these subdomains Best practices available No business advantage in re inventing the wheel

Core Subdomain

- Each business within a specific industry operated differently within the core sub domains to gain some kind of advantage over their competition. - The ”secret sauce” for the business resides in the core sub domain and the business is always looking to carryout things differently in these core subdomains to get some kind of competitive advantage, depending on the indusitry and the environment. - These core subdomains may evolve at a very rapid pace or may have a very high level of dynamicity. - Organisations are looking to gain some kind of competitive edge by or achieve cost savings which translates into value for the business.

Supporting subdomain

Supporting subdomains do not provide any direct business advantage but the core sub domains depend on the supporting sub domains. Well known practices for supporting subdomains, but solutions may not be readily available. - Even if the solutions are available, those solutions may need to be customised to meet the needs of the core sub domain. *Typically the supporting sub domain does not have high levels of complexity in terms of business logic*

4.2.3 Identifying sub domains

Start by looking at the business capabilities in that sub domain. Are there any known solutions available for the sub domain? - If yes, then that sub domain is likely to be generic. Otherwise need to check if that sub domain adds any business value. - Is there an opportunity for the business to differentiate itself from it competitors by doing things differently within this sub domain. If no then the next check is to see if the core sub domains depend on this sub domain. - If yes then it is likely to be a supporting sub domain. - If no then the the it is a generic sub domain where you may need to build a solution.

If the sub domain has the potential of adding some business value, then next check is to see if the business domain has high complexity. If the business domain does not have a high level of complexity then it is likely to be a supporting sub domain. Otherwise it is likely to be a core sub domain.

4.2.4 Why categorise the sub domains?

Businesses have limited resources and these resources here are referring to the HR as well as the money put aside for the various initiatives across the enterprise. Categorisation of the sub domains helps in prioritisation of the various initiatives. Second reason is return on investments, businesses would like to maximise their return on investments. - As a result, preferences will be given to the initiatives replated to core sub domains which will lead to the maximisation of the return on investments. Third reason is that categorisation of the sub domain helps the business in making the buy vs build decisions. If the sub domain is a generic sub domain, then the business would buy a IT solution rather than build it. Examples SAP, Workday etc. If the sub domain is supporting sub domain, then the business may decide to build the solution by way of outsourcing. Or they may buy a generic solution and customised it to meet the needs of their core domains. E.g salesforce etc.

If they sub domain is categorised as core, then the business will use its best teams and best talent to build the solution in this sub domain. The generic sub domain has readily available solutions that the business can buy. The core subdomain is the sub domain where the business has the opportunity to be different and insulate itself from competitors. Supporting sub domain is needed because the core subdomains depend on the supporting sub domains.

Categorisation of the sub domain help the business in making the build versus the buy decisions and the business gets the most return on investment by investing in the solutions in the core sub domain. Typically the business will use its best talent and resources for building the solutions in the core subdomain.

4.3 Understanding the Business Context

To make an objective decision, you will need additional facts or information on the situation. I.e the context. The idea is that as an IT team, you will be carrying out knowledge crunching exercises. To carry out these knowledge crunching exercises you must be aware of the business context. In order for IT teams to understand the business domain, they must understand the business context.

Lack of understanding of the business context may lead to confusion and misinterpretations and that will lead to misrepresentation of the domain models.

It is important that the IT team understands the business context, without the appropriate understanding of the business context the IT team may not be able to come up with accurate models representing the business domain.

4.4 Business Domain Language

Business teams use business language, whereas the technology teams tend to use technical terms in their communication. This difference in language between the business teams and the IT teams can lead to linguistic challenges. The message here is that the same term used in different regions may lead to confusion, that is even more true for the business language used on multiple domains. If you are using business terms from one domain in another domain then it may lead to confusion and misinterpretations.

Each industry has its own lingo, i.e a set of terms that only the experts in the industry or profession use. They all use a set of terms that have meaning only in their professions and within the same industry there may be specialisations. To understand the domain or to gain knowledge of the domain, one must understand the language used by the domain experts. Some challenges. The first challenge is that there are multiple business languages across the enterprise.

The challenge here is that for building complex, IT teams must learn multiple business languages used by the domain experts in the context of different domains or sub domains. The second challenge is that same term may appear in the context of different business domains. The challenge here is that the same term across multiple domains may have a different meaning depending on the context and cause confusion amongst the IT teams. The third challenge here is related to the fact that IT teams have their own lingo..... These are the terms that are commonly used by the IT teams and technology teams to translate the business terms into IT lingo.

When they receive information about the domain or the knowledge from the domain expert in domain terms, they translate it into technology terms. Communication from the domain expert to the IT expert happens in terms of the domain language, whereas the communication from the IT expert to the domain expert happens in terms of the technology language.

Domain driven design suggests setting up a common language within each business context and this language is used by all stakeholders, including the IT teams. This language is referred to as the ubiquitous language and it helps in breaking down the domain into multiple parts that are suitable for building microservices.

4.5 Strategic Pattern: Ubiquitious Language

Ubiquitous language is one of the strategic patterns in domain driven design. The domain driven design suggests setting up a common language within each business context and this common language is referred to as ubiquitous language. Ubiquitous language can be though of as a dialect that is used by the various teams within an organisation. A dialect is defined by the vocabulary and ubiquitous language has a clea definition of the context to which the vocabulary or the glossary applies to. The glossary contains the common terms and acronyms used within the defined context.

Optionally, it may also have example usage for the terms and acronyms and it may also have references or links to the relevant assets. Can be seen as an enterprise business dictionary with differences. Ubiquitous language is not created and managed separately, multiple in an organisation.

Business expert speaks in business terms. Tech expert speaks in terms of tech terms. There is a misconception that the it is the business expert defining the language. This ubiquitous language is not imposed by the experts. It is not a language used in the industry, it can be though of as a team language, or tribal language that develops or evolves over a period of time by way of collaboration between the business and the technology experts.

The team creating the ubiquitous language can use multiple techniques, such as drawing the process flow, diagrams, user stores and event storming. This lead to the creation of the ubiquitous language for that team. It is important to consider that this language is not a one time job. It is a continuous process as the language takes a while to reach the high level of maturity. There are no special tools for this. Any tool can be used as long as it will make the ubqiquots language accessible to all team memebers. Collaboration tools such as confluence and quip are commonly used. Any tools can be used, so long as it make the language accessible for all team members. Once the team is at a certain maturity level they should use it for verything.

If the team is developing the documentation and they create a new term or they find a new term that then that term shoudl be added to the ubiquitous language.

The ubiquitous language should act as a source of truth for all terms used by the team. This language should be used in the application code as well as in the testing code. Teams must use this language in all of their conversations.

What is the point? Translation between business and tech is no longer needed. Things become more consistent and easier to follow for both the domain expert as well as the IT expert. An interesting side effect is that it helps in identifying overlapping contexts. What that mean is that we can use ubiquitous language to break apart a business context into smaller parts. In domain driven design, these smaller parts or contexts are referred to as the bounded context.

Summary - very useful, should be used in all forms of communication include the code.

4.6 Strategic Pattern: Boudned Context

Business domain models are commonly created by laying out the domain capabilities in the business. Each represents a functional area. Each of these capabilities or functional areas need some kind of a customer representation in technical terms.

The technical expert will put together the class definition with multiple attributes and to do that the technical expert will gather the attributes from multiple experts from across these domains. Although the customer object may have multiple attributes each of these capabilities will need only a few of those attributes to manage the model in the long term.

One of the teams will be assigned the ownership of the common model, and this would lead to a dependency between the teams in other capability or functional areas on the central team that will be managing the model. The dependency will lead to a loss of agility, conflicts and complexity.

Domain driven design addresses the problems with the common domain model by breaking the domain in to independent parts referred to as the bounded context. This addresses the inherent complexity in the unified enterprise model or the common models. Keywords here are bounded and context.

There are certain characteristics that must be met for these bounded capabilities to be referred to as the bounded context. The first one is that each of these bounded contacts must be represented with its own domain model. There is no sharing of the domain model. The domain model is built for the bounded context is only applicable within the boundaries of that bounded context. Each bounded context has its own ubiquitous language.

Characteristics in detail. First one is independent domain models. Each capability has different modelling requirements. These models are created and managed independently by the technology teams assigned to each of these functional units.

Since these models are independent, teams from assigned to each of these functional units do not need to have knowledge of the models managed by other teams. Models can evolve independently across the functional units wihin an enterprise.

Another characteristic of the bounded context is that there is a language that holds meaning within a bounded context i.e ubiquitous language. Terms are only meaningful within a bounded context.

Summary Bounded contexts can be thought of as a conceptual boundary around the business capabilities. These bounded contacts have certain characteristics. Models can be developed in the bounded context, have applicable only within that bounded context. The language is only relevant in that bounded context.

4.7 Discovering the Bounded Contexts in a Domain -This is not a straight forward task. An art not a science.

Need to partner with the domain experts. Start by looking at the organisational structure. Typically they are available as diagrams that depict the various business capabilities or functional areas within the organisation. Next step, identify the business experts in the core domains and partner with them to understand their responsibilities and they key activities they are involved in. Go deeper into some of their key activities. Be on the lookout for clues in the business language used by the domain experts. If you organisation has invested in well-designed, modular, monolithic applications they can also act as a starting point. Look at modules in the monolithic applicaiton and create the boundaries for the business contexts realised in those applications.

Overrall idea is to gather the clues from these exercises and then use these to mark the boudnaries for the bounded context to describe each one of these and the sue of these clues.

Practically speaking. Do not think that the boundaries have to be perfect as this can take some time. Start with some boundary and then none.

Most organisations are good at documenting and maintaining their organisation structure In a new organisation, a good place to start is to give attention to the business cuntions depicted in the organisation structure. Business functions can be treated as a starting point for the bounded context. Just a starting point. Will need to delve deepening into each of these business capabilities to refine these boundaries further.

As a next step, may look at the responsibility of each of the business experts and keep in mind that these business experts may be responsible for multiple functions. During information gathering pay attention to the business language that these experts are using. The key activities of the business are the activities that the business model must carry out to make the business model work.

Going deeper into each of these activities in collaboration with the business experts will help the IT experts to understand the contextual boundaries within the consumer banking downmain.

Need to collaborate with the key business experts to identify the key actives in your organisation and then dig deeper into each of these activities as you are going through. Check for linguistic clues to help understand the bounded context from the business use cases and the process perspective.

Well designed application may already have these boundaries in place. Long story short monolithic applications should not be ignored and they may be used as a starting point for creating the bounded contexts in your business domain.

Summary Leverage available assets like organisation structure and well designed monolithic applications to identify the bounded contexts. Partner with business experts to understand their key responsibilities and activities. Need to pay close attention to the language used to help identify the bounded context.

Chapter 5

Bounded Context Relationships

5.1 Introduction to DDD Integration Patterns

Bounded contexts are independent, but the bounded contexts are not isolated from other bounded contexts around them. Models in the bounded context collaborate to fulfill the requirements of a system. The basic premise behind this is that when you build microservices, they will not fulfill all of the system requirements on their own. These microservices will need to interact with other microservices. These relationships imply some kind of dependency between the bounded contracts or the microservices.

There are multiple types of relationships between the bounded contexts. In a symmetric relationship, two bounded contracts depend on each other. - In this scenario, bounded context A is dependent on bounded context B and bounded context B is dependent on Bounded Context A. In a assymmetric relationship, one bounded context depends on another bounded context B in a one to many relationship. I.e multiple bounded context depend on a single bounded context.

There are multiple relationship patterns, these patterns define the dependency relationship between the bounded contexts.

Note: Big ball of mud is an anti pattern in software development. As a designer of micrservices this should be avoided. Context maps are visual representation of the relationship between the bounded contexts.

5.2 Managinc Bounded Context Relationships using Context Maps

Unmanaged bounded context relationships leads to the big ball of mud. I.e badly structured models that lead to spaghetti code. - This kind of a model and spaghetti code is typically created by unregulated growth and fixes over a period of time. - Big ball of mud in the context of domain driven design refers to an anit-pattern. Generally should be avoided.

Bounded contexts cannot stay isolated but the relationships need to be managed. Otherwise there will be a loss of model integrity, and there will be a loss of a team's ability to operate independently (very bad!)

Challenges: Consider sceanario with BC-A and BC-B. Each has its own model. Say there is a dependency between A and B. Now A depends on B and will have a dependency on B's models. - The impact is that now linguistic boundaries within the bounded context A does not hold. - There may be confusions in terms of the language used for A's and B's models, in other words the BC is no contaminated. This is referred to as loss of model integrity.

Bounded contexts are translated into a one or more microservices. Dependencies between the bounded context are eventually translated to the dependencies between the microservices. (consider) Changes will require some kind of collaboration between the teams owning these

microservices APIs, this means that teams will lose their ability to operate independently. - This will lead to loss of agility which is counter to one of the reasons why microservices architecture is adopted.

We know that bounded context dependencies or microservices dependencies cannot be avoided. The suggestion is to manage these bounded context relationships using appropriate domain driven design patterns.

Teams must make a conscious effort not to create a big ball of mud. They must make a conscious effort not to create a big ball of mud, and they must document the relationship between the bounded context using context maps.

Context maps are a visual representation of the systems, bounded contexts and relationships between them.

Multiple benefits of using context maps. First one is that it makes it easier for the team members to understand the big picture. The next is that it helps in understanding the interdependencies between bounded contexts. The third one iss that is helps the teams gauge the level of collaboration needed with other teams. The context map also helps in refinement of the bounded context and the models.

Summary - As a designer of the microservice, you must avoid creating a big ball of mud. - The idea is that if you create too many dependencies between the bounded context, it will lead to the loss of benefits that you expect to realise form the microservices architecture. - Next one is that as a microservices designer, you must use well-defined patterns for defining the relationships between your bounded contexts. - Last but not least, document the relationship between the bounded the bounded context, using context maps.

5.3 Symmetric Relationship Patterns

Three new strategic patterns. Separate ways, partnership pattern and the shared colonel pattern. There may be a scenario where the bounded contexts in a system has no relationship with other bounded contexts. In such a scenario, the bounded contacts are truly independent or autonomous.

Consider the scenario where there are two bounded contexts A and B which have no relationship. - This means that there is no sharing of models between these two bounded contexts since they are independent. - Teams can work on these two bounded contexts autonomously. - In other words, these teams do not have to collaborate or coordinate for any task. Could argue that there are opportunities to reuse parts of A and B or vice versa but one has to consider the trade off. - The tradeoff is that if there is reuse, then there will be a loss of autonomy. - To go a little deeper say there is a model shared by team B then the teams cannot work autonomously. - Why? because if A has to change the model, then the team for B will need to agree to the changes. - Regarding the reuse, when there is no relationship between the bounded contexts of A and B then this is referred to *Separate Ways pattern* - From the realisation perspective, this means that they will independent set of applications or services for each of the bounded contexts. — Teams will be able to independently develop applications and services for these two bounded contexts. — This means that these two teams can work independently at their own pace to meet the business goals of their respective business unit.

Sometimes you find bounded contexts that have mutual dependency on each other. This kind of a relationship between the bounded context is referred to as the symmetric relationship or bidirectional dependency. To depict a symmetric relationship, a solid line is placed between the bounded contexts. This mutual dependency leads to high level of coupling between the bounded context and this type of relationship is referred to as the partnership pattern in domain driven design.

From the realisation perspective pattern, this translates into services that have mutual dependencies. What this means is that the services may be developed by different teams,but

because of the mutual dependencies between the services, the teams cannot operate independently. Not only that, each of the teams engaged in this kind of a relationship would need to learn the business models and the ubiquitous language for the bounded context managed by the other team. What this means is that the teams in this kind of relationship will need to coordinate their changes, deployments and their releases and that will defeat the purpose of adopting microservices architecture. - How is this problem solved? - One way to solve this problem is to demarcate the boundaries for the shared models. - Consider two independent teams A and B. —– They can create a demarcation around the models that are shared between the two bounded contexts. —– The idea is to manage these shared models independent of the rest of the bounded context. —– What this means is that if A needs a change and this change is not part of the shared models then the team assigned to bounded context A can make independent decisions. —– Similarly if there are changes needed outside of the shared models and bounded context B the team assigned to bounded context B can make those decision independently. —— At any point, if there is a need for making a change to the shared models, then the two teams will coordinate.

The sharing od models between the bounded context is referred to as the shared kernel pattern. The important thing to keep in mind for the shared kernel is that overlapping parts of the context represent the shared domain model, the shared concepts and the shared business language between the two bounded contexts. Typically, the shared kernel is realised by way of shared libraries such as Java Jars, Python packages and Ruby Gems etc..... Teams can independently develop services that use these shared libraries.

It is okay for the teams to use shared kernel and shared libraries so long as the scope of sharing between the bounded context is limited to a small set of models for scenarios involving sharing of too many models. Between the bounded context, it becomes difficult to maintain the integrity of the boundaries of the bounded contexts and this is the reason for the suggestion to use shared kernel only, if we are talking about a small set of shared concepts between the bounded contexts.

Summary - Strategic patterns

First one is the separate ways pattern in which there is no relationship between the bounded contexts As a result, the teams working on the two bounded contexts can work truly independently.

The next one is the partnership pattern in which there are interdependencies between the bounded contexts. As a result, teams must coordinate with each other to make changes to their own bounded contexts.

The third one is the shared kernel in which it is suggested that the boundaries of the shared concepts and models be clearly demarcated and only the changes to these shared models need to be coordinated by the teams. In other words, if the team is working on changes that are not related to the shared models or concepts, the it can carry out those changes without needing input from the other team.

5.4 Asymmetric Relationship Patterns

Two terms, upstream bounded context and downstream bounded context. Three patterns, customer supplier pattern, conformist pattern and the anti-corruption layer pattern.

One bounded context has dependency on another bounded context. This kind of a relationship is depicted by way of assigning roles to the bounded context. The bounded context that exposes modesl to the other bounded context is referred to be in the upstream role an the bounded context that is dependent on the other bounded context is referred to be the downstream bounded context. This depicts a relationship in which bounded context A has knowledge of models in bounded context B, and since the relationship is asymmetric bounded context B does not have any knowledge of the models in bounded context A. - An important thing

to keep in mind here is that this relationship does not indicate the flow of data or information. - It depicts the dependencies from the realisation perspective. — The upstream bounded context exposes some functionality and models that are consumed by the downstream bounded context. — There are two options that the upstream bounded context has. — 1) It can expose the functionality and models based on the needs of the downstream bounded context. — 2) The second options is that the upstream bounded context exposes certain functionality and models without any consideration to the needs of the downstream bounded context.

These are two separate patterns. Option 1 - The upstream bounded context fulfils some specific needs of the downstream bounded context and this pattern is referred to as the customer supplier pattern.- - Can be thought of as the client server pattern where the server creates the interfaces based on the needs of the client. - From the realisation perspective, the supplier team always consults with the customer team to ensure that the suppler service fulfils the needs of the customer services.

Option 2 - The upstream bounded context exposes model with no regard to downstream bounded context requirements or needs. – In this sceanario, the downstream bounded context accepts the models exposes by the upstream bounded context. – This type of relationship pattern is referred to as the conformist pattern. – In this pattern, the downstream bounded context conforms to the upstream bounded context models. – To depict this relationship – BC A is conforming to the models exposed be BC B —– One important not here is that both the bounded contexts use the same model.

Consider a scenario in which the downstream bounded context decides not to conform to the upstream bounded context. - In other words, the team for bounded context A decides to create their own model instead of adopting the models for bounded context B — In that scenario, the models from bounded context will be exposed to the bounded context. — A would require some kind of a translation to convert the models from bounded context B to bounded context A. —- The suggestion is to isolate this translation logic in a separate layer. —- This layer for the translactoin is referred to as the anti-corruption layer and this pattern is also referred to as the anti-corruption layer pattern. — The idea between the anti corruption layer is to protect the bounded context from corruption. — See diagram. This kind of relationship is depicted by replacing D with ACL. ——- Depicting a relationship between A and B, wherein each of these bounded contexts have their own model. ——- They have no knowledge of each other's models except that the ACL has the required knowledge of both the models for A and B and carries out the translation from B's model to A's model.

Summary - The downstream bounded context depends on the upstream bounded context. – In the customer supplier pattern, the upstream bounded context adjusts the model to the downstream bounded context. – In the conformist pattern, the upstream bounded context does not give any regard to the needs of the downstream bounded context. —– SO the downstream bounded context conforms to the upstream models in order to protect the downstream bounded context. – Teams may decide to use the anti-corruption layer. —– This anti-corruption layer has the logic for translating the model from the upstream format to the downstream format that way, —– the downstream bounded context has no knowledge of teh upstream model context and hence there is no direct dependency. ——– All the code related to the dependency is isolated in the ACL

5.5 One to Many relationship patterns

Two patterns, open host service and the published langauge pattern. In the one to many relationship upstream bounded contexts referred to as providers offers common services to two or more bounded contexts. – THis is referred to as the as the open host service pattern. – To depict this common services pattern. – There is an open host service provider which offers a common language for integration. – This common language is accepted by teams working

on the downstream bounded context. — This common language is referred to as the published language, and this pattern is referred to as the published language pattern.

Summary: Open Host Service pattern in which the upstream bounded context provides a set of common services or common capabilities to the downstream bounded context. Published language pattern which goes hand in hand with the open host service pattern.

The upstream bounded context or the open host service provider exposes the common language for the common services. This language is managed by the team responsible for the open host service. The downstream bounded context accepts this publish language

Chapter 6

Domain Driven Design - Tactical Patterns

6.1 Introduction to DDD Tactical Patterns

6.1.1 Tactical Patterns

Tactical patterns - patterns that may be used for modelling and for the realisation of microservices. Model driven design provides a framework for the realisation of systems modelled using the domain driven design approach. The tactical patterns are building blocks in the model driven design.

6.1.2 Entities

Entities, value objects and aggregates are referred to as the domain objects. These domain objects are used for modelling the data in the domain model. Factories are for creation of the complex domains. Objects and repositories are sued for managing the persistence of domain objects.

6.1.3 Services Patterns

Then there are services patterns. Services are used for modelling the interactions of the domain, objects with other domain objects and with infrastructure and with other external components. Two terms - anemic and rich model. Can be modelled in UML and then tactical patterns can be built in plain old java classes.

6.2 Entity Object - Pattern

An entity - represents a uniquely identifiable business object that encapsulates attributes and a well-defined domain behaviour.

6.2.1 Characteristics of an entity

An entity is uniquely identified within a bounded context. These entities can their identities are meaningful only within their respective bounded contexts. An entity has a set of attributed that are defined by the ubiquitous language for the bounded context. An entity has a behaviour.....i.e it encapsulates business logic. This business logic is expose by way of operations. When these operations are executed against the entity it lead to a change of state of the entity. Business

logic is sometimes referred to as domain logic. Entities are meaningful only within the bounded context in which they are defined. It is common to see the same entity names appearing across multiple bounded contexts, but you have to keep in mind that the definition of the entity across those bounded contexts is not guaranteed to staty the same.

Recall that entities are uniquely identified within a bounded context, but sometimes it may so happen that the same attribute is used for unique identifying the entity across business contexts. But that is purely coincidental. I.e. Can't be relied on.

The teams defining these entities will work independently of each other and define the attributes and the operations for the entities based on the requirement in each of the business contexts.

Entity Persistence

Entities are persisted in long term storage (this is important!!!!!!!) The data in the long term storage represents the current state of the entity. It is common for RDBMS and NoSQL databases being used for persistent storage of the entities. In the case of an RDBMS a table represents a collection of entities. The rows within the table represent the entities that are uniquely identified by way of the primary key column. The rest of the columns have the value for the attributes for each of the entities.

6.2.2 Summary

First one is that entities are business objects that are meaningful only within a bounded context. Where they are defined, entities are uniquely identified within a bounded context. Next one is that the definition of the entity consists of attributes and behaviour. The behaviors implements the business logic that may change the state of the entity Entities are persisted in long term storage.

6.3 Value Object - Pattern

Value objects, unlike the entity objects have no conceptual identity in the bounded context. In other words, value objects attributes and behaviour does not map directly to the core concepts in the bounded context. Value objects may also be through of as utility objects.

One important distinction between the entities and value objects is that the value object is not persisted to the database as an independent object. The value object is either persisted to the database as part of an entity object or they are not persisted at all. Value objects are treated as immutable objects. All of the attributes put together gives a meaning to an instance of the value object as a best practice. It is suggested that you create a new instance of the value object instead of reusing the existing instances.

Value object and bounded context. Value object in one Bounded Context may be an entity in another bounded context.

6.3.1 Summary

Value objects do not have a conceptual identity within a bounded context. Value objects are not persisted in a database as an independent objects. It is either store as part of the entity object or it is not even stored in a database. An entity in one bounded context may be a value in another bounded context and vice versa.

6.4 Aggregate and factory pattern

An aggregate object is a cluster of entities and value objects that are viewed as a unifed whole from the domain concepts and data perspective. An aggregate consists of an aggregate root, also referred to as the root entity. The root entity has a unique identity from the domain perspective.

The second part of the aggregate is the cluster which is formed by the boundary for the aggregate. Within this boundary, there may be zero or more aggregates and value objects.

Objects in this cluster or the objects within the boundary are referred to as the inner objects or child objects.

Invoking methods on the object directly..... procedural programming ????? Advised against. It is good practice not to directly interact with the inner object.

The code outside the aggregate will invoke the function and this function will encapsulate the business logic for operating on the inner objects.

As a designer of the aggregate, it will be your responsibility to ensure that all of the require behaviour for operating on the inner objects is exposed as function by the aggregate root object. In general, the aggregate object are stored in multiple tables. In the case of RDBMS and the case of NoSQL data base the aggregate object may be stored across multiple collection when an aggregate is inserted or updated against these databases. This needs to be done in an atomic fashion. ACID i.e either all the changes are successful or all the changes are rolled back Unit work is an important consideration when you are designing aggregates.

Factory Design Pattern

Factory design pattern is a common pattern for building complex domain aggregates. The way it works is that you define an object that has all the logic for creating the domain aggregate. This factory exposes a function that can be invoked by the code. When this function for creation of the aggregate is invoked on the factor, the factor reads the data for the aggregate from persistent storage. Creates the aggregate and returns it to the caller This is a very comon design pattern and is not necessarily exclusive to microservices.

An aggregate with zero inner objects will be an entity. Keep in mind a aggregate with zero inner objects may also be referred to as an aggregate or an aggregate root.

> **Note:** Note: Entity aggregate and aggregate root may be used interchangeably.

6.4.1 Summary

- Aggregate may contain other aggregate entities and value objects.
- Aggregates must encapsulate behaviour to manage inner objects.
- All changes to aggregates are saved atomically
- Factory pattern is commonly used for creating complex domain aggregates.

6.5 Model Behaviour: Anemic and Rich Models

Two terms: Anemic and rich models. These terms are used for describing the model behaviour.

Models which do not expose any business logic are referred to as anemic models.

6.5.1 Formal Definition

A more formal definition: A domain model composed of entities that do not exhibit behaviour i.e operations applicable to the domain concepts are missing.

Opposite is the rich domain model and this term is coined by martin fowler in a blog. Source of research. Used an analogy to describe a bad model. If there is no behaviour then it is not a good model.

6.5.2 How to describe an anemic model

What to look at Do the entities lack behaviour? If the entities lack behaviour, then chances are it is an anemic model. If the entities do have some functions, the you should check that these are not just for CRUD operations. That is that they are not just getters and setters with update and delete operations. If they are just CRUD operations then the model is anemic.

The third thing that you need to look for is the implementation of business logic outside of the entity objects. The idea is that the entity object does ont have any kind of business logic implementation, but it does have the CRUD functions. The business logic code resides in an external component that invokes the getters to get the data from the entity object. Uses the data in the business logic code and then calls the setters on the entity object ot set the data back on the entity. This external component is commonly implemented as shared services or are coded as part of the application directly. In the case of services, the domain objets provide the CRUD operations to the services which reside in a separate layer. All of the business logic resides in the components or services implemented in the shared services layers. This kind of an architecture is commonly seen in organisation that have adopted service orientated architecture.

The second way in which business logic may be externalised is by way of creating applications with the business logic embedded within the application code. In this scenario, the business logic gets repeated across multiple applications. Since there are many applications which are dependent on the domain layer, managing changes in the domain layer objects will be a challenge. This kind of a situation should be avoided at all costs.

> **Note:** Common in legacy applications.

6.5.3 Rich model

A rich model implements the model behaviour inherently as a part of the entity object. Object has all the business logic in one place. It implements all of the domain concepts.

A positive effect of having all the business logic in one place is that hte models data integrity is maintained and this is because the owner of the object will have full control on how the data is managed. Versus every application implementing business logic on their own.

Guidelines between different models. Anemic is not bad for simple domains. Need to think about the scenario, your use-case and then decide. Is the business logic changing frequently. If so an anemic model may not be an issue. Generic data services like CRUD APIs then anemic is okay. Shared scenarios or use cases where there is shared logic that does not belong in a single model entity. This scenario would be okay for an anemic model. Anemic models can be considered an anti pattern in some scenarios and in others it may be okay.

Common misconceptions is that in a rich model every entity should implement the behaviour and that is not right because in an aggregate object the aggregate root implements the behaviour.

6.5.4 Summary

Anemic models - refers to models in which the entity is implemented with no behaviour. To identify these you need to look for symptoms such as entities not implementing the behaviour. Entities implementing basic CRUD behaviour And business logic being in external components rather than being in the entities. Not always an anti-pattern - use case dependent. Grayzone.

6.6 Repository Object - Pattern

6.6.1 Formal definition

Formal definition: A repository object acts as a collection of aggregate objects in memory. It hides the storage level details needed for managing and querying the sate of the aggregate in the underlying data tier An aggregate root is an entity that is managed in some kind of long term storage. RDMS or NoSQL or even basic file system. All of the logic to interact with the data storage is encapsulated by the repository object. The details of these interactions are hidden from the client or the callers of the repository object. The caller of the repository object function use the CRUD operations to manage and query the state of the aggregate. The called of these curd operations have no knowledge of the underlying data tiers. For these callers a repository is acting as a container for a collection of aggregate objects.

> **Note:** Important Note: The client or call of the repository object function has no knowledge of the data tier.

6.6.2 Repository pattern characteristics

Characteristics of the repository pattern or repository object. The first one is that for every aggregate defined in the domain model, you have on and only one repository. In other words, not a good idea to implement multiple repositories for a single aggregate. A repository may expose high level behaviour of function and this is apart from the typical CRUD function that are expected to be implemented by the repository. Repository objects are managed as part of the domain layer. In addition to the typical crud functions a repository may also expose higher level functions, mostly for queries.

Persistence. Consider a function on an object which requires two updates on two separate databases. The idea is that both databases will be updated in a single unit of work. The atomicity aspect of the aggregate is enforced by the repository. The idea is that all changes to the sates of the aggregate that is the aggregate root as well as the inner objects is carried out under a unit of work. Either all the changes to the database are successful or no change is made to any of the entities that are a part of the aggregate. Ths may happen if there is any error in the database operation for any entities in the aggregate.

The primary benefit of using a repository is that it keeps the domain model independent of the storage layer. The domain model is independent of the storage model. This means we can use a RDMS, NoSQL or a file for the storage of aggregates but the domain model doesn't need to know about it.

> **Note:** The repository keeps the domain model independent of the infrastructure Can use JDBC for SQL against a RDBMS or an SDK for NoSQL. The core idea is that any change to the database tier will be isolate to the repository object and will have no impact on your domain model.

6.6.3 Testing

Repositories can also help with unit testing and mocking. Idea is simple for carrying out the testing of the client code. The database may be replaced with mock up implementation of the database. This could be simple pojo that just returning static database responses. This allows for moving faster without any dependency on the availability of an actual database.

6.6.4 DB Performance

One common concern raised for repositories is that for large aggregate objects there amy be an impact on the performance. The reasons for this is that large aggregate objects may need to execute multiple database operations or a join across multiple tables may be required which can lead to suboptimal performance of the query. This concern can be addressed using caching solutions such as Redis, MemCache or Reactive Mongo.

Another common concern is related to the use of criteria based queries in repositories. Typically the repository exposes a retrieve operation that retrieves an entire aggregate object and some applications may not need it. (graphQL???) In other words only a partial result set is required which is not supported out of the box. There may be a scenario where data from multiple aggregates may be needed.

> **Note:** This is specifically of importance to a mobile application where slow speed may impact the users interest in the application.

Solution

Two ways to address this concern. First way is to expose high level function in the repository The second is to expose additional queries outside of the repository.

The realisations of the repository requires the developer to code the mapping between the domain objects and the domain objects and the database and vice versa. This mapping may be quite complex and the coding may be quite cumbersome for large objects. This can be addressed using mapping frameworks like hibernate These mapping frameworks simplify the task of mapping between the domain objects and the databases

6.6.5 Summary

Repository objects make the domain model independent of the database layer. Database operation on the aggregates must be atomic The repository object enforces the atomicity. Repository objects may also be used for unit testing and mocking.

There are some common concerns related to repository objects but these common concerns are related to the query functioning may be addressed By exposing high level query function in the repository object By using Reddis and Memcache and by creating Exposing query functions outside of the repository object.

6.7 Domain Service - Pattern

The idea is the implement the behaviour that does not belong an entity in a standalone independent services Caller and client code can invoke an operation on the independent service.

6.7.1 Formal Definition

It is a domain object that implements the domain functionality or concept that may not be modeled naturally as a behaviour in any domain entity or value object. Domain service is part of the domain model.

There are different types of services. Important to understand their characteristics.

Domain services.

- Always implement business behaviour for the domain.
- Domain services are stateless
- Domain services are highly cohesive.
- Domain services may interact with other domain services.

Details Since the domain service has the business behaviour,teh domain service object is aware of the other domain objects. In short, any kind of domain concept may be implemented in the domain service. The caller of the domain service does not have the awareness of the details of the behaviour implemented in the domain service. So a domain service insulates the caller or the client code from the business logic details. Domain service does not maintain state between calls.

What this means is that if you are calling hte domain service, there is not state variable or direct persistence mechanism built into the domain service. - Domain service depends on the entity objects to take care of the persistence. - The implication of no state variable or persistence is that there may be calls coming from multiple callers or clients, but there is no correlation between these calls and it does not matter whether the call is coming form the same client or different clients. - Bottom line is that there is no correlation between any calls originating from anywhere.

Domain services are highly cohesive. They do one thing and do it well. Domain services may interact with other domain services.

6.7.2 Section Summary

Summary Domain services are technology agnostic. There is a common misconception that all domain services should be exposed as APIs and that is not correct. A domain service is independent of the technology used for the invocation. Could be a POJO or it may be carried out over a network protocol such as HTTP or a MQ

Characteristics of a domain service. - The domain service implements domain behaviour that does not fit naturally in other entities and value objects in the domain model. - Other characteristics are that the domain service is stateless - The domain service is highly cohesive. - Domain service may invoke other domain services.

6.8 Application Services - Pattern

6.8.1 Formal definition

An application service is a domain object does not implement any domain functionality, but depends on other domain objects for exposing high level domain objects for exposing high level domain functionality to the consumers external to the model. The key difference between the domain service and the application service is that application service does not implement any kind of business logic or domain functionality. The other big different is that the application service is exposed to external consumers such as web application, mobile application or application servers.

6.8.2 Characteristics of an application services.

Application services have no domain logic and this is the amin difference between the application service and the domain service. Application services just like domain services are stateless. Application services may define an external interface. Application services are exposed over some kind of network protocol.

Details An application service has no domain logic. It depends on the other domain objects for domain logic. This is the main difference between a domain service and an application service. The application service orchestrates the execution of the domain logic.

Like the domain service and application service is also stateless.. There is no state management carried out in the application service. There is no state variable or persistence of domain object implemented in the application service. Application service depends on the domain objects for persistence An applications service exposes the interface used by the outside world. By outside world what is mean is external components which are outside of the domain of the model. Point to note here is that the interface exposed to the outside world does not need to be an entity or value object. In other words the schema of the request and response for the application service does not need to be aligned with any of the domain objects.

An application service exposed th external interface over network protocol in a domain model. The application service may be thought of as a boundary object that protects all of the objects within the domain model. The application service may be exposed as an API and this API is consumed by the external components over network protocol. This network protocol may be HTTP, MQ or may even be proprietary protocol.

The format of the data between the external component and the API is flexible. Can be JSON, XML, CSV or any other formate depending on the implementation of the application service. The external components may or may not have knowledge of the domain objects or their structures.

An application service may expose a domain service to external components.

6.8.3 Summary

Application services do not implement any domain behaviour They provide high level services by way of orchestrating execution of the domain logic in the domain objects Application services expose interfaces to external components That is components that are outside of the domain model by way of a network protocol such HTTP or MQ

6.9 Infrastructure Services - Pattern

An infrastructure service is defined as a service that interacts with an external resource to address a concern that is not part of the primary problem domain. It defines a contract used by the domain objects to interact with the outside services. Key word here is the external service.

Commonly used external resources. First one is logging system. Domain objects need to log messages and this can be done against any kind of external logging system such as Fluid and Elasticsearch Domain objects may need to send out notification as a part of the business process. Domain objects very likely need some kind of persistence mechanism and this persistence mechanism may be an external database or even a file system. It is common for the domain objects to be dependent on external services or APIs such as Salesforce,Google Maps etc. Just some common examples. LIke the application services infrastructure services have no domain logic. Infrastructure services follow the single responsibility principle and expose a standard interface or contract.

6.9.1 Details

Infrastructure service has no domain logic because it provides infrastructure service, not a business service. It does not have any direct dependency on the domain objects and the infrastructure service sis consumed by the domain objets and services to interact with the external resources. Single responsibility. The idea is that the service provides functionality for one and only one thing. The intent is to simplify the implementation and make it easy to understand the service.

An infrastructure service defines a standard contract between the model and the external resources. It can be thought of as an API which mean for consumption by the model objects and services. The realisation of the API is in the infrastructure layer and this make the model technology and external services agnostic.

The domain object when they need to interact with external resource will go through the API will go through the standard contract exposed by the infrastructure services. The infrastructure service will interact with the external resource by way of the SDK or APIs exposed by the external service. The infrastructure service will carry out the translation of teh call from the standard interface to the interface exposed by the external resource. It will also carry out any kind of transformation needed on thee data. Thus making the domain model independent of the external resource.

6.9.2 Summary

Infrastructure services like application services do not implement any domain behaviour Infrastructure services expose external resource by way of standard interface or standard contracts This standard contract mechanism insulates the domain model from changes in external services.

Chapter 7

Event Driven Architecture and Domain Events

7.1 Intro

Relationships between bounded contexts. Bounded contexts get translated into microservices and these relationships get translated into interactions between the microservices. REST over HTTP APIs are a common synchronous mechanism by which microservices interact.

Microservices can also produce different types of *events*. These are consumed by other microservices as well as the components within the bounded context where the event was produced.

Event driven architecture is an architecture paradigm promoting the production detection, consumption of and reaction to events. Microservices are producers and consumers of events. As a result, event driven architecture is commonly used for building applications with microservices. Events are asynchronous in nature and realisation of events based interactions require the use of some messaging technology. There are multiple messaging technologies available for this purpose. (Kafka / RabbitMQ)

7.2 Monolithic and Distributed Communication Patterns

First pattern is the monolithic object communication pattern in which the object calls methods on other objects. All of this is happening within a common memory space or within a common process. This is typical of a monolithic application. Referred to as a monolithic object communication pattern.

> **Note:** In distributed applications or systems the components reside in their own process space.

In other words there is no sharing of compute or memory resources between the components. These components communicate with each other by way of some kind network protocol. This network communication protocol may be synchronous or asynchronous in nature.

- Synchronous - the caller statys blocked until it received a response from the other component. Examples: HTTP, RPC
- Asynchronous communication - the caller does not wait for a response. Examples: Advanced Message Queueing Protocol

These are not mutually exclusive. Communication may be between two endpoints. Referred to as 1 to 1 communication, also known as single receiver communication. Common

example is HTTP, wherein the communication is between two endpoints. With synchronous protocols you wil always have 1 to 1 communication.

This communication pattern may also be realised by using the asynchronous messaging mechanism. It will look synchronous but the underlying protocol is asynchronous

In a one to many communication pattern, there are multiple components that are interested in receiving messages from the sender. This is commonly achieved by way of a pub sub messaging pattern. A

> message is published by a publisher to a topic ther components will subscribe to the topic and receive messages. ach of these components will carry out a specific task assigned to it.

Common protocols for this kind of communications is REST over HTTP for these microservices.

7.2.1 Messaging Products - Two types

For messaging there are two types of products. Products that are AMPQ compliant such as ActiveMQ and RabbitMQ. Then there are products that are non AMPQ compliant such as Kafka, Amazon, SQS, Amazon, SNS. These are only examples.

7.3 Event driven architecture

Events occur naturally in business scenarios. Events are an indication that something of significance has happened at a point in time or that something has happened in the past.

There are consumers who are interested in knowing about the events, so they would like to be notified. When an event occurs, one or more consumers are notified about the event. On receiving the event, the consumer may carry out processing, and this processing of the event is independent of the producer of the event and other consumers of the event. An event driven architecture is a software architecture paradigm that promotes the designing of systems as loosely coupled components that act as event producers and event consumers. Central to the event driven architecture is an **event backbone**.

> **Note:** This event backbone is an infrastructure component.

It is referred to by multiple names, event bus, event broker events, router events and mediator events hub. These are common terms used for this component. The name depends on the product features and functions and the vendor """"marketing"""" these products. An important to keep in mind that conceptually they are similar. They all provide a way for routing of the events from producers to the consumer. The producer of the event informs the event backbone of something of interest happening. They do this by way of synchronous mechanism exposed by the events backbone. When the events backbone receives the event, it figures out where that event needs to be routed depending on the consumers that are interested in that event.

The important thing to keep in mind here form the event consumer perspective is that they are not polling for these events. They are notified. Same as the pub sub messaging pattern. Commonly used for realising event driven architectures.

7.3.1 Data in the event messages.

The content of the event message may be the state data or the metadata. The consumers receiving the message representing the event will get all the relevant state data and they can carry out the processing using the state data that they will receive in the event message. It

is also possible to design the event structure such that it contains only the metadata. Some may need metadata only others may somehow return to the publisher and retrieve the detailed information. You would have to use both the state data and metadata in your event messages. There is no hard and fast rule but certain things which should be taken into consideration. If the message size it too big, then there may be challenges related to latency. In this scenario you may consider using only metadata in your messages. If the consumers are leading to a lot of chattiness in your application because they are reaching out to the producer for getting the state data, then it will be a good idea to go with the state data in the event message.

API Versus Event Driven Architecture

APIs are directed commands, whereas events are observables. The central theme in an architecture that depends on APIs is an orchestrator. An orchestrator may be thought of as a centralised component that holds the business logic and the flow decisions. When the orchestrator needs to do something it invokes an API, waits for the response, then invoke other APIs to carry out the required business processing.

In the case of EDA the orchestrator is replaced with an event producer and event consumers. These event consumers are the observers of the events which are produced by the event producer.

One big difference between these two architectures is that the event producers and event consumers all have business logic and are responsible for achieving the desired results from the business process. The event producer does not make a direct invocation on any component.

Rather, it simply triggers an event, a message that is passed on to all the consumers. Keep in mind that even producers may also be event consumers and event consumers may also be event producers.

7.3.2 Differences between API and Event Driven Architectures

In the case of APIs, the caller has a knowledge of the API endpoint, whereas in the case of events the producer doesn't know any consumer. APIs are synchronous, meaning that the caller has to wait for the response to come back. The caller is blocked. In the case of events, a producer just produces an event and moves on. Note it does not even depend on the availability of the consumers. What this means is that it can lead to higher availability in the case the consumer is not available The producer moves on and when the consumers comes up it receives the event data. APIs are based on the reqeust response paradigm wheres in the case of events the message consists of event data, which can be the state data or the metadata in the case of the API. Even with the distributed architecture, there is a relatively high level of coupling between the API caller and the API endpoint, whereas in the case of events , the event producers and the event consumers are highly decoupled and the architecture itself is extensible. The reason is that you may add or remove consumers without impacting the consumer in any way. Typically with APIs, the business logic is centralised, whereas in the case of events the business logic may be spread across multiple components. Each of these components, the producers and the consumers have the autonomy to make business decisions. With APIs, it is easy to understand the flow because the business logic is centralised. With event driven architecture, it is relatively difficult to follow the business logic.

EDA is preferred for microservices but both APIs and events can be used. The decision depends on your use case and your requirements.

7.3.3 Section Summary

Events indicate that something of significance has happened. Event driven architecture is a software architecture paradigm which is based on events. By nature, EDA uses asynchronous communication between consumers and producers. EDA is highly decoupled and extensible and the business logic and the business processes in the case of EDA are decentralized. That means all of the components in the system have some business logic which may be managed independently of other components in the system.

7.4 Domain Events - Pattern

Events are an integral part of the model defined for the bounded context. It is important to capture all the relevant events within a bounded context, as a part of the model and the ubiquitous language for that bounded context. These events are raised by the model components, when there is some kind of a state change. Each of these components can raise events to indicate some kind of state change. There may be other sources of events within a microservice. There may be application monitoring but this is not emitting events related to the bounded context. These are technical events and are not a part of the bounded context. Components that are emitting events are referred to as the event sources. Event consumers are the consumers of those events and these event consumers may be a part of the same microservice. In that case the events emitted by the event source are referred to as the domain events. Events commited by the event source may be consumer by other microservices. What that means is that an event defined in one bounded context is getting consumed in a different bounded context. In such cases, the event is referred to as an integration event. Event consumer may also be part of an external services. Basically can be thought of as a legacy service, APIs or any other service not implemented as a microservice in such cases. Semantically there is no difference between domain events and integration events. Its just the consumer that decided whether the event will be referred to as the domain event or the integration event.

7.4.1 Formal definition of domain event.

Formal definition of domain event. A domain event is a message that informs other parts within the same bounded context that something of significance has happened. This is the key part within the same bounded context. Now, when a domain event is triggered, it indicated a state change within the bounded context and the consumers of the event receive the event message and execute some busines logic within the same bounded context.

> **Note:** Note: The domain event is triggered within a bounded context and it is consumed within that same bounded context.

The reason why events have become an integral part of the domain driven design is that events occur naturally. Important Note: That is all domains have concept of events. A quick way to identify events in a domain is to look for statements like when this has happened then do this. This part represents the event and this part represents the reaction to the event that is the business logic. This reaction is known as a side effect.....!!!

Language plays an important role in domain driven design. Important to be very careful when you are naming you events. Always use past tense as event has already happened.

Best Practices

Best Practices: - Ubiquitous language so that there is not need of translation between IT teams and Business teams - Do not add ”event” as a suffix to the event name or do not add ”operation”

as a suffix.

Event handler refers to the implementation of the event consumer logic. In the case of the domain event, the handler is part of the same microservice codebase as the event producer. These handlers subscribe to the events of interest and there may be zero or more handlers per event. Like the events the handlers must be named appropriately.

Use of ubiquitous language as suggested as a best practice, you can name the handler the same as the event that the handler is handling.

Common misconception about event realisation is that events have to be managed with messaging but that is not true for domain events. Domain events may be emitted and consumed synchronously, for example by way of direct function calls. Note: Domain event producer and the handler are in the same process. You can use asynchronous mechanism such as in-memory messaging or even an external message broker. State changes and raising of the events need to happen atomically. In the case of synchronous calls, its easier to implement such mechanisms, whereas with asynchronous you will need to use the appropriate patterns.

7.4.2 Section Summary

Two types of events - domain and integration events. Domain events are emitted and handled within the same bounded context, whereas integration events are handled outside of the bounded context. Domain events may be handled synchronously as well as a asynchronously appropriate. Naming conventions should be used for naming events and the event handlers. State updates and raising of the event must be done in a unit of work or in a transaction.

7.5 Integration Events - Pattern

Event consumers are outside the source Bounded Context Can be consumed by external services An integration event is a message that informs components outside of the source Bounded Context that something of significance has happened A integration even does not lead to any state changes in the source Bounded Context

Domain versus Integration Events

- Domain - Within a Bounded Context i.e Microservices - State changed within a BC - Direct function calls - Synchronous calls - Modelled as part of BC model - Integration - Between BC or BC and External Services - No state change in source BC - Must be a Network Protocol - Asynchronous preferred - Consumer decides

Integration events communication - Asynchronous is preferred - To achieve higher levels of decoupling - Future extensibility i.e add new consumers - Enables one-to-many

Messaging Technology examples: Kafka, RabbitMQ, ActiveMQ.....add others

Relationship - Domain and Integration Events - Domain event may be published as integration event - Both events defined as part of the model for the BC - Semantically the same - Publishing mechanism is different - or same if external messaging is ued for both event types

Integration Event consumer - Consumer may leverage anti corruption layer

7.5.1 Section Summary

- Domain events may be published as integration events
- Integration events published asynchronously
- Consumer bounded context may use ACL

Chapter 8

Event Storming for creating shared knowledge

Events occur naturally in domains. What that means is that to understand the domain, you must understand the events consumed in the domain. Event storming is a collaborative exercise that is carried out by the stakeholders to identify the events, producers and consumers in a given scope. The objective is to create a shared understanding of the domain. The outcome of the exercise is a knowledge model for the domain. Event storming is carried out in the workshop format. There is a facilitator who works with the stakeholders from different parts of the organisation or different parts of the domain.

This workshop may be carried out in person or online using collaboration tools in one of the lectures.

8.1 Introduction to Event Storming

Knowledge crunching is a way by which teams process the knowledge received from the domain experts into a domain model.

How do receive the knowledge from the domain experts? - Interviews - Design thinking (from domain driven design)

Common technique called event storming. (Just examples)

Other ways in which you can receive knowledge from the domain experts in a structured manner. My focus here is on event storming. A formal definition of event storming is a collaborative, workshop based technique for creating a shared understanding of complex business domains and processes. Key to realises that this is creating a shared understanding.

The intent is not to design and model the system but to just create a shared understanding among the stakeholders within the domain, and the stakeholders are the business experts as well as the technology experts. Event storming may be used for creating this shared understanding from the overall perspective or the big picture perspective as well as it can be used for creating the shared knowledge from a business process perspective. The technique for this is quite flexible in terms of what it can be used for. The central theme is business events.

The participants in the workshop identify and understand the business events. They look for the cause of the business events as well as the effect of those business events. - Technique was created by Alberto Brandolini in 2012 **Biggest benefit of this technique is that it accelerates the development process for complex applications** One important think to keep in mind is that the knowledge created by way of event storming is used as an input for creating the models. In other words, you will not be replacing modelling with event storming as an input for creating your UML diagrams. See book about event storming.

The workshop

The most important thing about the event storming workshop is that you must invite the right set of domain experts. There is a dedicated facilitator who works with the participants. This dedicated facilitator should have some prior experience with event storming. The number of participants in the workshop is dedicated by the scope. If you are carrying out the event storming workshop for understanding the business process then you should expect have between 4 and 8 participants in the workshop. The duration of the workshop depends on the scope of the domain and the experience of the participants. It may vary anywhere from a couple of hours to a couple of days. In-person workshop is preferred over online workshops and the reasons is that event storming involves a lot of interactions when the participants are in the same room.

Recently online workshops have also become commonplace and are becoming more acceptable. Good tools to carry these out online.

An in-person workshop is conducted in a spacious room with a lot of walking area. This room must offer enough free space on walls to hang the plot of paper so that the participants have unlimited modelling space. I.e they will not have to stop throwing their ideas on the plot of paper due to running out of the space for it. Participants will be using a lot of different colored stickies. At the end of the workshop, the walls will have plotter paper hanging of the walls and there will be a lot of stickies on it. The stickies on the plotter paper, are color coded.

For an online workshop, participants join over a video call and use a collaboration platform for carrying out the event Storming activities.

All the participants can make changes to a common virtual board and these changes are visible to other participants in real time. There are many collaboration platforms that allow the participants to do exactly that.

The expected output form the workshop is the creation of a shared understanding of the business process. The shared understanding is then used for the modelling of the domain.

The objective is not to design to the system. The objective is not to be able to answer all the questions. It is not to produce the domain driven design models.

Event storming is a collaborative process for creating share understanding of the domain or the business process.

Event storming is carried out in a facilitated workshop format and it may conducted in person or online.

8.2 Elements of Event Storming

Read the book by alberto....

Business events are natural in all domains. They are the starting point of the conversation in an event storming workshop. An important point to note is that all business events are referred to is that all business events are referred to as the domain events in the context of event storming. The objective of the event Storming workshop is to understand the causation. What this means is that the participants discuss the domain events to understand the cause of those events and then they also discuss the effect of those events.

This cause and effect is depicted as teh knowledge of the domain by using the six basic elements.

There are six building blocks or elements which are used for depicting the knowledge or flow in an event storming workshop. Color coded stickies are used to represent each of the six elements. There is a suggested standard for the colors of these stickies or the elements, specific colors are not required.

You can decide on a standard for the colors of these stickies or the elements but you don't have to follow it as long as you are consistent through out the workshop. - A domain actor causes a state change in the domain and this state change is initiated by way of a command

invoked by the actor. - This state change is initiated by way of a command invoked by the actor. - Command is represented by a blue sticky on the workspace. - This command leads to the raising of domain events, a domain event is a representation of some fact that has already happened. - A domain event is represented in the workspace as an orange sticky. - When the domain event is raised, it may lead to a reaction, and this reaction is carried out by way of component that is referred to as the policy. - This policy is represented by way of a purple sticky.. – Since domain events represent something that has happened in the past, they should always be named in past tense.

Important to understand that a command is abstract. That is , it does not represent an active component within the domain. It simply represents the intent of an actor that must be carried out by the domain.

Business logic execution is carrier out in the command processes in the context of event storming. The element that carries out the processing of the command is referred to as the aggregate.

The element that carries out the processing of the command is referred to as the aggregate. An aggregate is represented by way of a yellow sticky. Apart from the command, an external system or service can also be a source of an event.

Think of the external service as something outside the domain under consideration. Such external services are represented by a pink sticky.

An event is directly or indirectly associated within a domain. A state change represents some kind of change in data within that domain. This data may be of interest to the stakeholder, so the way it works within event storming is that event has some data which is represented by a read model. This read model is the response to queries for the domain data. The read model is used by the user interface that the stakeholders can use. - This user can be an email - Doesn't have to be a pane of glass showing the data. - It can be a dashboard for the executor or it can definitely be a browser based application during the event Storming workshop. The read model is presented by way of a green sticky. It is also referred to as the. It is also referred to as the event data model or query model. They key point to keep in mind is that event has some data which is of value to the stakeholders and this read model repsents that data.

Summary

Central idea is the domain event. The domain event is caused by a command which is processed by the aggregate domain. Event may also be caused by a command which is processed by the aggregate domain. Event may also be caused by a command being processed by an external service. The effect of the domain event is realised by way of a policy that is triggered by the domain event. This policy may further invoke commands that may lead to other domain events. Domain event has some data of value to the stakeholder. The data of value is represented by way of a read model. This read model may drive a user interface.

8.3 Preparing for the ES workshop

Common for large organisations to hire outside consultants to carry out the facilitation. (Author says you don't need them.....)

Existing team members such as project managers and Scrum masters can easily be trained to become event storming workshop facilitators. Bottom line is anyone can learn to become a facilitator by observing experienced facilitators and by practicing.

Make the assumption here that the facilitator is identified and the participants have been invited. What does the facilitator have to do next? Prepare the room for the in-person workshop, or if it is a remote workshop then they need to ensure that all the tools are ready to go. Second

thing they need to do is on the day of the workshop, educate the participant on what is the event storming workshop.

Define the scope of the workshop. Make sure all the participants are on the same page in terms of which business process or processes are in the scope of the worksop. Then level set the expectation by discussing the expected outcome. Once these tasks have been completed by the facilitator, its time for the facilitator to dive in.

Details of each task. To set the stage for an in-person workshop, you need to make sure that the room that you're using is spacious. Move all tables and chairs out of the way. Reasons for this is to provide space for the participants to move around and hang the plot of paper on the walls with masking take and then draw a timeline from left to right. The number of plotter papers that you will hang on the wall will depend on the scope of the exercise. Don't have to hang the plot of paper all around the room, you can do it as you go along. Idea is to be prepared to set up the stage for a remote workshop, ensure that all the tools are ready to go at least two days prior to the workshop. Ensure that all the tools are ready to go at least two days prior to the workshop. I.e have the video conferencing setup. Must have the tools for the collaboration setup, create a board and make sure all teh participants are able to connect to the video conference and they are able to use the collaboration tools as well. –¿ Avoid addressing technical challenges faced by the participants. This can be a dampener. —¿ make sure everything is working from day one before the workshop starts.

May have a group of participants who are already experienced with the event storming, but there is no harm in spending a few minutes reminding everyone as to what is involved in the workshop and as part of this education do not use any technical terms. Specifically do not talk about domain driven design. Keep it business focused. Note, participants are no necessarily technologists. Discuss, the purpose of colored stickies and the workspace, but you don't have to spend too much time on these aspects as participants will learn as you processed through the workshop. Something that really helps is put all the colored stickies and what they represent somewhere in the workspace so that it is visible to anyone who has a question about what color to use, which is an extremely common question.

In the beginning of the wrokshop, after the general overview of event storming, it is time for the facilitator to define the workshop, after the general overview of event storming, it is time for the facilitator to define the scope of the workshop. The scope of the workshop may be a big picture from the domain perspective, or it may be a single business process. Irrespective of what the scope is. It is important for all participants in the room to be on the sme page and ensure that everyone stays on track. Place the high level objective for the workshop in the workspace so that it is visible at all times.

Another thing that is important is that as you will start going through the workshop, it is very much possible that there may be discussions around aspects which are out of scope from the workshop perspective, but they may be valuable from the domain perspective. As a result, you don't want to lose out those aspects, so create a dedicated space to list out all of these out of scope elements. You can follow up on these items after the workshop.

Facilitator, before the workshop is set the expectations and this is best done by way of sharing real experienced from the past workshops and if possible, sharing pictures form those workshops. These expectations should be realistic.

Engage the participants and ask them what their thoughts are on the event storming workshop a swell as on what they expect. Idea is to get everyone excited so that they become active participants in the workshop and they should all be looking forward to learning and teaching. Use this part of the workshop for ice breaking activities. At this time, the facilitator is ready to dive in. One of the important roles for the facilitator is to make sure that everyone is having fun and is energised because low energy will lead to an outcome which is going to be of bad quality. Participants must feel engaged. Participants must be active throughout the workshop, so a facilitator must keep those aspects in mind.

8.4 Conducting the ES Workshop

Facilitators must keep in mind that event storming doesn't require the participants to use all of the elements to create the knowledge model. It is the facilitators job to help the participants pick up the relevant elements to design the knowledge model. A lot of flexibility in how the event storming workshop is conducted.

Facilitator may adjust the steps, the flow and the pace as needed. These adjustments depend on multiple facts such as the facilitators and participants, past experiences, complexity of the domain and granularity of the knowledge model. Whether it is going to be high level or whether it's going to be detailed. There may be other factors as well.

General steps. The first step involved in the workshop prioritises the participants to identify the domain events. Once the domain events have been identified

as a next step, participants discuss the ordering of the events across the timeline. In this step, duplicate events are also revealed and removed.

In the third step participants are asked to identify the cause and effects of events. Here commands, policies and external services get added to the knowledge model.

In the fourth step, the commands are associated with the aggregates.

———

First step in the knowledge gathering exercise. The facilitator asks the participants to brainstorm and hang as many events as possible in the workspace. Initially participants may be hesitant. This is common so it is suggested that the facilitator be the first one to place an event lol. At this point, all participants should be encouraged to hang events once all the participants have placed the events that they could come up with.

Exercise moves to the next step. In this step, the participants discuss when each of the events occur and order the events across the timeline from left to right.

One important tip for the facilitator is that facilitators should always remember that their role is to facilitate and they need to let the participants do the modelling. Once the participants have ordered the events look for duplicates.

Step number three. Participants need to think about commands policies and actors to identify the cause and the effect of events.

These concepts are a little tricky compared to the domain events, so the point the facilitator is encouraged to ask questions to help the participants make some progress.

In step number four, the facilitator asks the participants to think about the business. This is the logic that responsible for creating the domain events. At the same time it is important for the facilitator to keep track of time. Every 30 minutes it is suggested that the facilitator review the progress and adjust the pace and the direction as needed.

Facilitator may ask the participants to focus, put people in flow. Focus may shift to the next event for which we don't have the domain logic information.

Post workshop activities carried out by the facilitator. Facilitator takes pictures of the workspace before taking down all of the poltter pare from the walls. It will be much more easier for the facilitator to go through the pictures rather than rolling out the long sheets of paper.

At the end of the workshop the facilitator asks the participants for their feedback. What worked well? What needs to be changed?

The expectation is that the facilitator will incorporate this feedback in their next workshop. Within 2 to 3 days.

The facilitator consolidates the knowledge and shares knowledge model with all of the participants. Facilitator requests the participants to go over the knowledge model to ensure its accuracy.

Facilitator decides on the next steps and ensures that all of the parking lot items are addressed.

Chapter 9

Microservices Data Management Patterns

9.1 Introduction to Microservices Data Persistence

Microservices can independently make changes to their own database without impacting the other microservices. Conversion of a monolithic application to microservices requires the designer to not only think about the refactoring of the business logic and the components of the monolithic application, they also need to think about the refactoring of the database. Refactoring the database would mean breaking the common database instance to multiple database instances

This breaking of common database instance into multiple database instance is not straightforward. There are multiple challenges.

Pattern that are covered here - The shared database pattern - Separate database pattern - Strangler pattern

9.2 Monolithic Apps - Shared Persistence Storage

The persistence of data is achieve by way of writing to the file systems or by more commonly writing the data to databases. There are multiple types of databases available today, but the most commonly used databases are the relational databases and NoSQL databases

Legacy applications typically use RDBMS or relational databases for all types of data and this is because NoSQL databases were not available until the early 2000s.

A typical client server application development pattern used by these legacy applications involved multiple enterprise applications, sharing a common instance of the database, and the enterprise application would manage the data by way of SQL statements against this shared database.

Buy todays standard this architecture will no be acceptable.

There are good things. First thing is that it will lead to simplified data management process. Second is that since the database engine is shared by multiple applications, the cost of the solution will go down. The third is that a single team of database administrators can manage the database for all applications.

Challenges with the shared database pattern. The first challenge is that database challenges need to be managed very carefully.

Consider the change of database schema. Very complicated coordination between multiple applications. Overall this will lead to high cost of changes. Will lead to higher risk because making a change to the database may lead to breaking of other applications. This will lead to longer time to value.

Second challenge is that on may negatively impact all other applications, and this can happen if the application intentionally or unintentionally start to use up a lot resources on the database. Hard to diagnose since each application will only be looking at their own application logs.

The third challenge is that the shared database acts as a single point of failure. Failure of the database will lead to the failure of all of the applications.

Could argue that a single point of failure may be removed by way of some kind of availability technology, by those high availability solutions are quite complex and they will lead to higher costs. Other challenges are that it is complex to carry out capacity planning for the shared database as the teams need to coordinate on forcasting the needs that they have from the database perspective. Application teams will need to be aware of the structure of the data, so onboarding new developers in the teams will be a challenge.

Shared database is an anti-pattern (Reaaak?) but some enterprises are still using it.

SOA originated as a solution but it was very temporary.

9.3 Service Orientated Architecure (SOA)

Data is exposed by way of CRUD services. CRUD..... Hide the details of the structure of the database from the applications..... Apart from providing these low level data operations. So as services may also be built by encapsulating the business logic to provide high level services. The services layer is placed between the applications and the database. These services provide the CRUD and high level business operations.

Applications do not need to use SQL to carry out the data manipulation. They connect with these services over some kind of network protocols that is decided by the service provider.

The server services layer provided multiple advantages since it hit the structure of the database.

The applications became much more simpler to code and manage. Led to the reuse of code by way of reusable services and the change to the database will become more manageable with the services layer. Unfortunately it does not address many of the other shared database challenges, such as single point of failure, uncontrolled use of resources and impact on other applications and capacity estimations.

Common misconception. A small service is not a microservice.

A set of SOA services are part of the same bounded context, whereas a micro service represents a bounded context.

Summary - Services insulated the applications from database changes, but SOA services did not address all of the challenges associated with shared databases.

9.4 Separate Database Pattern

Recommended for Greenfield microservices initiative. Greenfield - New application with no constraints from technical debt perspective Brownfield - Existing monolithic app to be converted to Microservices architecture

Microservices from the ground up. Microservices teams assigned the bounded context No interdependency between teams Each team decide on their tech stack Service interactions via defined interfaces No direct access to data

Separate database pattern Each microservice team owns and manages their database Benefits - Simpler change management Reduced blast radius on database failure Capacity planning - scaling at DB level becomes simpler Each team can decide on Database - doesnt have to be a RDBMS

Review - Separate database pattern is recommended

9.5 Brownfield Microservices - Database patterns , options

Share database pattern is considered an anti-pattern in the context of microservices. Strangler pattern is used for converting brownfield applications to microservices.

in a brownfield project, a monolithic applications is targeted to be refactored into microservices. The monolithic application will more that likely be a database instance. The designer of the microservice will have three options from the database perspective.

They can go with the separate database pattern wherein each microservice will have its own instance of the database, or they can retain the common database and do database refactoring to carryout logical separate of the database. In the last two options the shared database is retained.

Very commonly seen that legacy applications implement business logic as stored procedures. The idea here is that the databased has a lot of complexities and breaking the database into multiple databases may not be a straightforward task.

A common strategy that a lot of folks take for converting the brown field applications to microservice sis to simpley keep the database in place and convert the application to microservices. - This will lead to a set of microservices sharing the database and hence they will suffer the same challenges. Sharing an instance of the database, the shared database pattern in the context of microserviceS is considered an anti-pattern. The reason for it is that with the shared database, teams lose their independence and not they are interdependent.

Now any change they they will make will need to be coordinated. Changes would need increased testing effort and overall this will lead to the slow down of delivery of value and on independent scaling will be possible. The instance of the database may be the choke point, although a shared database is an anti-pattern from the microservices perspective, it is still okay to use shared database pattern for transition state architecture.

.... Start with a monolithic application Focus only on the application part of it and keep the database as is. You arrive at this transition state architecture where all the microservices are using a shared database. Gradually each of these microservices will be refactored to use their own database instance. This way the shared database is used only as a transition state, not a target state.

One important point ot note about the shared database pattern for microservices, is that the microservices should not have any SQL statement in the code. The reason is that having the SQL statement in the code will require more effort for switching the database.

Instead, the microservice designer should consider using the strangler strategy.

A collection of services is created for providing the database access to the microservice. Each of these microservices depicted here will have their own set of services. These services are just referred to as the strangler services or just stranglers. The idea behind the services is that switching off the database will have no impact on the microservices, as all of the changes will be confined to the strangler services. The microservice teams can decide on their priorities and switch databases at their own pace. Over a period of time, all of the microservices will have their own instances of the databases. As a result the target state architecture will be achieved. The strangler pattern is a generic pattern Can be used of modernisation of any legacy services.

A big benefit of this approach is that microservice code will be insulated from all backend changes as these changes will be confined to the services layer.

Key points - The shared database pattern is considered an anti-pattern from the microservices perspective. - Strangler pattern may be used for transitioning from the shared database implementation of the microservices to separate database implementation of microservices.

9.6 Shared Database Pattern

Two options for considering when building microservices against shared databases.

First one is the database refactoring and the logical database separation. Although shared database pattern is considered as an anti-pattern in the context of microservices, there may be times when the microservices teams will not have the flexibility to use the separate database pattern. There may be multiple reasons for it. - Time constraints, budget constraints, lack of skilled resources (hmmm) and the complexity of the database are some of the most common reasons. In this scenario teams have to consider the shared database pattern. When using the shared database, the objective of the microservice designer is to achieve maximum isolation of data within the same database instance such tha teach microservices owns part of the data in the database and each microservice ice has access to only the data it owns.

If possible the microservices designer should use the database features to achieve the highest level of isolation between the microservices own data within the database instances.

Chapter 10

Microservices DB Performance Management

10.1 Need for more Data Patterns

One common technique used for addressing the database bottleneck is to separate the database instance into write optimised and read optimised and read optimised database instances. It is very common to see multiple read instances in large scale applications. Clarity: This separation the databases is different from separate databases for the microservices. This mechanism of separating the databases into read and write optimised instances will work as long as the data integrity is maintained across the multiple instances of the databases. In other words, when the data gets written to the write optimised database, that data is consistently reflected across all the copies of the read optimised database. To achieve this, there is a need for additional patterns.

Patterns to ensure data integrity across multiple databases. Command query separation, command query responsibility segregation pattern (CQRS), Event Sourcing. Potential tools: RabbitMQ, PostgreSQL and MongoDB

10.2 Commands Query Separation (CQS)

An operation is either a command or a query. An operation refers to a message that tells the application or system to do something. If this message leads to the change in state of a domain object then this message is referred to as a command. If the intent behind the message it o get the state of the domain object, then that message is referred to as a query. A query message does not change the state of a domain model.

The command Query Separation principle also known as the command query separation pattern suggests that the designer of the domain object shoudl demarcate the methods as either commands or as queries. In other words, same method cannot be used for both command and queries. Idea was suggested by Burton Myers in early 2000

Since this principle leads to the simplification of the domain model, it has been very well received by the practitioners.

Refresher - Command as part of the lectures on an event. Storming a command is an action initiated by user system service or is triggered by some mechanism in time. A command is imperative, meaning that the action or the intent in the command must be carried out in the domain.

Commands are always named as verbs. Similarly, read models also referred to as the query models are used for projecting domain data on the user interface. The reasons behind the command query separation is the fact that there is a different set of concerns to be kept

in mind when you are implementing the command and the query operations. When executed these command should not lead to any kind of data integrity issues. Last but not the least these functions also have to deal with the challenges of scalability.

At this point, you must be clear that commands are the ones that write to a database and the queries are the ones that read from the databases.

Typically, the commands and queries are part of the same component and they are built and managed by a common team.

Key points from this lesson. Commands represent the key points from this lesson. Commands represent actions or intent.

Queries service domain data and the command query separation pattern suggests that operation exposed by the domain object should either be a command query.

10.3 Realisation of Commands and Queries

The commands and queries may be exposed by way of synchronous as well asynchronous protocols. In this microservice there may be a domain aggregate that is exposing the commands and the queries which may be invoked by way of a HTTP or gRPC API Some of these commands and commands and queries may also be executed by way of a messaging protocol. The format of the reqeust and responses for commands and queries is flexible. It may be JSON, CSV or XML protocol, buffer format or any other format that the designer of the microservice format that the designer of the microservice finds to be suitable for the requirement. A command writes to the database and the query reads from the database and the query reads from the database before the command writes to the database. It needs to translate from the command domain model object to the language that database understands in the context of the commands. The common model is referred to as the right model or sometimes just right side. Similarly, when the query is read, the data from the database it converts from database representation to the common domain model object representation. In the context of queries, the domain model is referred to as the read model or just the read side.

Pseudocode for the command and they query processor actor invokes the command, which is received by the command processor. The first thing the command processor does is it converts the reqeust data to domain model as a next step. It executes the business logic and then converts the model to the database representation and then writes the data to the database.

On the query side, the query processor receives the request and converts the and converts the received request to domain model. Then it gets the data from the database, converts the response data to domain model and then sends its back to the caller of the query.

One point to note is that in a typical system a common read write model and a common write database is used for both commands and queries. It is commonly seen that there are more query requirements that the write requirements.

It is more commonly seen that there are more query requirements that then write requirements.

Typically you would find that systems provide multiple queries that serves up the same data in multiple formats or multiple views. The challenge with this is that there is a need for the common model to be changed for meeting all of these requirements. To achieve an optimal performance level for the various read requirements or the query requirements, the microservices designer needs to think about the indexes they need to create in the database. The challenge with indexing is that it leads to negative impact on the performance of the writes.

Now for most systems ther performance hit on the writes may be acceptable but for internet scale systems or high volume systems. The performance hit on the writes may be acceptable, in which case alternate designs may need to be considered.

Term collaborative domain. A collaborative domain is a domain in which multiple actors invoke commands in parallel on the same data. Collaborative domains have low tolerance for performance degradation. Commands in such systems have complex business logic and the commands are implemented with finer level of granularity. Also, the business logic and the change frequently in collaborative domains.

Key points Common model is used for commands and queries. It is common for most systems to have multiple query requirements and these multiple query requirements may need to a change in the common model. Indexes are commonly used for making the queries performant but that impacts the performance of the write collaborative domains have little tolerance for performance degradation and so for collaborative domains, alternate designs need to be considered for implementing commands and queries.

10.4 CQRS - Command Query Responsibiltiy Segragation

Applies the CQS idea to Domain Model rather than operations. Here instead of splitting the operations in to command and query, the suggestion is to split the domain model into the command and query models. The command model is also referred to as the right model and the query model is referred to as the read model. Idea was suggested by Greg Young.

A more formal definition of CQRS suggests the use of separate read and write models for the realisation of collaborative domains. The typical characteristics required form the command model are finer granularity and support for ever changing business rules. Similarly the desired characteristics for the read model are that it should be able to meet different read requirements and provides very high levels of query performance. The read and write models may use the common instance of the data store. This may lead to the loss of ability to optimise the database for reads and writes. Note: If you create too many indexes on a database then the performance on the write side will degrade. —— If you remove indexes from the database if will impact the query performance. —— Balancing/ Optimisation act for use case.

What if you need both? One solution to both would be to use independent data store for the read and the write side. Not only that, depending on the use cases for the read and the write side, different database technologies may be used. Use of different database technologies within the same domain is sometimes referred to as the polyglot persistence.

Example of why it may make sense to go with different database technologies. - Say requirements for the write side is to have ACID properties for the transactions and the desire is that no indexes should be created on the database tables to achieve high throughput for the writes. - On the read side, say the requirement is to have highly performant relationship queries. In this scenario, the developers may decide to go with a RDBMS database on the right side and a graph database on the read side. With this setup the developers will be able to achieve the requirements for both the read and write models. Another advantage of using independent data stores on the read and write side is that each side may scale independently. - Example, developers are dealing with a read heavy application. — In that scenario they have the choice of horizontally scaling the read side or if that does not suffice then they can consider the read replicas for the read database or use sharding. — Since the two sides are independent, actions performed on one side would not impact the other side.

Read performance considerations. Typically there are more reads than writes in applications. Authors Experience: 85% reads to 15% writes as a result. Application developers are always looking for options to achieve highest read performance. Common ways by which it is achieved is by way of caching or by they way of materialised views that would not require model translation or use complex joins as the data is already persisted in the format in which it will be served to the requester. Sometimes this may not suffice. Example: There may be multiple different types of queries that need to be fulfilled by the read side. For example, there may be multiple different types of queries that need to fulfilled by the read side. – Example of

such queries are free text queries and reports form multiple data sources. In this scenario, the same type of database on the read side will not be able to efficiently handle all different types of queries. – In such cases we may need to further break the read model into independent models and use different data stores.

In certain scenarios involving collaborative and dynamic applications, there may be a need to frequently make changes to the read and the write models for such applications. Firstly consider leveraging different teams for managing the read and write models independently. With this kind of a setup, the teams can manage their own code base. They do not need to collaborate with other teams to make changes to the code bases that they're managing and all other deployments can be independently carried out by the owner team.

This is an extreme case of an application that may have some stringent requirements in terms of availability, speed to market and performance.

Common misconception - The common misconception is that in domain driven design CQRS must always be used. This is not correct. — Depends on the use-case and yor objectives.

Certain consideration that you need to keep in mind when deciding between using or not using CQRS.

The first one is that you must be able to define the advantages of using CQRS. Bear in mind that using CQRS would mean a higher cost solution with more moving parts and components would need to be managed over the lifetime of the application.

Key points May consider using CQRS for collaborative domains with - The write side and the read side may independently managed form the performance scalability and changes perspective.

10.5 Data Replication between WRITE-READ sides

Needed for independent READ and WRITE data stores

This data replication may be synchronous or it be asynchronous. In the case of synchronous replication, there may be some technical challenges as synchronous replication may not be available natively for the database tha you are using. Or if you are using different database technologies, it may not be possible. Event with natively available synchronous replication technology there may be impact on write performance.

Example: AWS DB that supports read replicas in the same AZ with synchronous replication. This kind of mechanism is supported for multiple databases such PostgreSQL, MySQL and SQL Server and Oracle.

Asynchronous replication. - The write side writes to its data store and then some kind of asynchronous replication mechanism kick in, picks up the data from the write side and updates it on the read side. As a result there may be some lags and the data is eventually consistent. The implication of eventual consistency is that the read side may not reflect the current state of data. There are multiple data replication options available and the decision on which on to use will depend on the use can and the non-functional requirements such as the consistency of the data. - Some applications may not fulfill the requirement if they have eventual consistency from the read end. In that case you would have to go with some kind of synchronous replication technology.

Data replication options. Capture changes in one data store and store in the other data store. Can use native replication such as Amazon RDS, MySQL replicas. Third party tools: Such as AWS data migration services. There is messaging streaming based replication in which the write side emits the data updates as events and the read side recieves these events to update its own data store. (EDA)

Summary Data replication between read and write data store is needed when you segregate the command and the query and use different data stores. This application may be

synchronous or asynchronous. Would need to look at specific requirements and supporting technologies to decide which replication technology which would make most sense for a use case.

10.6 Event Sourcing and Event Store consideration

State Management - State-Orientate persistence system maintains only the current state - Event sources persistence systems persist all state changes. — Domain events are stores as they are received

Event Sourcing - Event Sourcing suggests persisting the domain events in an Event store and using it for recreating the state of the domain objects at any point in time. - Event store is an append only event log used for persisting the received events

Performance consideration - Recreating the state from events will lead to bad performance Can be fixed by storing the state in a separate data store Multiple data views may be created for optimising queries

Benefits - Event replay to create ”point in time state” - Multiple read models views - Accurate out of the box auditing - Simplified Reconciliation - Temporal and Complex historical queries

CQRS and Event Sourcing - Commonly used together - Purpose built data store can be added for specific requirements

When to use Event Sourcing? - Evaluate use case from the event sourcing benefits perspective — Event replay to create ”point in time state” — Multiple read models — views — Accurate out of the box auditing — Simplified reconciliation — Temporal and Complex historical queries

Realization of Event Store - Traditional datastore’s may be used. — RDBMS - PostgreSQL — NoSQL Database - MonogoDB — Specialised databases - EventStore - http://eventstore.com

Quick Review - Event-Sourced persistence systems persist all state changes —- Current state managed in a separate data store for performance —- Out of the box Audit, Reconciliation, Temporal Queries

Events are stored in Event Stores that my be traditional DBs

Chapter 11

Microservices Managing the Data integrity

Using messaging there is potential loss of the message if the MQ or the messaging broker is unavailable. The write side, writes the data to its database but is unable to put the message on the queue. Now the read side will never recieve the message.

As a result the data across the two database instances is now in an inconsistent state. This kind of data loss may be prevented by using a reliable messaging pattern. In this pattern, the message is guaranteed to be delivered. The idea is that when the right side encounters a failure on the message sent, it continues to retry until it is successful in sending the message.

These retries may cause a delay in getting to a consistent state of data across the databases instances, but the data will eventually be consistent across the database instances.

Another scenario that my lead to inconsistent state. In this scenario the right side sends duplicate messages and the read side processes the same message, more than once and that may lead to inconsistent state.

11.1 Designing for failure

The concept of design for failure suggests that you should always anticipate that there will be failure. As a designer of the software, you should identify the failure points in your architecture.

Consider where are the failure points? Data base may go down or even the network may not be available to a microservice inorder to connect with the message bus etc. Even if the network is available, the external service may not be available. So the suggestion is that once you have identified the failure points in your architecture, proactively address the failure poitns.

Note: The best way to find out all the failure points in an architecture are to assume that there will be failures in all interfaces and components. Database may go down. Run out of resources, command object may not be able to push a message to the MQ due to the failure of the MQ server. May be failures on the read side.

As a designer, once you have identified the failure points you need to think about the impact of those failure points.

Example solutions: Write to the database and publish a message to the MQ in a single unit of work or transaction. Two phase commit is a mechanism that can be used for carrying out the write and publish in a single unit of work or transaction. Popular for the last three decades. In two phase commit a distributed algorithm is used for coordinating all processes involved in the distributed transaction. It is also referred to as the extended architecture or just ECS for short.

In the two phase commit, there is a transaction manager that coordinates the transaction across all of the involved resources.

The challenge with two phase commit is that it is quite complex to implement. A bigger issue is that a lot of distributed technologies do not support it.

Due to these challenges the two phase commit is not very popular with the designers of distributed systems.

An alternative would be to break the database write and publish steps into steps and use local database transactions. In this courses it is called the ”reliable mesagin pattern”. In this pattern, the right side writes the domain objet data and the event data in the database tables with a local transaction. Then the evens are replayed against querying system in a separate step.

Key points. You must identify failure points in your architecture and you must proactively address those failure points. This is the concept of designing for failure.

Two phase commit can be used for executing distributed transaction across multiple resources. Keeping in mind there are multiple challenges in using two phase commit for preventing the loss of messages or events.

You may use reliable messaging patterns.

Chapter 12

Microservices and Kafka

12.1 Use of Kafka in Microservices

Kafka is a streaming platform, which is extremely propular with microservices designers and developers. Many of the Kafka's features overlap with the capabilities offered by other messaging platforms such as RabbitMQ and ActiveMQ.

A microservices designer needs to make a decision on whether to use kafka streaming or to use a messaging broker like RabbitMQ or ActiveMQ for their microservices. To answer this question, the microservices designer must understand the capabilities of Kafka streaming versus the capabilities offered by these message brokers.

12.2 Kafka Overview

Kafka is a high performance, open source distributed open source event streaming platform. Kafka can ingest up to 2 million messages per second. (Hmmmmm) Kafka can ingest consists of multiple nodes or machines that are spread across a wide network. Event streaming here refers to the PubSub messaging model. At a high level the Kafka cluster exposes pubsub messaging model. At a high level the Kafka cluster exposes PubSub messaging capability to the producers and the consumers.

Looks very similar to a typical messaging platform and you're right but this is where the similarities end. Kafka cluster may consist of thousands of topics. The message data in the topic is spread across multiple partitions. Think a partition as a shard. The data in the partition is replicated across multiple machines in the Kafka cluster. As a result the Kafka cluster is highly fault tolerant. Kafka was developed at LinkedIn (noice) and open sourced in 2011 Kafka was built for scale from the ground up. (Do research on numbers when required)

Capabilities of kafka It provides high messaging throughput with latencies as low as two milliseconds Its highly scalable Can easily add more machines or broker nodes to the cluster to scale it as per your requirements. Can handle trillions of messages per day and it can handle petabytes of data. Kafka cluster stores all the messaging data in persistent storage so messages are not lost due to server failure. Its highly available. Data is replicated and failure of broker nodes does not impact the producers and the consumers. Visit kafka at Apache to read up more on aspects of Kafka and see industry usage (standard approach lol)

12.3 Kafka Concepts

To be combined with over Kafka stuff.

A Kafka cluster consists of one or more broker nodes in production. A minimum of three broker nodes are recommended when a topic is defined on the cluster. That topic is replicated across all of the brokers. The topic data is partitioned into multiple partitions which are replicated across the broker nodes. Producer connects to the cluster and publishes a message. Consumers connect to the cluster and get the data pushed into the partitions for specific topic that they have subscribed to. Let's dive deeper into the details of how the Kafka cluster works. When a message is published by the producer that message gets added to the partition and the partition is then replicated across multiple brokers.

Example: Topic has three partitions and all of these three partitions are replicated across the three brokers. Since the partition is replicated across three brokers the replication factor is set to be three. If we had configured this cluster to replicated the partition only across let's say two brokers then the replication factor will be set to be to the benefit of this replication is that if there is a failure of one of the brokers the data in the topic will still be available and the consumers will not be impacted due to this failure. So the idea is the higher the replication factor, the more fault tolerant is your cluster. but obviously you will need to allocate more resources to your cluster at any point in time. Each partition has one broker node assigned as a leader and this leader broker node is responsible for all of the reads and writes to that partition.

So in this illustration for Partition one, let's say the leader node is broker one. In that case, whenever a message will be published to topic A, and if the data gets added to partition one then the broker node will be responsible for replicating the data for partition one to other broker nodes in the cluster. In case of failure of the leader node one of the other available broker nodes take up the role of the leader for the partition.

All of this happens behind the scenes and is transparent to the producers and the consumers of the messages. Message data is partitioned on the message key. This message key is provided by the producer as part of the publishing of the messagee. The message key is optional.

Scenario in which the producer provides a message key. Lets say the producer publishes a message key. Producer publishes a message for and ID - 1,2,3 as the key. In that case the broker calculates the hash value for the key provided by the producer and then the broker uses the hash value to determine which partition the message data will go to. In this case broker pushed the message data into partition one. Broker will calculate the hash value and may decide to push this message. TLDR - every time the same key is used the message will be pushed to the same partition. Say another message is published by the producer with key equal to customer ID one, two, three then that new message will end up in partition one.

Scenario in which the producer does not provide the key. The broker carries out the partition assignment in a round robin fashion. That means is that the first message will end up in partition number one, two will end up in partition two, third will end up in partition number three etc. Typical messaging systems do not retain the order in which the messages are sent but in the case of Kafka messages are ordered within a partition. —- Key here is to realise that this ordering is only guaranteed within a partition. The other way of saying this is that messages ARE NOT ordered across a partition. Ordering guarantee is one reason fro picking up Kafka over a typical messaging platform. Each message in a partition is assigned an offset within the partition. Array numbering within a partition. A kafka cluster allows maximum of one active consumer per partition.

The reason for this is that with a single consumer the order of the messages is guaranteed. Kafka cluster maintains the current offset of the message that has been read by consumer. As new messages arrive, the current offset changes for consumer A and consumer B by maintaining the current offset on per consumer basis. Kafka is able to avoid sending duplicate messages to the consumer. Message reads in Kafka are non-destructive. That is when the consumer reads the message from the topic, the message is not deleted. Only the offset gets

updated and consumer can at any time reset the offset. What this means is that by resetting the offset, the consumer can replay the log messages from any offset. Messages on the kafka topic are not retained indefinitely. There is a retention period which is associated with the topic that basically defines the time to live for the message after the retention period has expired. For the message, the message is automatically deleted from the topic.

Although Kafka maintains the current offset on per consumer basis. Consumer also has the option of managing the offset on its own outside of the Kafka cluster. In messaging systems, it is common for multiple instances of the consumers to read messages of a common queue. This is done for high performance and high through put in Kafka. It can be done by way of grouping the consumers and the idea is that each message in the topics is received by only one of the consumers in the group. This grouping of consumers is carried out by way of groupID.

Kafka allows only one consumer per partition the maximum number of consumers in a group is equal to the number of partitions. We have three consumers so we can only have three partitions. If a fourth consumer is added then it will go into a wait state which means it will not receive any messages until one of the consumers die or stop listening to messages.

Key points. The message is retained in the Kafka topic Partition for the retention period define on the topic. Topics are split into partitions. Producer can specify the message key, which determines the partition to which the message data gets added. Kafka maintains the offset of the last read message on per consumer basis. Consumer can reset this offset. The consumer can maintain the offset on its own. Consumers can read the messages from a topic as a group by way of groupID

12.4 Kafka vs AMQ (Rabbit MQ)

Consider this question. Which should be used for microservices. When a message is sent to a Kafka topic it is always persisted in the long term. Storage message expires after the set duration on the Kafka topic. Messages are not removed on read. What that means is that the consumer of the message can reread the message now comparing AMQ. Once the message is read from a queue on RabbitQM or ActiveMQ the message is deleted from the queue. Kafka uses a custom binary over TCP — Important to not that it is custom. What that means is that it is not a standard protocol which is in use my multiple products (Think FIX) This particular protocol is in use only by Kafka On the other hand AMQ is a standard protocol that is implemented by multiple messaging platforms such ActiveMQ and RabbitMQ. Although kafka is a messaging platform, it does not have a concept of a queue. It supports only publish subscribe messaging pattern whereas AMQ based messaging platforms such RabbitMQ support bot point to point and publish subscribe messaging patterns. Kafka does not support routing. You may use dynamic routing but it is not available outside of the box. On the other hand AMQ has a very flexible routing mechanism. Refresh - in the case of RabbitMQ we could create exchanges of different types and then associate queues with the exchanges with bindings. This kind of a mechanism is no available on Kafka.

In other words kafka has no concept of exchanges, queues, binding or message priority. Kafka consumers always pull messages from the broker by way of polling. Consumers subscribe for the messages and pull for specific duration and receive messages in batches.

On the other hand in AMQP based platforms there is support for both push and pull models for receiving the messages. Kafka by default guarantees message ordering within a partition.

Guaranteed message ordering implies that consumer A will receive all messages in the order they were published. THis is an extremely important feature of Kafka as it will allow you to build systems that require message ordering. Important thing to keep in mind is that AMQP based messaging platforms do not support message ordering. In the case of RabbitMQ, message in a queue may be read by multiple consumers. Each of these consumers process the recieved message independent of the other consumers. As a result, the order in which

these messages get processed is not guaranteed, wheres in the case of kafka there is only one consumer per partition. As a result the order in which the messages get processed is the same as the order in which the message were published.

Key points When the messages are read by the consumer from a topic the messages are not deleted from the kafka topic. They are retained. Kafka does not support the AWQP concepts such as queues, exchanges, routing priorities etc. Kafka uses a custom binary TCP protocol. What that means is that there is not other platform that is using the kafka protocol. Kafka supports only pub sub and pull based message receive. Kafka guarantees message order within a partition.

Chapter 13

Managing Distributed Transactions with SAGA

13.1 Distrubuted Transactions with SAGA

13.1.1 Motivation

> **Note:** Recap: it is not always possible for microservices to use a two phase commit for transactional management.

The alternative is to use the Saga pattern. The challenge with building sagas is that they are complicated.

The challenge with building Sagas is that they are complicated, but there are frameworks for it (Reactive Specification and WebFlux). These frameworks hide the complexities needed for the realisation of this pattern.

13.2 SAGA Pattern for distributed transactions

Refresh - two phase commit which can be used for managing data consistency within a single data base. In the case of distributed transactions, there may be multiple databases across multiple microservices.

The design for failure principle suggests that you should always anticipate that there will be failures. You should proactively identify the failure points in your architecture involving distributed components of microservices. Then you should address of the failure points in your architecture. This is the best practice for building a distributed systems.

The idea is that since each microservice has its own database, if failures are not addressed it will lead to an inconsistent state of data across the multiple databases owned by the microservices.

> **Note:** Important: This needs to be avoided at all costs.

The designer of the distributed system must ensure that the data is consistent across all of the distributed services or microservices. The State should be consistent across all databases. It can be all successful or it can be all failures, but it cannot be a mix of success and failure.

> **Note:** One important point to keep in mind is that local transactions cannot be used for reverting the state of the database.

The Saga pattern provides a solution for managing data consistency across microservices. It involves use of local transactions coupled with compensating transactions.

The role in compensating transactions is to revert the database changes. Via compensating transactions the consistency of data will be achieved across all databases. This means each transaction requires a compensating transaction.

The Saga pattern may be applied to monolithic and distributed systems. When applied to the distributed systems it is sometimes referred to as distributed saga. In microservices we are referring to distributed Saga.

Saga is not new. Introduced in a paper in 1987 (add link).

13.2.1 Section Summary

A Saga pattern is used for managing data consistency across microservices. Involves the use of local transactions data to the databases, and it involves use of compensating transactions for reverting the database changes in case of failures. The reason compensating transactions are used is because it is not possible to use database transaction rollback for reverting the database changes.

13.3 SAGA Choreography vs Orchestration

There are two ways to build saga call flows.

A Saga is implemented in two ways.

- The first way is to use what is referred to as event choreography, in which there is not central component to manage the transactions.
- The second way to do it is by way of command orchestration in which there is a central component for managing the flow of Saga.

The central component is referred to as the Saga Execution Coordinator.

13.3.1 Details

In the event choreography saga, the microservices emit and recieve domain events and these events are the processed by each of these services independently, Services may listen for events of interest and react to those events in an autonomous fashion and as part of transaction processing these services may emit new events. The use of event choreography leads to highly decoupled services.

Challenges

There are challenges. The first one is that is difficult to implement test and debug. Next one is that events are emitted and consumed by each of the services in the saga in an independent fashion and that can lead to out of sequence events. Each of the services in the event choreography needs to make sure that out of sequence events are handled properly. Handling the failure scenarios in event choreography is difficult because there is not centralised service.

Coordinating the recovery from failure scenarios in a command orchestration saga implementation. A central component manages the calls to services in Saga. There are two ways in which command orchestration saga may be built int he first method a domain object, which is part of the domain object model, acts as the central component for managing the calls to other services. In the second method, there is a dedicated orchestrator which is defined outside of the domain model to coordinate the calls to services in the saga.

The central component is sometimes referred to as the Saga Execution Coordinator.

An external coordinator is preferred over coordinator built into the domain object. The external Execution coordinator is also referred to as the orchestrator. It is a generic component that does not have any kind of business logic.

It provides certain capabilities to define the flow and it carries out the invocation of the services, by way of the flow defined by the designer of the microservices. It also carries out the state management in its own dedicated persistent storage that has not relationship with the domain data storage.

There are multiple external orchestrators that may be used for building Saga with microservices. - Examples are a BPM or a workflow tool. - Step functions are another one. — If you are using cloud then this is a quick way to build sagas. - Then there are multiple frameworks in different languages. Spring can be used.

Step Functions

AWS Step functions let the develop define the process by way of JSON. There is also a visualisation tool that shows the flow service calls. Typically the business logic, which is the transaction and the compensating transactions are implemented in lambda functions. The step function service in AWS coordinates the executions of the transactions and the compensating transactions.

These are the services that are called as part of the Saga and these are the compensating transactions which revert the effect of the service calls.

The command orchestrator based Saga implementation is less decoupled compared to the event choreography based saga. Also the command orchestrator introduces a single point of failure. Even with those weaknesses, command orchestrators are much simpler to work with since there is a central component that defines the flow of the services calls. Developers can quickly understand how the command orchestrator saga is working and as a result they are simple build test and manage.

Failure scenarios may also be handled much more easily in command orchestrators compared to the event choreography based sagas. Some saga execution coordinators or workflow engines also provides a centralised way to check the state of the flow.

13.3.2 Section Summary

Building Sagas using the event choreography pattern. No central coordinator for the flow of transactions.

The other way to build Saga is to use the command orchestrator pattern in which there is a component that acts as the coordinator for the flow of transactions. A domain object may act as the Saga execution coordinator or an external component such as a workflow engine or framework may be used as the Saga Execution Coordinator.

13.4 SAGA Implementation Considerations

Aspect to consider - Synchronous vs Asynchronous calls.

Service in the Saga may expose commands by way of APIs. Example, REST APIs and these Rest APIs may be invoked by way of an orchestrator in a synchronous fashion. This is feasible...but asynchronous mechanisms such as messaging is the preferred way to go. The bottom line is that whenever possible use messaging instead of synchronous protocols such as REST over HTTP.

Each transaction in a saga should be assigned a unique identity (guid or some equivalent). Idea behind this is that it helps with even sourcing and the services in the saga may use it to

identify duplicate transactions. The idea behind this unique identity is that it helps with event sourcing and the services in the saga may use it to identify duplicate transactions. The idea is simple, if the transaction with specific ID has already been processed then it doesn't need to be processed again.

Idempotence is the property of certain operations in mathematics and computer science where they can be applied multiple times without changing the result beyond the initial applications.

In a saga the service transactions mus te idempotent. The unique identity given to each of the message can help in achieving the important behaviour of the transactions.

How services may retain the processed transactions in a log. When a transaction is received, the service will check if the transaction ID has already been processed. If it has not been processed then the transaction will be processed. A response will be created and saved to the database and sent back to the called. Say another request comes in with a transaction ID that has already been processed. In this case the service will not process the transaction again. Rather, it will pick up the past response from the database and sent it back to the caller.

13.4.1 About Failures

Service request failures are okay if they are followed by compensating transactions, but the compensating transactions cannot fail. They must execute successfully. Otherwise data across the microservices will get into inconsistent state in order to build a capability for recovering from failed compensating transactions. Consider using event sourcing.

Recap - Step functions may be used for building a command orchestrator based Saga flows. If you aren't using AWS, you may still use certain frameworks that provide the foundational components for building distributed transactions. Example frameworks: Eventuate, AxonIQ, Seata —- These can be leveraged for accelerating the development of microservices.

13.4.2 Summary

Key points Messaging is preferred for the intercommunication between the services in a saga. Each transaction should have a unique identity which helps the services achieve idempotency. That means the same transaction may be executed without the loss of data consistency. Transactions can fail but, if a transaction fails then the compensating transactions must be executed. The compensating transaction itself cannot fail.

Chapter 14

Microservices and API

14.1 Microservices - API Realisation

14.1.1 API - Application Programming Interface

API - Application Programming Interface is an interface that defines the interactions between multiple software applications or mixed hardware software intermediaries. The focus here is on microservices, which can be thought of as independent applications. Microservices expose APIs for consumption by other microservices.

It is common for these APIs to be exposed over HTTP but messaging is the preferred protocol for these interactions between APIs. These APIs are a separate set of APIs that may be exposed to the external components such as user interfaces or legacy applications. Some of these consumers may even be outside of the organisation that own the microservice for such interactions.

HTTP is commonly used and the reason for that is that messaging would require dependency on a specific messaging technology. In turn this would make the use of the API very restrictive. HTTP being an open standard makes the API more available to any type of client that can make HTTP calls.

Microservices may realised these APIs in multiple ways. Designers of microservices commonly use RESTFUL APIs and GraphQL APIs at a high level. The different between the two is how they define the contract between the client and the server.

Both have pros and cons.

14.2 Introduction to REST API

Is your architecture useful? - consider the Richardson maturity model. Some developers think that they by using HTTP and JSON they are creating something that qualifies as a RESTful architecture. That may or not be true. These applications may not be RESTful but they may be considered ”like REST” or RESTish

What makes an architecture restful? In order for an architecture to be restful it must follow six design rules. These six rules are referred to as the rest Architecture constraints.

Note: Described by Roy Fielding in his Dissertation in 2000.

Six constraints - consider researching if required.

- First constraint for REST is client server. This constraint suggests the use of client server design principles for impllementing the rest APIs
- Second one is the uniform interface that suggests use of well-defined contracts between the client and the server.
- The third one is the statelessness, which suggests that the server should not manage the state of the application.
- Fourth one is caching - the server should use HTTP caching headers to cache responses to the requests received from the client.
- Fifth one is layering - this suggests that the architecture should be layered and each of these layers should be manageable independently
- Last one is code on demand - this suggests that a server can not only send dataa to the client in response to the request but it can also send code that the client can execute

Note: The final one is optional.....but interesting :D

14.2.1 Summary

To summarise - as long as your implementation is following the first five constraints you architecture will be considered as RESTful. Recall that Rest APIs are not restricted to the HTTP protocol. So if your API is using HTTP and it is following rest architectural constraints then it is said to be an HTTP REST API or REST of HTTP.

14.3 REST API Resources and Design Constriaints

All real world objects or resources may be described by way of attributes. In software systems, these attributes are managed in some kind of persistent storage. This could be an RDBMS or a NoSQL database. Certain attributes of the object may change over a period of time. All state changes are captured in a persistent storage. What we are saying here is that representational state of the car is managed in the database and multiple instances are uniquely identified by some kind of unique ID. Each can have different attributes.

Think of a system that can query this database. The system may ask the database and will respond by sending back the representation state of an object or entity. This is the foundational concept.

From the source database, this representational state data may be in any format. This could be name value pairs or it may be in a database record format. This internal representational state is converted to other formats. Those are not the only formats it can be converted to. Representational state may be converted to JPEG, PDF, Excel, excerter.

If we replace the logic with the REST API, then it is a good representation of the restfulAPI from the data formats perspective. What this tells you is that a REST API is not tied to any specific data format. In fact the same instance of a RESTFUL API can convert and send back the representation state af a resource in different formats based on the clients needs.

All modern APIs use HTTP as the communication protocol and we call such APIs HTTP Rest APIs. The reason is because the rest APIs style itself, is not tied to HTTP. You can build REST APIs with other protocols as well. REST over HTTP APIs expose an endpoint and this endpoint is used for managing the state of the resources.

> **Note:** Keyword here is managing the state.

What that tells you is that you can carry out the CRUD operations on the resources exposed by the REST APIs.

You can create, retrieve, update and delete the resources in order to carry out the CRUD operations. The client has to use an appropriate http verb.

14.3.1 An example

Consider ACME travel HTTP endpoint.

- Client will use this endpoint to perform CRUD operations to create a resource (POST).
- GET to retrieve the HTTP resource.
- PUT for updating the state of an resource (consider idempotency).
- DELETE verb needs to be used for deleting a resource.

14.3.2 Summary - What makes an API Restful?

Summary - What makes an API Restful

- REST APIs - follow a set of design principles
- they follow the REST architectural style
- they expose resources one which the rest client can carry out operations.
- REST APIs are not tied to any specific technology
- REST API does not define any standard for request or response payloads or how the API should be built.
- REST APIs can use any communication protocol or any data format.

> **Note:** Note: Not tied to HTTP and JSON, common misconception.

14.4 API Management

APIs exposed by the microservices are consumed by three different types of consumers.

- First type are the private or internal API consumers. These private consumers are part of the same microservices team or are developers of other microservices. The idea is that the private consumers belong to the same organisation as the owners of the APIs
- The second type of consumer are the public or external consumers. These are independent developers outside of the organisation.
- The third type of consumer is the partner API consumers.

> **Note:** From the implementation perspective there is not difference between the three types of API. The difference is how they are managed.

Private vs Public.

Private API consumers could invoke the API more often then a public one. Another example is that a private consumer could have access to more features, wheres public consumer is restricted.

These aspects of an API are managed outside of the microservices implementation by way of an API management platform. Can think of an API management platform as a technology

that is used to address the common API concerns. These common concerns are typically the concerns related to the access and authorisation to the APIs logging and analytics quotas.

Microservices expose their APIs by way of these API management platforms.

- The API management platform exposes the API by way of a proxy or endpoint.
- The consumers of the API invoke the API by way of the proxy endpoint.
- They never connect directly to the API exposed by the microservice when the consumer invokes the API.
- The API management platform applies the management control and based on the outcome of those controls, the API management platform either allows the invocation oif the API on the microservice or it denies the reqeust for invocation.

14.4.1 Management control

Most API management platforms offer a declarative or policy based management features. What this means is that the developer of the microservice doesn't have to code any of these management controls. They can simply put together policies for defining these controls.

Example A policy may be defined for the internal or private consumer to be such that maximum flexibility is provided to the consumer. So in other words, most of their calls will go through the public domain developers, the policy may be extremely restructured to partner APIs. The policy may be such that it defines some kind of a SLA.

14.4.2 How are these policies defined?

How are these policies defined? These policy definition mechanism is totally dependent on the API management product. JSON is commonly used for defining these policies.

For further research look at Apigee, Mulesoft etc. To understand these JSON policy documents. Also check out cloud equivalents. The reccomended approach is to use an API management solution. There are multiple benefits, but one of the biggest is:

- The team can focus on the domain or the business logic, rather than spending time on common concerns such as security logging etc.
- As a result microservice code is cleaner.

Last but not least, since the consumers of the API do not connect directly to the microservice change management becomes easier. What that means is that the microservices development team can make the changes to the API and adjust the proxy on the API management platform to insulate the end consumer from those changes.

14.4.3 Key Points

- There are three types of APIs depending on the type of consumers
- The private API is consumed by the consumers within the same organisation.
- Public APIs are consumed by consumers that are outside the organisation and are in the public domain
- Partner APIs are used by the partners of the organisation
- Microservice expose APIs and the consumers do not connect directly to these APIs, but they connect by way of an API management platform that provides a policy based mechanism for addressing common concersn related to APIs.
- Since these common concerns are address by the API management platform, microservices implement only the business logic for the APIs.

14.5 Introduction to GraphQL

GraphQL is a query language for APIs that is not tied to any specific database or technology or network protocol. GraphQL is a specification for query language. At the time, the term GraphQL is also used for referring to a component that implements the GraphQL specifications. A GraphQL server exposes the GraphQL API endpoint that is used by the GraphQL clients to invoke GraphQL operations.

> **Note:** History: Developed by Facebook/Meta for use in mobile applications in 2012 and then became open source in 2015.

The are many small to large organisations that have adopted GraphQL as a standard for their APIs

14.5.1 REST vs GraphQL

REST API exposes the API interface to the client by way of contract. The contract defines the schema for the request payload that the client sends to the server and it also defines the schema for the response payload that the server sends to the client. The structure of the response and response schemas is fixed and the client receives all of the data in the response payload, *whether it needs it or not*. If you think about the REST API server as a data source then it is like executing SELECT * from the table. As the client has no choice in terms of what it needs in the response schema. This issue, where the REST API client is fetching more data then it needs is referred to as the *over fetching issue*.

14.5.2 GraphQL contract

GraphQL contract is in the form of a GraphQL schema that defines the operations that are supported by the GraphQL server. These operations can be query operations, which are equivalent to the retrieves, and the operation can be type mutations which are for updating, creating or deleting the data manged on the GraphQL server.

The GraphQL schema can also have type definitions. Clients can invoke the queries and tell the server what it needs in the response payload. Now the client has control over the response. This is like calling the SELECT with field names etc.

> **Note:** This way the client is able to get only what it needs and not have to be concerned with over fetching issues.

14.5.3 API granularity

Generally REST APIs are built as highly granular services and this is done to meet the needs of multiple types of clients. This leads to an issue. Under fetching, which results in more network calls from a microservice.

> **Note:** This leads to latency and a performance hit.

If this was to be replaced with a GraphQL, the api the provider will put together the GraphQL specification which will be used by the web developer to invoke the GraphQL queries against the API endpoint. In the query, the client side will specify the fields that it would like to receive as part of the response. Server will process the query and respond back with the requested fields in the response. With this setup, with a single call the client is able to retrieve the required data. Hence both the under fetching and the over fetching issues are addressed by graphQL.

A graphQL server is implemented as a layer to manage all the client interactions. it sits in front of the application tier. This layer implements the GraphQl specification and is responsible for managing the client interaction and an invocation of the API.

This layer interacts with the components in the application tier. The developer of the GraphQL API needs to put together the GraphQl specifications which are used by the GraphQL server layer.

> **Note:** There is no standard implementation of GraphQL.

There are multiple frameworks available for different languages. Each of these frameworks have a different requirement from the application perspective. Most frameworks require the developers to create GraphQL components that get wired to the GraphQL server, *so the components that the developer needs to put together depend on the framework*.

GraphQL operation flow.

In a typical implementation of the GraphQL server client invokes the operation on the endpoint exposed by the GraphQL server. The GraphQL server on receiving this invocation validates the request against schema definition. If everything is good, then it requests the application to create an instance of a component provides the implementation for the operation. This component is also referred to as the query resolver. The data fetcher component then interacts with the data sources, which can be a database or it can be other components. The idea is that the data fetcher retrieves the data from on or more sources. After receiving the data from the source. The data fetcher then creates one or more instances of the resolver. These resolvers are then returned back to the GraphQL server layer. The GraphQL server layer invokes the functions on these resolvers to get the data for the fields that need to be passed in the response.

The graphQL server implementation is available in multiple languages.

> **Note:** Link for further research: https://graphql.org/code/

GraphQL Advantages

Recap Already learned that GraphQL addresses the over fetching and under fetching issues faced by the rest API clients. The graphQL API client is in full control of the responses. Documentation for the GraphQL is provided in the form of schema, so there is not need for separate documentation. The graphQL server layer provides very descriptive error information to the clients which can be used by the client side which can be used but the client side to understand the exact issue in their requests to the GraphQL server.

GraphQL Disadvantages

Biggest one is that there may be performance challenges with complex queries. Web caching is not as straightforward to implement with GraphQL as it is with the REST APIs REST APIs are easy to learn compared to GraphQL APIs from the design perspective. REST APIs are defined in terms of resources and endpoints, whereas the GraphQL APIs are exposed over a single endpoint and the contract is in the form of a schema definition from the control perspective. The client side has no control on the response with a REST API, whereas the client has full control on the response in the case of graphQL. REST APIs expose crud operations using HTTP verbs whereas in the case of GraphQL all operations are invoked with either a HTTP GET or POST. There are three types of GraphQL operations: query, mutation and subscription. REST

API client needs to make multiple network calls to gather the required information, whereas GraphQL API leads to reduced network traffic and that may lead to better performance from use cases.

> **Note:** From a use-cases perspective, REST APIs are suited for resource driven applications whereas GraphQL is more suited for data driven applications

14.5.4 Summary

- GraphQL is a specification for APIs
- GraphQL addresses the under fetching and the over fetching issues related to the rest APIs.
- A GraphQL server implements the GraphQL specification.
- A GraphQL developer needs to put together the schema definition for the API and they also need to implement the components needed by the GraphQL server.
- The type of components that the developer has to put together depends on the GraphQL server in use.

14.6 GraphQL Schema Definition

See this link for details: https://graphql.org/learn

GraphQL and REST are not mutually exclusive. Pick the one that makes sense for your use-case. Evaluate requirements and then decide which one to go for.

The graphQL type system refers to the fact that a GraphQL service defines a set of types which completely describes the set of possible data you can query on the service. The incoming queries are validated by way of the schema definition language, which is *programming language agnostic*.

14.6.1 Schema Definition Language

Schema definition language is a JSON like language for defining the schema for the GraphQL APIs. A schema consists of two parts.

- The first part are the operations supported by the GraphQL server. These operations can be queries, mutations and subscriptions.
- The other part has the types which are the server defined objects and each of these objects have fields of specific types supported by the GraphQL schema.

In the context of schema definition, language operations are referred to as the *route types*.

Route Types

The route type query is used by the client for retrieval of objects defined on the server in the schema definition language. The route type query lists out all of the named queries.

Each query is defined with a list of arguments. These arguments are the query criteria.

A graphQL server may allow the client to modify the state of objects maintained on the server and it does this by defining the route types mutation.

Route Type Mutation

The route type mutation is similar to the route type query in the sense that there are named mutations in the specification and these named mutations also take an argument and return objects like named queries. - The last route type is the subscription route type, which is different from the query and mutation in the sense that subscription operation is not initiated by the client. - The client subscribes to the events on the server and the server pushes the data to the client in response to the events. - Types are used for defining the structure of the domain objects. - The scalar types are the standard types defined in the schema definition. - Language integer, float, string and boolean are foundational scalar types. - ID is a special type which is not part of the business domain data but is primarily used for managing the caching of the query data.

The GraphQL service defines the complex types by combining them with the standard scalar types. This no different from the way you define objects in any object orientated programming language. Language fields may be thought of as attributes in an object. These attributes have a type which may be a scalar or complex type.

The provider type is defined separately in the schema definition by the server. What this means is that complex types maybe nested and contain other complex types.

> **Note:** Key point: Deep nesting of object definitions can impact the performance of the server. This is because deeply nested complex types would require hierarchical resolver implementations and that will in turn also lead to the complexity of implementing the GraphQL server.

Things that you need to keep in mind when designing types

These are certain things that you need to keep in mind when designing your types.

Client query execution on GraphQL server. The first thing the client needs to do is prepare the request payload. This request payload is a JSON like document. The structure of this document is defined by GraphQL and the client has to comply with it. The first thing you see here is the route type which is set to query. This could be mutations or this could be subscription as well. Then there is a named query.

The products query is defined in the schema definition to take multiple optional arguments since for the products all of the arguments are optional. The client may specify zero or more argument values. These arguments act as the criteria for the query.

What makes GraphQL different from a restAPI query?

The client can specify the fields that it desires to receive from the server in the response.

> **Note:** This is a key point of GraphQL. *This cannot be done in REST APIs*.

Example flow

Example flow GraphQL server will first validate the reqeust. The server will delegate the reqeust processing to the microservices components. The microservice will the create the resolver instances and pass those back to the GraphQL server The graphQL server in turn will create the response payload based on the reqeust from the client and the reqeust from the client.

GraphQL ecosystem has evolved (check this) Tools exist to assist with GraphQL development. Example graphQL playground.

Summary and Key Points

Summary:

- Think of the schema for your graphQLs APIs.
- Think of a schema definition as a shared language.
- Don't forget the ubiquitous language for the domain.
- Make sure that you schema definitions are mapping to the terms in the ubiquitous language
- Take an evolutionary approach to create the API
- Don't try to do everything at the same time

> **Note:** There idea here, is to publish version on of the API and then look at how the clients or the consumers of the API are using your service and based of that, evolve the schema for your APIs.

Key Points

- Schema definition language is used by the server to define the schema or the contract for the API server.
- Uses the schema to validate the incoming request for operations
- Server also uses the schema for creating the responses that it sens back to the client.
- Client uses the schema to create the reqeust payload and to parse the responses it receives from the server.