**Contents**

**Abstract**

About the traditional design and why BCRS

**Motivation**

1. DDD (Domain Driven Architecture)
2. Bounded Context Mapping
3. Customer journey example

**Intent and Applicability**

1. Intent explanation
2. Applicability

**Real Time Scenario**

Scenario with example

**System Design:**

1. Separation of concerns
2. Scalability and performance
3. Configurable

**See Also**

Links and appendix

**ABSTRACT**

As the system becomes more and more distributed it becomes almost a necessity to break business in domains and those domains will be designed and implemented using microservices architecture as Bounded Context.

We have several patterns around how we can efficiently utilising microservices architecture basis business requirement. We will be focussing on pattern DDD, CQS, CQRS which we want to utilise in our design pattern.

For DDD we need to cleanlily break our business in clearly specified domain each having its own bounded context assignable to team to work independently.

CQS (Command Query Segregation) pattern tells to have separate methods for querying so that query method will not mutate the state and write will mutate but will not return anything

CQRS (Command query Responsibility Segregation) is about segregating the read and write requests. Basic difference between CQS and CQRS is read and write having its own objects to handle request.

CQRS is bifurcation as higher level i.e. bounded context instead at component level

This paper will describe about segregating read responsibility which can be pull and push.

**Why BCRS?**

While working on one of business requirement on customer journey we encounter a scenario where our sales team needs notification as soon as customer completed feedback in a session so that they will derive something out those feedback events and offer him a better deal for being a possible lead and sometimes we need to publish these customer journey events to outside organisation as they want to know about customer journey through our platform.

We already have separate read store for fetching those events on query but that’s not enough as client has to poll these records after every given time slot and if the frequency of polling is fast with higher number of client then polling doesn’t look to be the right solution. So, we must need a separate mechanism to publish customer journey events notification to other bounded contexts which will save a lot of unnecessary dumb polling and we can also easily manage our subscriber through notification service.

We haven’t segregated the responsibilities of query side which should be segregated to further read data and publish notification.  
  
So we created separate notification service for configured subscribers for publishing these customer journey events.  
  
**Note** : This pattern is targeting to specific problem where system need both push notification and read data at a scalable level

**Computing Classification System (CCC) Concepts**: Software and its engineering 🡪 Software creation and management 🡪 Designing software 🡪 Software design engineering

**1. Motivation**

Before directly jumping into architectures let put some limelight around the origination/motivation behind this pattern so that audience will co-relate with the real problem easily.

From many years Divide and Conquer used for dividing the complex set of problems to sub problems until it would pretty easy to write, maintain, collaborate and then rewritten in case we need it.

Divide and conquer is everywhere either its business where we are break entire business in domains. Back in time when Domain Driven Design gaining popularity over monoliths for better understanding rules and processes domain can contains.

Fig 1.1 Let take an example of customer journey domain driven design (DDD)

Diagram

Description automatically generated

Fig1.1 **Flow from customer journey to Bounded Context Mapping**

**Customer Journey**:

Customer journey is way to analyse and full fill the gaps customer is facing while experiencing the product

**Produce Domain Driven Models:**

Functional breakdown of system is necessary to understand it fully. For customer journey example it’s important to see the present and future requirement of customer. After thorough analysis of system and customer requirement we need to document domain model in ubiquitous language must be understand with technical and non-technical teams

**Define Bounded Context:**

No strict rules defined for defining your bounded context. We need to consider exiting teams and application code organised and working to fulfil process flow.

But yes, we definitely need our Bounded Context to clear define its boundaries and interaction with other internal/external bounded.

**Define Services:**

Based on bounded context defined we can focus on separating services to team responsible to make changes.

Now here we have an interesting development which is widely used in industries now i.e. Microservices

So basically, with microservices architectures we focus on

1. Single Responsibility Principle
2. No tight coupling with other services
3. Independent code for further extension independently

**Bounded Context Mapping:**

Help to find the relation between bounded context and teams responsible for developing it. This will help for effective integration of bounded contexts.

Teams has to decided which pattern is best suited for the purpose. Several patterns used are:

* Partnership
* Shared kernel
* Customer supplier
* Conformist
* Anticorruption Layer
* Open Host Service
* Published Language

Bounded Context Mapping needs to be choose wisely based on the relationship now and evolution of bounded context in future.

Diagram

Description automatically generated

Fig1.2Traditional DDD design

Imagine customer journey service developed as monolith which starts growing so that we are able to scale individual parts of customer service like leads generation, billing, feedback, customer events etc. Tight coupling makes things worse for independent development and testing.

So, (Fig 1.2) we start migrating from monoliths to DDD using series of microservices.

Each bounded context explained before is strong contender of being a separate microservice which need maintainability, extension, scalability, independent development etc.

We further use CQS to bifurcate read and write at method level e.g.:

**CustomerJourneyEventsService**

Customer getCustomerLoginService(CusomerId)

CustomerJourneyEventSet getCustomerJourneyEvents(CustomerId)

void registeredForSession(CusomerId, EventId)

void CreateCustomerJourneyEvent(CustomerId, CustomerEvent)

and with using CQRS (fig 1.3) we start segregating at macro level i.e., bounded context where we separate write and read responsibility altogether as separate service so that they will be individually scalable, eventual consistent, manageable.

With CQRS (fig 1.3) above CustomerJourneyEventsService would be bifurcated as:

CustomerJourneyWriteService

void registeredForSession(CusomerId, EventId)

void CreateCustomerJourneyEvent(CustomerId, CustomerEvent)

CustomerJourneyReadService

Customer getCustomerLoginService(CusomerId)

CustomerJourneyEventSet getCustomerJourneyEvents(CustomerId)

A picture containing graphical user interface

Description automatically generated

Fig. 1.3 Traditional CQRS design

CustomerJourneyWriteService is responsible for command issued in above fig.

Let focus on CustomerJourneyReadService which consist these two methods:

Customer getCustomerLoginService(CusomerId)

CustomerJourneyEventSet getCustomerJourneyEvents(CustomerId)

If we notice getCustomerLoginService and getCustomerJourneyEvents are simply fetching details customer login details and journey events data and everything looks fine.

Now business comes with requirement that they need to outsource these customer journey event with other internal and external bounded contexts  
(**internal bounded context represent internal domain services and external bounded context represent external service**)

So, our first thought is to utilise existing query service, but things go nasty if we mix both pull and push context in single read responsibility which leads us to segregate query service further in query data and push notification service.

Now CustomerJourneyReadService Service would look like:

CustomerJourneyReadService

Customer getCustomerLoginService(CusomerId)

CustomerJourneyEventSet getCustomerJourneyEvents(CustomerId)

CustomerJourneyNotificationService

void pushCustomerJourneyEvent(CustomerEvent)

void publishNotification(CustomerEvent, Subscription)

**INTENT AND APPLICABILITY:**

So, the intent is to bifurcate read responsibilities in push notifications and pull based between bounded context, **so that everybody should not start spinning their own pipeline to get notification or pulling data**

**Graphical user interface, diagram

Description automatically generated**

Figure 2.1 Bounded Context Read Segregation (BCRS)

**Applicability**

The pattern is solely for resolving a particular case of problem which we face during bounded context interactions where pull the data is not efficient, scalable and unified.

So, we need to analyse how our data is going to be processed even if looks simpler like just to expose an endpoint to pull the data from the source of truth, but things can turn around quickly and it becomes un-feasible to pull the data between bounded contexts.

If bounded contexts are expecting any change capture, then we need push notification mechanism to publish the changes.

This communication is applicable for all type of service simple/complex.

**REAL TIME SCENARIO:**

In our case we were injecting the customer related facts into write datastore which eventually copied to read datastore and here we had only single customer journey facts gatherer whose responsibility is to provide the near real time customer journey plus PII information available.

Now the gamification team is looking for real time notification to engage the customers at booth and meeting place. So, the team create a pipeline starts eventually listening the events.

After some days Outside vendors start looking for the customer journey facts data notifications for their platform, so we soon realise that we can’t spin up separate pipelines for notifying each internal/external bounded context.

We need unified solution for the notifications which we missed beforehand. We created notification service which listens to the stream of events needs to be published/notified to the subscribers.

We clearly divided the pull vs push mechanism into its own set of services.

Now notification service is responsible to listen to the events and notify the respective bounded context

Now CustomerJourneyReadService Service would look like:

CustomerJourneyReadService

Customer getCustomerLoginService(CusomerId)

CustomerJourneyEventSet getCustomerJourneyEvents(CustomerId)

CustomerJourneyNotificationService

void pushCustomerJourneyEvent(CustomerEvent)

void publishNotification(CustomerEvent, Subscription)

In our case we use CDC to capture any changes and processing the changes via notification service.

**We are able to add many subscribers configured with different settings on how they want these notifications.**

Now there is single notification listening point for anyone who wants to get notified

**SYSTEM DESIGN:**

In fig 4.1 we are showing the read segregation as Application service responsible to handle all the consumer/clients and Notification service for configurable webhooks/subscribers

**Separation of concerns**: In fig: We are separating the responsibilities of push notification and pull data to their own sub-design system like here we have simple microservices to handle such scenario.   
Although we can have much more complex sub-design on both side pull and push notification.

For handling external webhooks and subscriber we need a simpler unified design to handle both these concerns in a cleaner and configurable way

**e.g.:** Now customer details fetch for login is separated from notifying customer events like registration, offers, email etc.

**Scalability and performance:** Rather handling both push and pull on same service we separated to different services/sub-designs so that they can scalable at their own pace using horizontal scaling or depends on the sub-design.

**Configurability:**

We are able to add many subscribers configured with different settings on how they want these notifications.

e.g.: Subscriber A only wants registration notification but not on any offers given and Subscriber B wants some delay before delivering notification.

Any number of configurable possibilities are there for any business at Notification Service sub-design

**Graphical user interface, application

Description automatically generated**

Fig 4.1 Read Segregation

**Links and Work cited**

Martin Fowler official CQRS design doc:

<https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf>

Martin Fowler application architecture:

<https://martinfowler.com/tags/application%20architecture.html>

Microservices patterns:

https://microservices.io/