Understanding Domain-Driven Design, Command Query Responsibility Segregation & Event Sourcing and their importance in Computer Architecture

Author: Shailee Vora

B Tech Final year in Computer Engineering, VJTI, Mumbai, Maharashtra, India - 2016

Abstract

The layered architecture which consists of UI, Business Logic and Database as the three sections of it, has been proven to have quite a lot of issues with respect to scaling, maintenance and optimization. Here, I have explained the basic idea behind Domain Driven Design (DDD), Command Query Responsibility Segregation (CQRS) and Event Sourcing(ES) and how they can be leveraged for overcoming the shortcomings of the layered architecture.

1. Concepts

This section explains the three concepts this paper presents, namely, DDD, CQRS & ES, and their working with suitable diagrams where required.

a. Domain-Driven Design (DDD)

DDD is about trying to make your software a model of a real-world system or process. This term was coined by Eric Evans in his book by the same name. In using DDD, you are meant to work closely with a *domain expert* who can explain how the real-world system works. For example, if you are building a stock market trading system, your domain expert could be an experienced stock trader. Between yourself and the domain expert, you build a *ubiquitouslanguage*, which is basically a conceptual description of the system. The idea is that you should be able to write down what the system does in a way that the domain expert can read it and verify that it is correct. In our trading example, the ubiquitous language would include the definition of words such as ‘stock’, ‘market’, ‘future’, ‘options’ and so on.

The concepts described by the ubiquitous language will form the basis of your object-oriented design. DDD provides some clear guidance on how your objects should interact, and helps you divide your objects into the following categories:

- *Value objects*, which represent a value that might have sub-parts (for example, an address may have a street name, town/city name, district name, state name, country name and zipcode/pincode).
- *Entities*, which are objects with identity. For example, each Customer object has its own identity, so we know that two customers with the same name are not the same customer.
- *Aggregate roots* are objects that own other objects. This is a complex concept and works on the basis that there are some objects that don't make sense unless they

have an owner. For example, an 'Order Line' object doesn't make sense without an 'Order' to belong to, so we say that the Order is the aggregate root, and Order Line objects can only be manipulated via methods in the Order object.
DDD also recommends several patterns:

- *Repository*, a pattern for persistence (saving and loading your data, typically to/from a database)
- *Factory*, a pattern for object creation
- *Service*, a pattern for creating objects that manipulate your main domain objects without being a part of the domain themselves.When a significant process or transformation in the domain is not a natural responsibility of an *Entity* or *Value Object*, add an operation to the model as standalone interface declared as a *Service*. Define the interface in terms of the language of the model and make sure the operation name is part of the *Ubiquitous Language*.

b. Command Query Responsibility Segregation (CQRS)

Many people think that CQRS is an entire architecture, but they are wrong. CQRS is just a small pattern. This pattern was first introduced by Greg Young and Udi Dahan. They took inspiration from a pattern called Command Query Separation (CQS) which was defined by Bertrand Meyer in his book "Object Oriented Software Construction". The main idea behind CQS is: "A method should either change state of an object, or return a result, but not both. In other words, asking the question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects." (Wikipedia) Because of this we can divide methods into two sets:

- **Commands** - change the state of an object or entire system (sometimes called as modifiers or mutators).
- **Queries** - return results and do not change the state of an object.

In a real situation it is pretty simple to tell which is which. The queries will declare return type, and commands will return void. This pattern is broadly applicable and it makes reasoning about objects easier. On the other hand, CQRS is applicable only on specific problems.Many applications that use mainstream approaches consists of models which are common for read and write side. Having the same model for read and write side leads to a more complex model that could be very difficult to be maintained and optimized.The real strength of these two patterns is that you can separate methods that change state from those that don't. This separation could be very handy in situations when you are dealing with performance and tuning. You can optimize the *read side* of the system separately from the *write side*. The *write side* is known as the domain. The domain contains all the behavior. The read side is specialized for reporting needs.Another benefit of this pattern is in the case of large applications. You can split developers into smaller teams working on different sides of the system (read or write) without knowledge of the other side. For example developers working on read side do not need to understand the domain model.

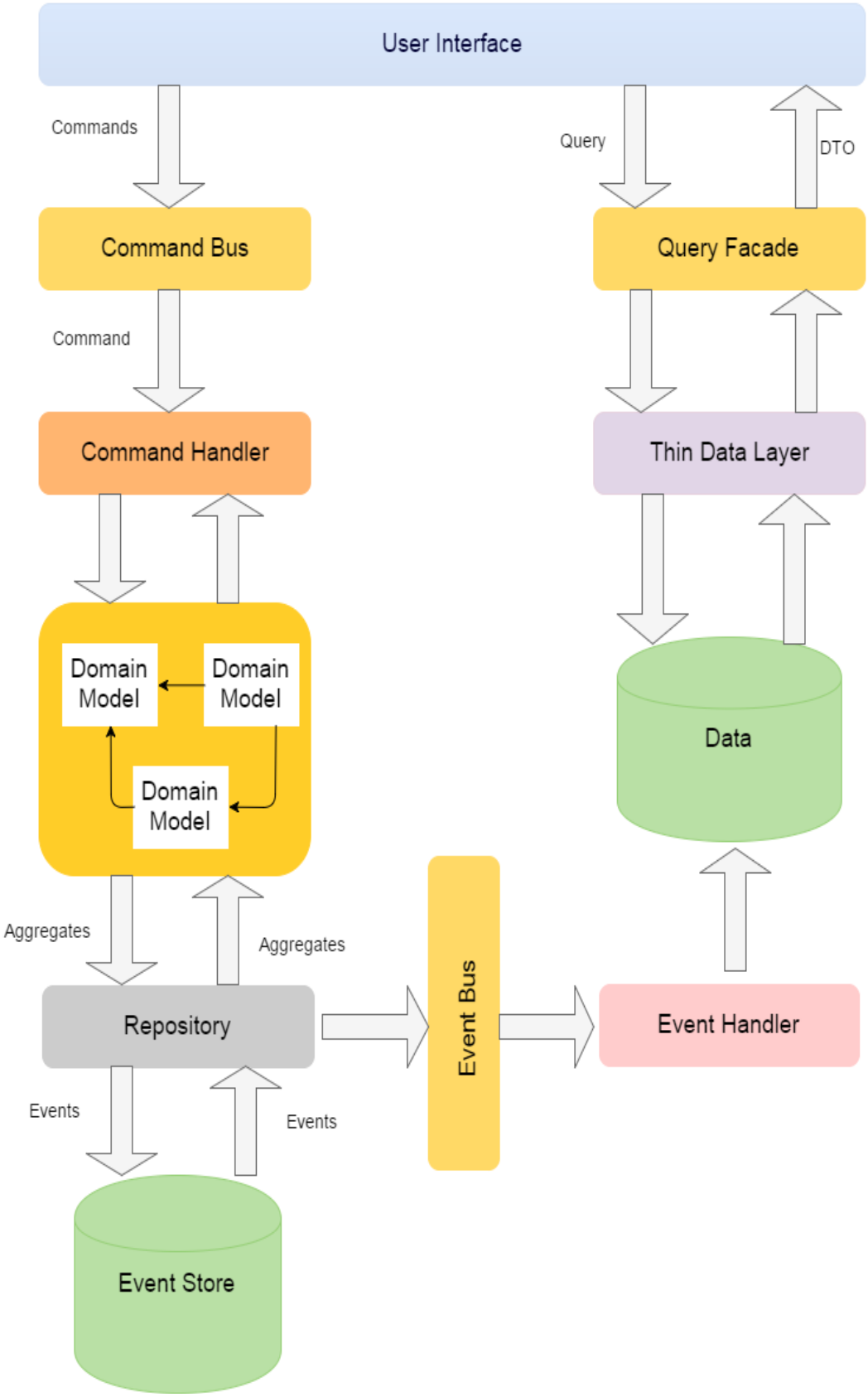

Figure 1: CQRS flow

i. Query side

The queries will only contain the methods for getting data. From an architectural point of view these would be all methods that return Data Transfer Objects(DTOs) that the client consumes to show on the screen. The DTOs are usually projections of domain objects. In some cases it could be a very painful process, especially when complex DTOs are requested.Using CQRS you can avoid these projections. Instead it is possible to introduce a new way of projecting DTOs. You can bypass the domain model and get DTOs directly from the data storage using a read layer. When an application is requesting data, this could be done by a single call to the read layer which returns a single DTO containing all the needed data.

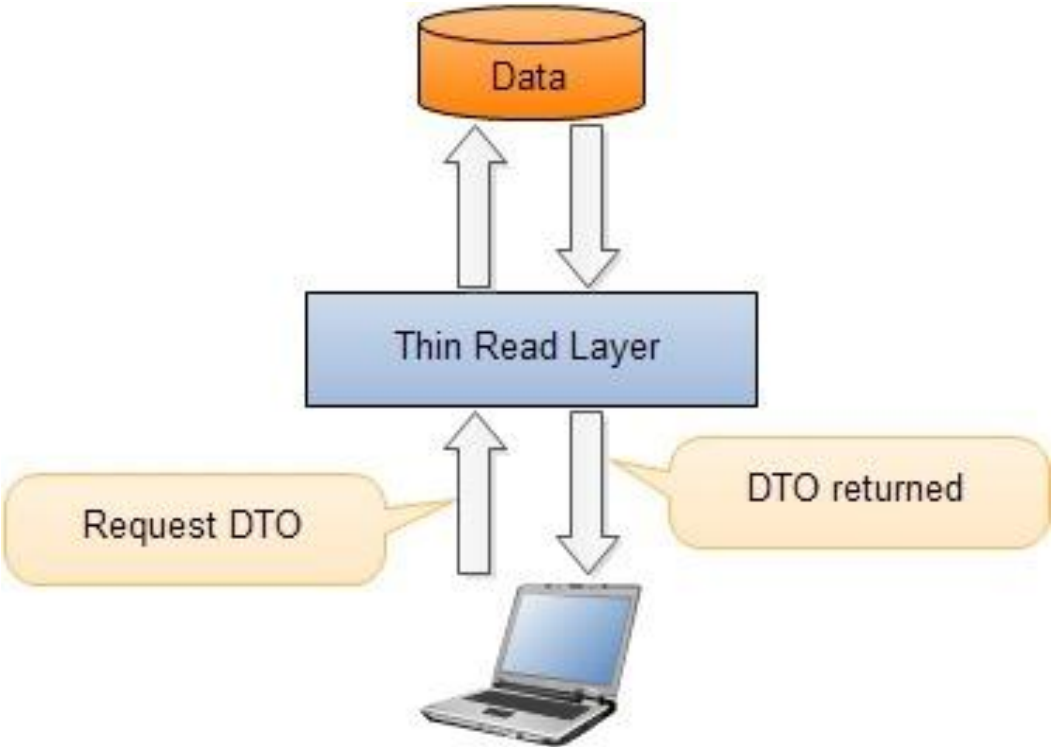

Figure 2: Query side example

The read layer can be directly connected to the database (data model) and it is not a bad idea to use stored procedures for reading data. A direct connection to the data source makes queries very easy to by maintained and optimized. It makes sense to de-normalize data. The reason for this is that data is normally queried many times more than the domain behavior is executed. This de-normalization could increase the performance of the application.

ii. Command side

Since the read side has been separated the domain is only focused on processing of commands. Now the domain objects no longer need to expose the internal state. Repositories have only a few query methods aside from*GetById*.

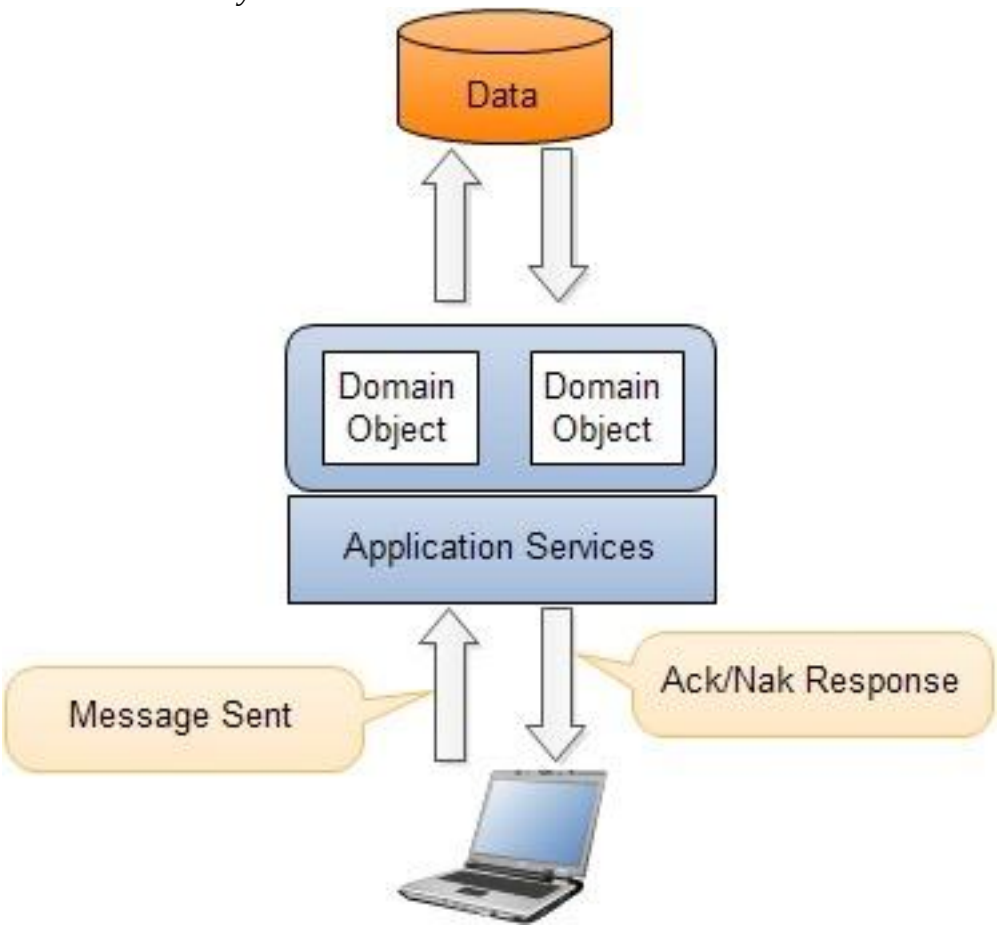

Figure 3: Command side example

Commands are created by the client application and then sent to the domain layer. Commands are messages that instruct a specific entity to perform a certain action. Commands are named like DoSomething (for example, ModifyName, CreateOrder, DeleteOrder ...). Commands are handled by command handlers.

iii. Event-Driven Architecture

Event-driven architecture (EDA), also known as message-driven architecture, is a software architecture pattern promoting the production, detection, consumption of, and reaction to events.

An *event* can be defined as "a significant change in state". For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold". A car dealer's system architecture may treat this state change as an event whose occurrence can be made known to other applications within the architecture. From a formal perspective, what is

produced, published, propagated, detected or consumed is a (typically asynchronous) message called the event notification, and not the event itself, which is the state change that triggered the message emission. Events do not travel, they just occur. However, the term *event* is often used metonymically to denote the notification message itself, which may lead to some confusion. In Figure 1, the Event Bus is used for the propagation of these event notifications.

c. Event Sourcing

i. Why should one use ES?

You can use event-driven architecture to solve the distributed data management challenges in a microservices architecture. However, one major challenge with implementing an event-driven architecture is atomically updating the database and publishing an event. Consider, for example, the Create Order use case. The service that implements this use case must perform two operations: insert a row into the ORDER table and publish an OrderCreated event. It is essential that both operations are done atomically. If only one operation happened because of a failure then the system would behave incorrectly.

The standard way to do it atomically is to use a distributed transaction involving a database and a message broker. However, due to some drawbacks of this approach this is exactly what we do not want to do.

ii. Working of ES

A great solution to this problem is an architectural pattern known as event sourcing. The traditional way to persist an entity is to save its current state. Event sourcing uses a radically different, event-centric approach to persistence. A business object is persisted by storing a sequence of state-changing events. Whenever an object's state changes, a new event is appended to the sequence of events. Since that is one operation it is inherently atomic. An entity's current state is reconstructed by replaying its events.

To see how event sourcing works, consider the *Order* entity of an Online Transaction System. Traditionally, each order maps to a row in an *Order*table along with rows in another table like the *Order_Line_Item* table. But when using event sourcing, the *Order Service* stores an *Order* by persisting its state-changing events: Created, Approved, Shipped, Cancelled. Each event would contain sufficient data to reconstruct the Order's state.

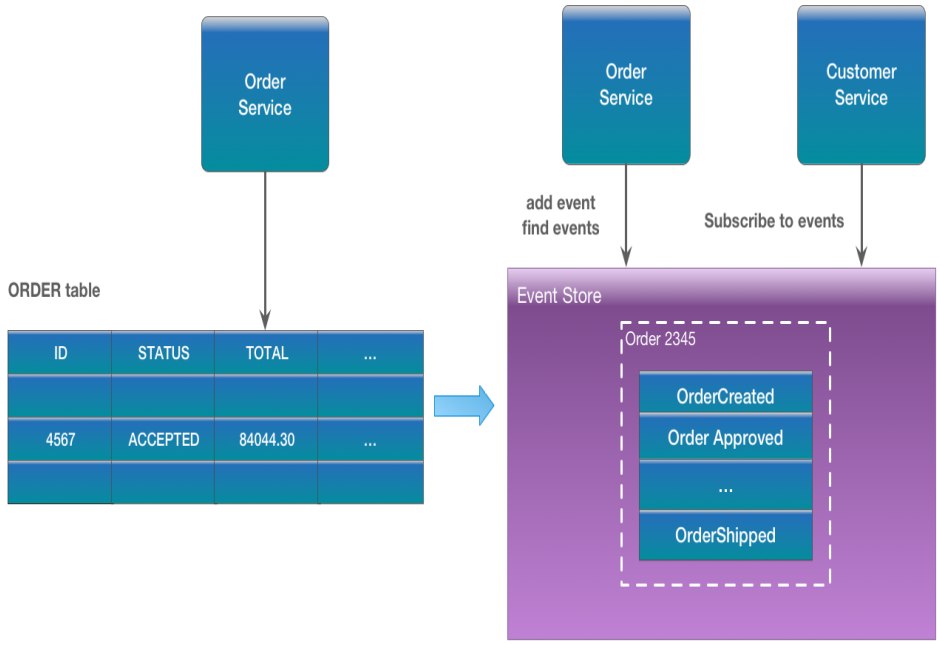

Figure 4: Persistence of state changing events (OrderCreated, OrderApprovedetc)

Events are persisted in an event store. Not only does the event store act as a database of events, it also behaves like a message broker. It provides an API that enables services to subscribe to events. Each event that is persisted in the event store is delivered by the event store to all interested subscribers. The event store is the backbone of an event-driven microservices architecture.

In this architecture, requests to update an entity (either an external HTTP request or an event published by another service) are handled by retrieving the entity's events from the event store, reconstructing the current state of the entity, updating the entity, and saving the new events.

Here is how the *Order Service* handles a request to update an*Order*.

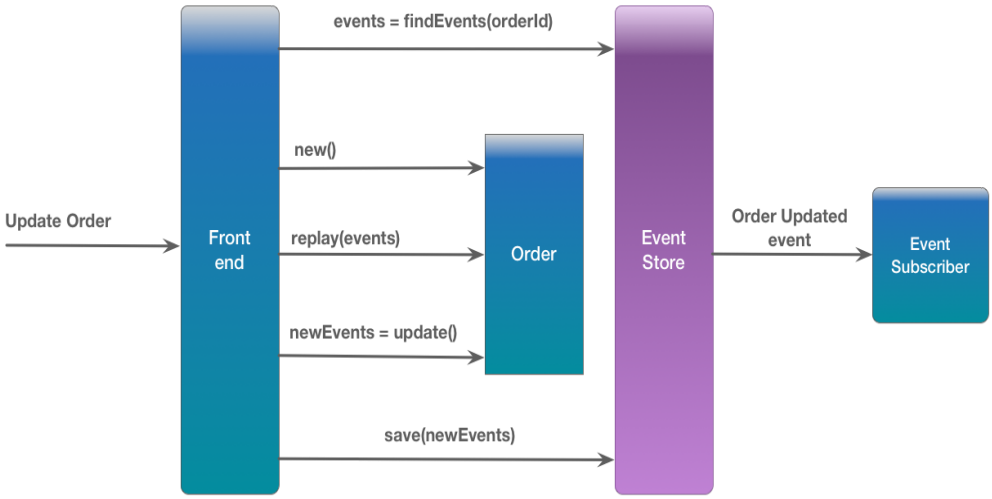

Figure 5: UpdateOrder request handled by Order Service

iii. Other benefits of event sourcing

As you can see, event sourcing addresses a challenge of implementing an event-driven architecture by getting rid of the procedure of updating a database and then using message brokers to publish events. Additional significant benefits of persisting business events include the following:

- **100% accurate audit logging** - Auditing functionality is often added as an afterthought, resulting in an inherent risk of incompleteness. With event sourcing, each state change corresponds to one or more events, providing 100% accurate audit logging.
- **Easy temporal queries** - Because event sourcing maintains the complete history of each business object, implementing temporal queries and reconstructing the historical state of an entity is straightforward.

2. Relationship between DDD, CQRS & ES in a nutshell

a. Domain Driven Design and Command Query Responsibility Segregation

The database connected to the *writeside* of the CQRS system, has the domain model connected to it thereby providing the benefits of DDD which are mentioned in section 1.a to the CQRS system.

b. Command Query Responsibility Segregation and Event Sourcing

The database connected to the *write side* of the CQRS system is persistent storage and stores events as per the rules of ES.

3. Need for DDD, CQRS & ES

a. Traditional layered approach has the database as the single point of failure.
b. In a data driven approach of modeling data, there was no way to carry out logging, auditing, tracing.
c. Each change in the database structure takes exponentially more time than the previous one in the layered approach.
d. The CAP theorem of distributed system, namely, Consistency, Availability and Partitioning, cannot be satisfied always as the size of data grows and the transactions increase in number.
e. Read and Write optimization on the same database can’t be achieved.
f. Data is always stale and cannot be put to good use.

4. Advantages of using DDD, CQRS & ES

a. The database is not a single point of failure when DDD is used with CQRS and ES because events can be recreated owing to the ES mechanism and the lost state of the database can be recreated.
b. Logging, auditing, tracing etc can be carried out if ES is used in the system.

c. When used together, they work with stale data to build a performant distributed system.
d. *Read side* and *write side* databases are different, thus optimizing each separately is possible.
e. Changes to the database structure don't take exponential time because in DDD, to add a new functionality to the software, a new domain is added and connected to the existing domains instead of having to modifying the existing domains.

5. Applications of DDD, CQRS & ES

a. Designing online shopping websites.
b. Designing the software for hospital management systems.
c. Designing software's for banks.

6. References

The links given below, were the reference material used to write this paper.

a. http://www.slideshare.net/MSDEVMTL/how-ddd-cqrs-and-event-sourcing-constitute-the-architecture-of-the-future

b. http://www.codeproject.com/Articles/339725/Domain-Driven-Design-Clear-Your-Concepts-Before-Yo

c. http://squirrel.pl/blog/2015/09/14/achieving-consistency-in-cqrs-with-linear-event-store/

d. http://stackoverflow.com/questions/1222392/can-someone-explain-ddd-in-plain-english-please/1222488#1222488

e. http://www.codeproject.com/Articles/555855/Introduction-to-CQRS

f. http://martinfowler.com/bliki/CQRS.html

g. http://martinfowler.com/eaaDev/EventSourcing.html

h. http://eventuate.io/whyeventsourcing.html

i. http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/

7. About the author

Hello folks! I'm Shailee Vora, a final year B. Tech Computer Engineering student at Veermata Jijabai Technological Institute (VJTI) Mumbai, Maharashtra, India. I

had worked on a project in an MNC as an intern where I had to understand the concepts of DDD, CQRS and ES in order to begin with the project that I had been assigned. The duration of the internship was just a total of 8 weeks which meant I needed to learn all these concepts very quickly, but I figured how difficult it was to look up books and websites to gain considerable proficiency over these concepts in such a short time as each of these concepts is quite vast on its own. Thus, the motivation behind writing this paper was to make a concise reference material for readers interested in this domain of computer architecture. I wrote this paper with the aim to just give a basic insight into these topics from my experience.

****