OKONKWO VINCENT IKEM Follow

A journeyman in the field of software craftmanship. A coach. CEO core infrastructure at @ andel...

Jun 26, 2016 · 8 min read

Building Scalable Applications Using Event Sourcing and CQRS

About a year ago, I came across the terms event sourcing and CQRS. I have been fascinated by it ever since. Right now, I am in the middle of building out event-driven microservice infrastructure at Andela using event sourcing. Check out *Building Out an Antifragile Microservice Architecture @ Andela—Design Consideration;* my previous blog post to find out some of the design consideration we had to make.

In this blog post, I will be discussing event sourcing, cqrs, kafka and how we are using it to build scalable data driven infrastructure. I am by no means an expert on the subject. So, if you have experience building an event sourced application, please do me and anyone else reading this a favor by pointing out things I get wrong.

Traditional System

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with the data. For example, in the traditional create, read, update, and delete (CRUD) model, a typical data process will be to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data. This approach has many limitations:

- It requires 2 phase commit(2PC) when working with event driven systems by using a distributed transaction involving the database and the Message Broker. 2PC commit reduce the throughput of transactions. To know more about 2PC, check out this video.
- In a collaborative domain with many concurrent users, data update conflicts are more likely to occur because the update operations take place on a single item of data.
- Unless there is an additional auditing mechanism, which records the details of each operation in a separate log, history is lost.

Event sourcing

Event sourcing achieves atomicity without 2PC by using a radically different, event-centric approach to persisting business entities. Rather than store the current state of an entity, the application stores a sequence of state-changing events. The application reconstructs an entity's current state by replaying the events. Since saving an event is a single operation, it is inherently atomic. The diagram below shows how an event stream is used to reconstruct current state.

In an event sourced application, there is an eventstore that persist events. The Event Store also behaves like the Message Broker. It provides an API that enables services to subscribe to events. The Event Store delivers events to all interested subscribers. The Event Store is the backbone of an event-sourced microservices architecture. For more in depth description of event sourcing, check out introducing event sourcing and this post by Martin Fowler.

Why should I use event sourcing?

Event sourcing provides a lot of benefits besides having a history of events. They include, but not limited to:

- **Audit trail**. Events are immutable and store the full history of the state of the system. As such, they can provide a detailed audit trail of what has taken place within the system.

- **Integration with other subsystems**. Event store can publish events to notify other interested subsystems of changes to the application's state. Again, the event store provides a complete record of all the events that it published to other systems.
- **Time Travel**. By storing events, you have the ability to determine the state of the system at any previous point in time by querying the events associated with a domain object up to that point in time. This enables you to answer historical questions from the business about the system.

Command and Query Responsibility Segregation(CQRS)

CQRS is a simple design pattern for separating concerns. Here, object's methods should be either commands or queries but not both. A query returns data and does not alter the state of the object; a command changes the state of an object but does not return any data. The benefit is that you have a better understanding of what does, and what does not change the state in your system.

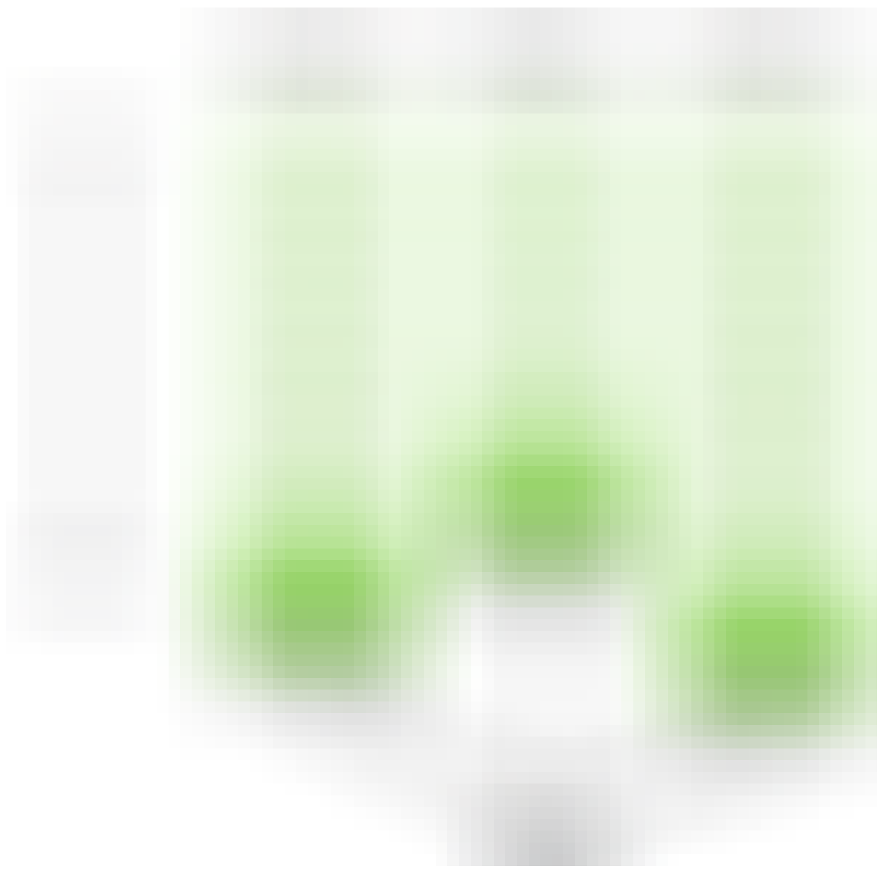
CQRS pattern comes in handy when building an event sourced application. Since event sourcing stores current state as a series of events, it becomes time consuming when you need to retrieve the current state. This is because to get the current state of an entity, we will need to rebuild state within the system by replaying the series of events for that entity. What we can do to alleviate this program is to maintain two separate model(write model— eventstore and read model-a normal database). Once an event is persisted in an event store, a different handler will be in charge of updating the read store. We can read data directly from the read store but not write to it. By using CQRS, we have completely separated read operations(via readstore) from write operations(via eventstore).

Kafka as an EventStore

Kafka is typically a message broker or message queue(comparable to AMQP, JMS, NATS, RabbitMQ). It has two types of clients:

- *producers:* send messages to Kafka
- *consumers:* subscribe to streams of messages in Kafka

What makes Kafka interesting and why it's suitable for use as an eventstore is that it is structured as a log. The way kafka works can be summarized using the diagram below:



Anatomy of

- Producers, write messages to kafka topics eg a topic called *users*
- Every message that is sent to a Kafka topic(eg user's topic) by a producer is appended to the end of a partition. Only write operation is supported by Kafka.
- Topics are split into multiple partitions and each partition is a log(a totally ordered sequence of events)
- Partitions in a topic are independent from each other, so there is no ordering guarantee across partitions
- Each partition is stored on disk and replicated across several machines(based on the replication factor of the partition's topic), so it is durable and can tolerate machine failure without data loss.
- Within each partition, messages have a monotonically increasing *offset* (log position). To consume messages from Kafka, a client reads messages sequentially, starting from a particular offset. That offset is managed by the consumer.

Applying event sourcing with Kafka

At Andela, each kafka topic maps to a bounded context, which in turn maps to a microservice. I will be using our *user-management-service* microservice as a case study. *user-management-service* is responsible for managing all users of our platform. We started with identifying all the types of event that will be published to *users* topic. We were able to come up with the following list of domain events:

- UserCreatedEvent
- UserUpdatedEvent
- UserDeletedEvent
- UserLoggedInEvent
- RoleAssignedToUserEvent
- RoleUnassignedFromUserEvent
- RoleCreatedEvent
- RoleDeletedEvent
- RoleUpdatedEvent

These domain events can be published by any microservice(even by user-management-service) and all interested microservice will consume this event. The diagram below is a simplified flow of a user request.

A user issues a POST request to users endpoint of the API gateway. The gateway in turn makes an RPC(remote procedure call) to *CreateUser* method of *user management service. CreateUser* endpoint performs a set of validations on the user input. If the input is invalid, it will return an error to the API gateway which in turn forwards the error to the user. Once the input has been validated, a *UserCreatedEvent* with the user payload will be published to *users* topic. In our example, *users* topic has 3 partitions, so the event will be published to one of the 3 partition based on some defined logic. We built our system such that all events of a particular user end up been sent to the same partition(this is very important since there is no ordering guarantee of messages sent to different partitions).

Different microservices will be listening to different events sent to different topics. In our case, *user management service, email notification service* and *slack notification service* all subscribed to *UserCreatedEvent.* userCreate handler is invoked when a new *UserCreatedEvent* is received by user management service. This handler actually creates the user in our postgres database. Other microservices, eg *Email Notification service and Slack Notification service* send email to the user and send slack messages to other admin users respectively when they receive the published event.

Notice that in our example, we have 3 instances of *user management service* and each of the instance subscribe to 1 of the 3 available partitions. This way, when a message is published, only one of the 3 replicas receives the message and the message is processed only once. If for example, *users* topic has 10 partitions, the 3 replicas will divide the partitions equally among themselves. In this case, 1 replica will end up consuming from 4 partitions. The maximum number of replicas of *user management service* that we can create is dependent on the number of partitions of each topic been consumed. So it is advisable to over partition from the start. We started with 50 partitions per topic.

Read operations like list all users, fetch a specific user, etc will retrieve data directly from the *read store(postgresDB)*.

Conclusion

Though event sourcing solves a lot of problems inherent in traditional systems, It comes with a lot of challenges which makes adoption very difficult. Some of them include:

- It is a different and unfamiliar style of programming and so there is a learning curve.
- There are very limited resources available out there for building event sourced applications.

In view of the above challenges, I will be writing a series of blog posts where I use event sourcing to build out a twitter clone from grand up. I am thinking of using Nodejs or Golang, but I have not decided yet. Please use the comment section to specify the language you would prefer.

Further Readings

- The Log
- Turning the database inside out
- Why local state is a fundamental primitive in stream processing
- Kafka’s Design
- Using logs to build a solid infrastructure
- The log: The lifeblood of your data pipeline

If you liked this, click the below so other people will see this here on Medium. Also, if you have any question or observation, use the comment section to share your thoughts/questions.