Software Architecture Research Paper

Willie Li

Introduction

This report aims to analyze a variety of software architectures for web applications. Issues with the current software include scalability, testability, data consistency, and ability to develop across a team. The different types of architecture that will be considered are Model-View-Controller, Event Sourcing, and Command Query Responsibility Segregation. Each will be evaluated with respect to the current issues of the web application and its ability to resolve those issues [6].

Model-View-Controller

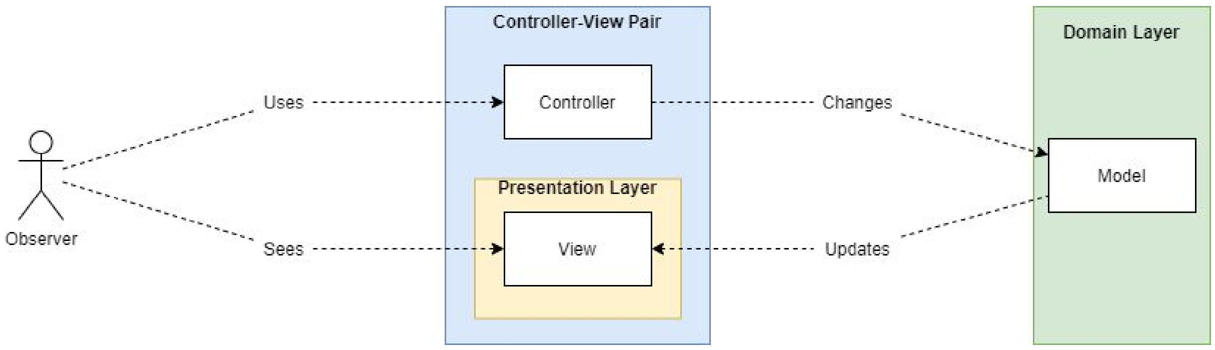

Figure 1: Standard MVC [3] [4]

Model-View-Controller (MVC) is best known for the design decision of separated presentation - layering such that there is segregation between the code that manipulates the presentation and all other code in the program [2]. How MVC breaks down each layer is shown in figure 1. This type of separation between the domain and controller view pair allows for the pairing of multiple presentations with just one underlying model; this is also known as observer synchronization [5]. The model updates all the data and the view is notified given a change in the model. In scenarios where there are multiple views, each can update without needing information from other views as they all refer to the same domain layer. Communication between the controller and the view also happens through the model thus allowing for complete segregation of the three main components [1]. This type of software architecture supports development across a large team, and data consistency.

The extent of separation between MVC components creates independent work streams, and large teams could potentially have one division for each component. As communication between the controller and view happens through the model, it follows that the degree of separation between the controller and model should match the separation between the view and model. Thus, the code for all three components would not have significant interdependencies. Alternatively, different view controller pairings could also be considered independent work streams as views do not require data from other views.

The MVC design also addresses the issues present with data consistency as a result of observer synchronization. Since multiple view controller pairings all depend on the same model, they will all be consistent with respect to what is being displayed.

One area the MVC architecture sometimes struggles with is testing. The separation between components allows for easy unit and component testing but issues can arise with respect to the interaction between the view and model. These bugs are difficult to detect due to the naturally implicit behavior of events triggered by the controller.

In addition to testing, scalability is often another issue for MVC architecture. Of the two types of scalability, vertical and horizontal, MVC appears to struggle with both. The problem presented by vertical scaling is the required down time for hardware. All views depend on a central model, so an upgrade to the model would affect all users. Additionally, server side strain of supporting a large number of views can occur. With respect to horizontal scaling, only views and controllers can be run on different machines but the model must be on a central device. Thus, it's impossible to horizontally scale the view controller pair outside of what is supported by vertical scaling for the model.

Event Sourcing

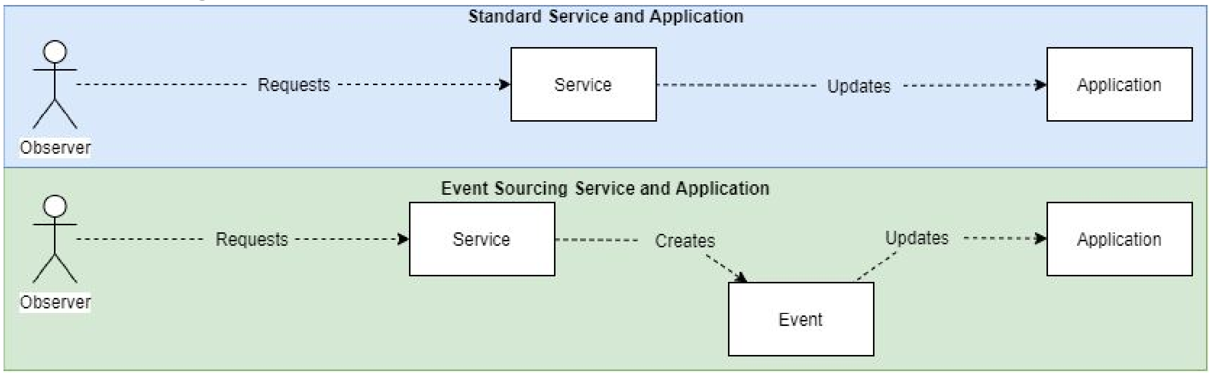

Figure 2: Service and application interaction with and without event sourcing [7] [8]

Event sourcing focuses on translating different application states into a series of events, allowing for easier retracement towards prior states. As shown by the figure 2, there is little difference between the service and application with the addition of event sourcing. However, the event that is created by the service exists for the entire lifetime of the service and application instance. This guarantees that all changes in the application can be traced to event objects, thus allowing for additional functionalities that extend a simple logging service. These include complete rebuild, temporal query, and event replay [7]. Respectively, these functionalities serve to rebuild an application from scratch using the event log, trace the application state with respect to time, and reversing old events to play new future events. These functionalities provide event sourcing with data consistency, testability, and scalability.

Without delving into the additional features, event sourcing promises both data consistency and testability when only considering the event log. As every request has a corresponding event, causes for variances in data can easily be identified, thus guaranteeing data consistency. This feature, alongside event replay, also allows for more efficient integration testing as events can be reversed, then tested using different events, when bugs appear.

When considering software scalability, event sourcing offers excellent horizontal scalability and some vertical scalability. Due to the total rebuild feature enabled by the event log, an application can be replicated to another device using only a default application and event log. Additionally, the same instance on different devices could be updated by passing only the event log. This also allows for no down time during vertical scaling. The old event log can be used to create a duplicate instance when upgrading a machine while the new event log can be used to recreate that instance upon upgrade completion.

Event sourcing does not lend itself well for development across larger teams. As seen in figure 2, event objects depend on both the service and the application. Thus, the team would need knowledge of all component dependencies in order to implement changes to the software. However, event sourcing can be implemented as an additional feature to existing architecture such that it only involves a few components.

Command Query Responsibility Segregation

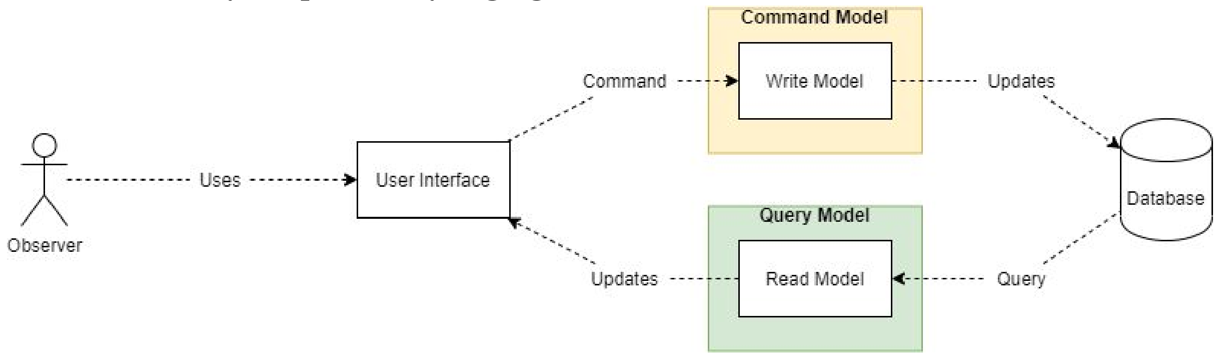

Figure 3: Standard CQRS [9]

Command Query Responsibility Segregation (CQRS) adds a layer of complexity to any software that includes a model and a database by using two different models for reading and writing, respectively, as opposed to just one model for both tasks [9]. Ideally, CQRS would be used in scenarios where the queries and command have enough separation such that two models are required. Thus, CQRS tends only to see usage for specific portions of a larger system, often in tandem with event sourcing [11]. Outside of specific models, CQRS offers a performance boost as the reads and writes can be processed separately [10]. Together, the benefits of CQRS allow for scalability, testability, ability to develop across a large team.

By segregating the read and write commands on two models, CQRS provides the software architecture with increased vertical and horizontal scalability. The two models can run on different machines, and both can run twice as fast given hardware upgrades to both machines. Although this may lead to increased downtime during vertical scaling, the result is a system that can handle a significantly greater number of commands and queries.

Component testing becomes more efficient with CQRS due to a higher degree of segregation for the model. There is minimal effect on integration testing as no communication occurs between models so integration testing only focuses on the communication between the models and database/user interface.

Similarly, having an extra model creates an independent parallel work stream without creating any additional integration points already required by the original model. Although the absolute number of relevant dependencies between the model and other components remains constant with a CQRS implementation, each model team effectively only needs knowledge of half as many dependencies as the command model focuses on writing and the query model focuses on reading.

CQRS may struggle with data consistency depending on the bigger software architecture. Through the introduction of a second model, the data between the models can differ greatly. In instances where the command model is updating the database, the query model may retrieve outdated information if a user request is submitted during this time, resulting in lag.

Software Integration and Conclusion

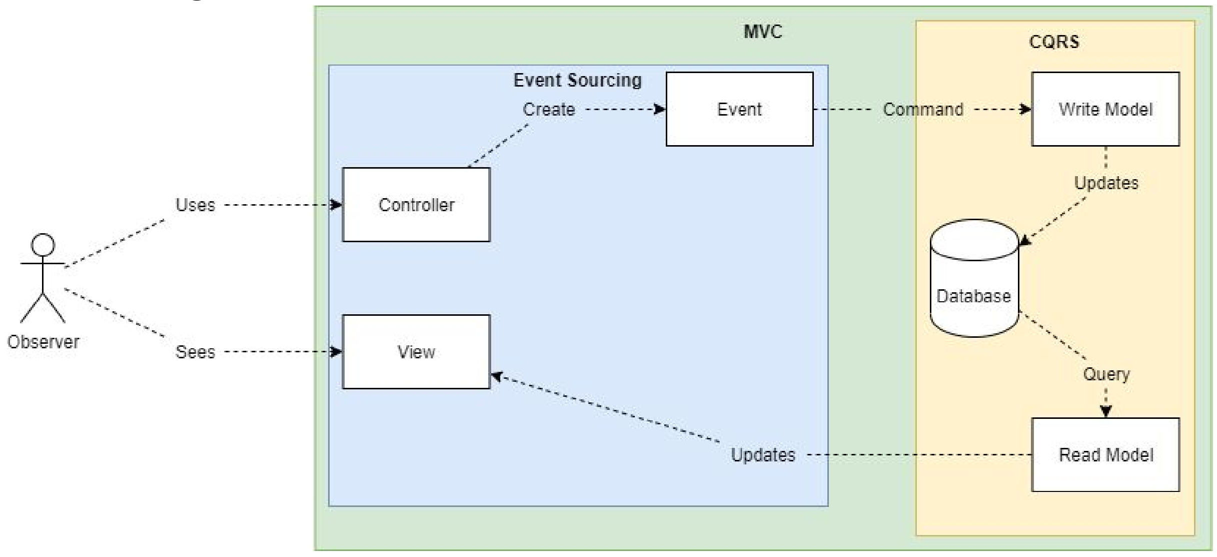

Figure 4: Software architecture solution with MVC, event sourcing, and CQRS

As no single software architecture can address all four issues of scalability, testability, data consistency, and the ability to develop across a large team, a combination of all three architectures is suggested to address the problem as shown in figure 4. As MVC struggles with scalability but addresses all other issues, it has been used as the overarching software architecture. To resolve the scalability issues, event sourcing and CQRS were combined as both architectures excel in scalability. By implementing event sourcing, instances can now be stored and duplicated using just the event objects. Finally, CQRS has been utilized with event sourcing so that an additional model could be used to handle the load of writing and reading a series of event objects. This solution provides the MVC design with horizontal and vertical scalability, which when combined with the design's natural testability, development across a large team, and data consistency, address all the issues mentioned in the scenario.

References

[1] M. Fowler, "GUI Architectures", martinfowler.com, 2006. [Online]. Available: https://www.martinfowler.com/eaaDev/uiArchs.html.

[2] M. Fowler, "Separated Presentation", martinfowler.com, 2006. [Online]. Available: https://www.martinfowler.com/eaaDev/SeparatedPresentation.html.

[3] "MVC Design Pattern", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/mvc-design-pattern.

[4] G. Krasner, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System", ResearchGate, 1988. [Online]. Available: https://www.researchgate.net/publication/239452280_A_Description_of_the_Model-View-Controller_User_Interface_Paradigm_in_the_Smalltalk80_System.

[5] M. Fowler, "Observer Synchronization", martinfowler.com, 2004. [Online]. Available: https://www.martinfowler.com/eaaDev/MediatedSynchronization.html.

[6] O. Aubin "SENG401: Characteristics of Software Architecture Analysis," University of Calgary, Faculty of Electrical and Computer Engineering. [Online]

[7] M. Fowler, "Event Sourcing", martinfowler.com, 2005. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html.

[8] "Event Sourcing Pattern", Microsoft Azure. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

[9] M. Fowler, "CQRS", martinfowler.com, 2011. [Online]. Available: https://martinfowler.com/bliki/CQRS.html.

[10] G. Young, "CQRS Documents by Greg Young", 2011. [Online]. Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

[11] CodeBetter.com. [Online]. Available: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/.