Responsive Software Architecture Patterns for Workload Variations: A Case-study in a CQRS-based Enterprise Application

Gururaj Maddodi
dept. name of Computing and Information Science
Utrecht University
Utrecht, Netherlands
g.maddodi@uu.nl

Slinger Jansen
dept. name of Computing and Information Science
Utrecht University
Utrecht, Netherlands
slinger.jansen@uu.nl

I. EXTENDED ABSTRACT

In any software system, end-users' workloads vary due to different requirements and business models. In enterprise applications, variation is typically caused by the types of business an organization does, e.g. whether it is a wholesale business with large numbers of orders from the same customer, while supermarkets process single orders for each customer, and each of the orders contains many items. For applications deployed in the cloud, the current approach is to scale the hardware as the usage varies. However, it may be profitable to dynamically adjust the application architecture itself based on the usage of an end-user organization. We term this *responsive architecture*. Responsive architecture can be dynamically adapted to varying workloads as the components of software maps directly to business elements of the business domain the application is serving. This mapping helps to connect the architecture to the application usage.

In the case-study an architectural pattern called Command-query responsibility segregation (CQRS) [1, 2] is used. CQRS is a distributed computing approach, where the system handles requests in the form of commands and queries. CQRS pattern advocates a separation of the request types where the paradigm is that request to view the state of the system should not change its current state. Hence, commands are defined as the actions that create a new state or modify the existing state of the system, whereas queries are the requests that access and present the current state. The separation of command-side and the query-side of the application using CQRS architecture can provide opportunities to optimize the architecture that is used to build the states, the storage mechanisms etc. based on the requirements, hence providing flexibility. CQRS pattern is often used along with Event Sourcing [2]. In event sourcing the creation or modification of new states are recorded as events which are then played back in sequence to obtain the present state of the system.

The command-side in CQRS approach, handles the requests to create new state, hence the framework to build the state is present on the command-side. Also, the mechanism of how the dependencies that exist in the domain elements are handled on command-side in the form of aggregates. Aggregates are concepts from domain-driven design (DDD) [4] that from a functional and business point-of-view consist of entities which can be processed together. Formally, the aggregates can be defined as groups of entities with a defined consistent domain boundary and dependency structure. An example of such a domain boundary could be, a *person* who has an *order* placed for purchase, and *order* contains the *items* that the person wants to purchase. Here, the entities person, order, and order items are all belonging to a single domain boundary of purchasing, and there is hierarchical structure where the person first opens an order and then selects the items that he/she wants to purchase with the order.

Though the aggregates are formed from several entities, separate aggregates can be formed from individual entities. Though they are not in same domain boundary, they can still interact with each other by mechanism of internal events which are not played back while building the state. There are advantages as well as disadvantages to each choice, i.e. a single aggregate containing all the entities or separates aggregates with entities or a combination of them. Firstly, the creating new state is much simpler with single aggregate as everything is in a single boundary and does not need communication as in separated aggregates, but modification of state need the whole aggregate state to be built, while with separate aggregates individual entities can be updated separately. The usage patterns also involve validations which requires attributes to be shared between entities, which in case of single aggregates is simple but complicated in separate aggregate case. Also with the number of attributes increasing, building state in single aggregate become very memory intensive, while in separate aggregate case it being separated can build states only when needed. In this presentation, we present a case-study of impact of workload patterns involving combination of validations, attributes, and entity to entity ratio (e.g. items to order) on architectural choices in terms of resource utilization.

REFERENCES

[1] Jaap Kabbedijk, Slinger Jansen, and Sjaak Brinkkemper. 2012. A case study of the variability consequences of the CQRS pattern in online business software. In Proc. of the 17th European Conference on attern Languages of Programs. 2:1–2:10.

[2] Greg Young. [n. d.]. CQRS and Event Sourcing. Feb. 2010. URl: http://codebetter. com/gregy oung/2010/02/13/cqrs-and-event-sourcing ([n. d.]).

[3] Martin Fowler. 2005. Event sourcing. Online, Dec (2005), 18

[4] Eric Evans; Domain-Driven Design—Tackling Complexity in the Heart of Software; 2003, Addison Wesley, ISBN 0-321-12521-5.