Robert Stockwell

West Texas A&M University

CIDM 6330 Spring 2022

May 4, 2022

# Concepts Overview

Percival and Gregory explore the following significant concepts in the next section: Event-Driven Architecture, Command-Query Responsibility Segregation, and Dependency Injection.

# Event-Driven Architecture

With advancements to the program to a level where events can be captured and command processed, it reaches a level having microservices. Having the microservices is a step forward due to their implementation of the SOLID programming principles. They also can listen for external messages through an External Message Bus. Yet how do the microservices handle processing both internal and external events? The use of Asyhncronous Messaging.

Looking at the applications that have been created and reviewed over the past semester, all of the objects have been thought of as nouns. Examples include Batches from the book, bookmarks from Barky, and Athlete from Shooters\_Log. Each noun has a service that goes with it to perform actions inside the system. This setup creates a tightly coupled organization, where steps need to happen together, and this can quickly fall apart. When this happens, Percival and Gregory state, “We can think of this failure cascade as a kind of temporal coupling: every part of the system has to work at that some time for any part of it to work. As the system gets bigger, there is an exponentially increasing probability that some part is degraded.” (Gregory, 2020) To remove the time factor in the process, Asynchronous Messaging steps in. Asynchronous Messaging allows both parties in the conversation to “start, pause, and resume conversational messaging on their own terms, eliminating the need to wait for a direct live connection.” (TTEC, 2022)

The domain model is modeled against business processes, the verbs, not the nouns in the process when the domain model is developed. These eventually, through evolution, become microservices with their boundaries and events. Two advantages of Asynchronous Messaging are that each service can fail independently and reduce coupling strength. Having the system fail independently allows the other microservices to continue to work and enables the process of degraded behavior to be easier to handle. Decoupling the system allows for easier internal changes to the application without affecting upstream or downstream operations, such as changing algorithms for commission calculations.

# Command-Query Responsibility Segregation

As applications become more complex, reading and writing become much more involved and resource-intensive—especially the writing side when multiple events have access to the same datastore. Keeping data corruption can be a challenge when various events are occurring simultaneously. Typically the Domain model is not written from a read perspective, more from a writing perspective. Changing this can help in the future in writing hardened code and processes. One method of attempting to do this is Command-Query Responsibility Segregation. This attempts to separate the process into halves, the read and the write sides.

Table Read verses write.

|  |  |  |
| --- | --- | --- |
|  | Read Side | Write Side |
| Behavior | Simple Read | Complex business logic |
| Cacheability | Highly cacheable | Uncacheable |
| Consistency | Can be stale | Must be transactionally consistent |

(Gregory, 2020)

Most users will not be affected by a read query being out of sync. Creating an event to read the model and show it to customers on an Adhoc basis can help keep the system’s performance up. Leaving available resources for other functions in the application, such as the much more complex Write side, as noted in Table 1 above. This is a step in breaking down the processes into their core actions, Query for reading, Command for writing, and further moving down the Interface Segregation Principle path.

The more complex Command responsibility starts to take over. Write operations following the domain model and business process closers. It’s nice to display the current status, but tracking what changes is the system’s goal. Double allocating items can cause issues in the business process or consistency errors in the data store. Here there are several options for addressing the issue. Use the existing repository, ORM, or completely “Jump the Shark,” use raw SQL. All have their strengths and weaknesses. Performance considerations should be kept in mind as development occurs. The book calls out the “SELEC N+1” issue specifically. This is a typical performance issue with ORMs and results from the ORM selecting a list of index values and then initiating individual queries to return the attributes for each returned index.

# Dependency Injections

So after reading this chapter several times, my head hurts, and I’m not sure what I’ve read. Bootstrapping with Dependency injections is understandable and desirable, allowing the system to come up live. The bootstrapping allows the explicit requirements to be moved to the abstraction level rather than an implied dependency on implicit detail at the event level.

This chapter will take many readings and a building up of skills for me to understand. I can see how “Dependency injection is regarded with suspicion in the Python world” (Gregory, 2020)