Software Architectures for Web Applications

By: Braeden King

1. Introduction

Many different software architectures have become commonly used in a variety of applications. These three architectures outlined below are architectures that can commonly be found in web-based applications. They utilize different means of decoupling to allow for easy scalability, testability, and ease for large scale development. All three have very different uses when being implemented for a web application. MVC lends itself well to graphic user interfaces, while event sourcing allows for full records of the system states and Pub/Sub allows for decentralized push notifications.

2. MVC

In an MVC architecture, there are three distinct classification of objects. The model objects, view objects, and controller objects. These classifications allow for each of part of the MVC structure to handle a specific task well and not become too complicated or overburdened. Model objects are those that are furthest from the end user, and they handle all manipulation of data, accessing memory or databases, and most importantly responding to events from the controller. View objects are those that handle any graphic user interface. They have the role of taking the data acquired or manipulated by the model through the controller and converting it into a visual display that can be easily interpreted by the end user. This could be anything from converting data in memory into graphs to creating menu systems based on a model object. The final type of object is the controller objects. These have the responsibility of responding to user inputs, as well as being the middleman between the view and the model. This could be a mouse click or typing on the keyboard, and the controller is responsible for making sure that the signal is handled properly, and any updates to the model or view are made. This layered architecture leads to very low coupling since MVC triads [1] are kept separate from one another. This also means that there is very high cohesion within these MVC triads [1] because they are grouped together based on a single responsibility.

The MVC architecture allows for very easy scaling, because of its low coupling. MVC has low coupling in two different places, first the relationships between all the views for a specific controller, and second, the relationship between different MVC triads [1]. If a new view is required for a certain controller then all that must be done is the creation and linking of the view to the controller. This is a simple task and guarantees that the creation of the new view does not break any others. Since all the MVC triads [1] are kept separated, adding more is also very easy. In example, if a new menu system is needed for the web application, then that MVC triad

[1] is completely independent and can be tested by itself. This leads into how easy testing is within an MVC architecture. The low coupling between MVC triads [1] allows for tests to be made for just that MVC group without the need to test the whole system. Data consistency is one of the biggest benefits of an MVC architecture. Model objects have a single responsibly for a specific set of data, which means that the only way that specific data gets manipulated is by that specific model object. Therefore, any controllers trying to access that data for one of their views gets the same data as any other controller, since data manipulation only happens in one place. MVC architecture is very team friendly and allows for a whole team to concurrently work on the model, controller and views.

An MVC architecture makes sense when dealing with a web application. It allows for easily adding new graphic user interfaces with little conflict from other parts of the system as well as ease when testing it. The low coupling of an MVC system also allows for new features to be added without the worry of breaking other parts of the system by adding new MVC triads [1].

3. Event Sourcing

Event sourcing is based around the concept of always having a record for any state that a system has been in instead of just being able to see the final state. To facilitate this, event sourcing must have domain objects and event objects. Domain objects are those that would normally be in the system, like a parking space object in a parking lot system. An event object is one that is time specific and based on the domain object, like an arrival event for that parking space. The domain object must have an ordered list of all event objects that have been created for that domain object. It is also integral to the architecture that changes to domain objects are set off by event objects. This means that the change for a parking space object from available to occupied is made by the creation of the arrival event object and not by the parking space object. These two concepts allow for a couple of unique characteristics. The first is called Complete Rebuild [3]. This means that the state of the system can be destroyed, and it can be recreated by following the event sequences from the event log. The second is called Temporal Query [3]. This means that the state of the system at any point in time can be recreated through rerunning the event log up to that specific point in time.

Scalability is quite easy for an event sourcing architecture, since all events are independent of one another. This allows for new events to be added without them interfering with current events. Event sourcing lends itself to good testability. This is because a tester will always have the sequence of events to get to the current state. These sequences give detailed information on how the system was behaving at various times leading up to the current state of the system, which helps testers diagnose where bugs occurred in the system. Data consistency is handled well in an event sourcing architecture. This is because any changes to the state of the system will always be recorded along with the time at which the change was made. This leads to

always having up to date data by checking the most recent event that took place. Event sourcing lends itself well for developing in a large team. This is due to the very low coupling, which allows for developers to be working on different events without them interfering with each other. When it comes to a web application, event sourcing can be a useful architecture to implement. It allows for many different subsystems in a web application to update simultaneously with the user only pressing a single button. If a user was to press the update button, then the system could create an event that set off the update of the search application, the post application, the topic application and so on, while all being saved in a state log.

4. Publish/Subscribe

Publish/Subscribe or Pub/Sub is an event-driven pattern for sending messages or notifications with very loose coupling. Pub/Sub achieves this by keeping the sender(publisher) and receivers(subscribers) completely separated without either being aware of the other. This is possible by having an intermediary message buffer, known as a topic, be responsible for message distribution. A publisher can create a message and send it to a specific topic. That topic then pushes that message out to all subscribers of that topic. This pattern alleviates the strong coupling that comes with the typical client-server architecture where the sender and receivers of messages must be aware of each other and constantly listening for one another. This allows for subscribers to asynchronously get the most recent messages on that specific topic without having to poll for the update.

Pub/Sub lends itself well for scaling out, since very few connections are needed amongst the components and it can be scaled in many ways. The number of topics, publishers, and subscribers can all be very quickly increased to accommodate higher loads. It is also possible to increase the size of the topic to allows for longer queue of messages or longer messages themselves. It is very easy to test Pub/Sub systems, since each topic will deal with a specific type of message and have a narrow scope. This pattern also allows for reliable data consistency, because each publisher is only creating a single message and all subscribers will receive that message. Pub/Sub allows for large teams to concurrently work on a system, since each topic is completely independent of all others. This design pattern could be implemented in a web application as a notification system. Pub/Sub lends itself well to notification systems because it compiles all notifications about a specific topic and send it to every user that is subscribed to that topic.

5. Conclusion

The three software architectures that were researched all showed usefulness in different aspects of a web application. From MVC being used to decouple the GUI from the logic of the system and having a controller as an intermediary that handled user inputs. To having event-driven design patterns like event sourcing to keep records of the states of the system to help

improve debugging and testing. Or using Pub/Sub as a way of decoupling notifications. Each pattern brings a different use and all could be used in different parts of a web application to further decouple it for improved scalability and testability.

References

[1] Reenskaug, T. and Coplien, J. (2009). *The DCI Architecture: A New Vision of Object-Oriented Programming*. [online] Web.archive.org. Available at: https://web.archive.org/web/20090323032904/https://www.artima.com/articles/dci_vision.html [Accessed 2 Feb. 2020].

[2] Burbeck, S. (1992). *How to use Model-View-Controller (MVC)*. [online] Web.archive.org. Available at: https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html [Accessed 2 Feb. 2020].

[3] Fowler, M. (2005). *Event Sourcing*. [online] martinfowler.com. Available at: https://martinfowler.com/eaaDev/EventSourcing.html [Accessed 2 Feb. 2020].

[4] Michelson, B. (2006). *Event-Driven Architecture Overview*. [online] Elementallinks.com. Available at: http://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf [Accessed 2 Feb. 2020].

[5] Google Cloud. (n.d.). *Pub/Sub: A Google-Scale Messaging Service | Cloud Pub/Sub*. [online] Available at: https://cloud.google.com/pubsub/architecture [Accessed 2 Feb. 2020].

[6] Amazon Web Services, Inc. (n.d.). *What is Pub/Sub Messaging?* [online] Available at: https://aws.amazon.com/pub-sub-messaging/ [Accessed 2 Feb. 2020].