**EventStore Architectural Overview**

The EventStore is designed to store a series of events, known as an event stream, to durable storage. Because of the simplicity of the model of an event stream, it can easily be persisted to durable storage using a large number of storage engines, including relational databases, plain files, NoSQL document databases, or even key-value stores. Herein is a brief summary that describes some of the guiding architectural principles and the underlying philosophy that governs the implementation of the EventStore and dictates the programming interface and calling contracts.

**Event Sourcing**

Event sourcing is a concept whereby a series of events are applied to an object to bring it back to a known state. Event sourcing itself is beyond the scope of this document, but suffice it to say that event sourcing and the concept of storing events as a stream have a number of desirable and interesting properties. Among these are true, verifiable, and correct business audit logs, the ability to reconstruct alternate models from the stream, as well as easier consistency due to the immutability of the persisted events.

**The Model**

The model in a “typical” event store focuses and surrounds two key concepts: The stream and the events or messages for that stream. Ancillary concepts may include metadata such as dates and times as well as snapshot—an optimization for large streams of events.

The model for the EventStore is somewhat different while allowing a great deal of flexibility with regards to underlying storage engine technology without sacrificing any of the valuable properties of event sourcing and storage of events. In fact, this new model is what allows many NoSQL implementations to be leveraged as a durable store for the events.

This new model focuses on two main concepts: streams and commits. At first glance this may not appear to be any different than the typical model. Indeed, it’s very similar. The critical difference lies in the focus on the commit rather than on the event or message. An event is an atomic unit, a “something happened” notification with application and business significance. This fundamental concept cannot be abandoned. But the focus of our new model is on the storage of events, not the events themselves.

By focusing on such a fine-grained unit of atomicity, a typical event store may require multiple roundtrips to the underlying storage engine to fully persist all events generated during a single unit of work. Herein lies the problem. Only handful of storage engines have the capability to handle multiple, disparate writes as a single unit, such as a relational databases and only a handful of NoSQL engines support multiple writes as part of a transaction. This severely limits our ability to choose a technology to better fit our needs.

The new model, by focusing on the concept of a commit, allows the implementation to be much simpler and allows development and operational teams to select almost any storage engine available based upon need and applicability. In essence, a commit is a collection of events resulting from a unit of work. But a commit need not consist of just a series of events. It may also consist of metadata or other attributes that have business or operational significance such as date and time of the commit, IP address and user agent of the user as forwarded by the web server (or client application), *as well as the actual command* that was issued to the aggregate which caused the events to be generated.

Ancillary concepts of the new model include snapshots along with a corresponding snapshotter as well as a dispatcher which is responsible for notifying interested parties of the commits that have just occurred. The dispatcher deserves additional consideration because it also facilitates and enables choice among storage engine solutions. In a typical model, a message is received and events are generated. But a significant problem arises when those events are then persisted durably and simultaneously dispatched to interested parties. This creates what is known as a two-phase commit (2PC) which are often very expensive. Besides the additional latency incurred by having a two-phase commit, a fundamental problem is that only a few select storage engines and queuing systems are capable of participating as cohorts in the commit. This, again, limits our technological choices. By eliminating 2PC, we are able to utilize virtually any queuing infrastructure, including Amazon SQS, Azure Queues, RabbitMQ, ZeroMQ, etc.

The dispatcher liberates us from 2PC by allowing asynchronous dispatching of committed events. In fact, the commit and dispatcher concepts have a harmonious relationship. Unlike a simple set of events, the commit concept carries with it all metadata and context forward to the dispatcher. This allows the dispatcher to append metadata and other pertinent header information when publishing or dispatching the events held by the commit. If the dispatcher did not have all of the corresponding metadata from the commit and simply received a series of events to dispatch, recipients of the dispatched messages could be missing valuable context that might have otherwise enabled business decision-making processes.

**Storage Engine Choices and Implementation Considerations**

Several of the interfaces and programming contracts in the EventStore have been influenced by the need to keep a consistent, non-leaky abstraction of storage and queuing infrastructures. At the same time, it must be recognized and understood that different technological choices and implementations have different strengths and weaknesses. There are two critical examples of this that should be understood.

**The Dispatcher Interface**

The contract of the dispatcher is such that, as a commit is fully dispatched, it notifies the underlying storage of that fact so that it can be marked as dispatched. There is a small possibility during failure scenarios where a set of events may be dispatched but not yet marked as such. This may result in the same set of messages being redispatched. In a message-oriented system, the concept of at-least-once delivery is always something that should be considered and this situation is no different. Message consumers should be made idempotent (potentially by tracking a list of previous messages) to avoid duplicate incoming messages. Again, the trade-off that we gain by avoiding 2PC is the ability to utilize different storage and queuing systems.

Furthermore, in a relational database or some other kind of local, yet transactional system, it would seem logical to number all commits, regardless of stream identifier, with an order-preserving, monotonically increasing number—an auto-increment column of sorts. Then the dispatcher could simply dispatch all events for a given commit, mark the index as dispatched and continue to the next. Indeed, Greg Young’s paper on this very subject recommends this method and it is a solution that works very well. Nonetheless, our purpose here is to understand how the varying capabilities of storage systems, which may or may not be able to create auto-incrementing values, affects the programming contracts of the dispatcher and the EventStore in general.

**Message Idempotency**

One other core area where storage engine differences have affected the programming contracts slightly is that of duplicate commits. A duplicate commit occurs when a message is received twice and processed twice. With idempotent messages this is never a problem. But not all messages are idempotent. When utilizing a relational database as the storage engine for the EventStore, each commit is marked with a commit id based upon the identifier of the incoming message being handled. If an attempt is made to perform a commit resulting from a message that has already been handled, the relational database will throw an exception.

Not all storage engines have or enforce secondary indexes and their uniqueness. In fact, beyond relational databases, very few do. The simplicity of alternate storage engines results in additional implementation burden for the EventStore, but the burden is far from insurmountable. Yet this understanding has influenced the core interfaces utilized to call the EventStore and may be required to facilitate true message idempotency.

When loading a stream of events in order to process an incoming message, the EventStore will query the storage engine and load all the way from the previous snapshot, if any, until the most recent commit. It will then return what is known as a “committed event stream” which contains only the most basic information necessary to facilitate processing of the incoming message. Most incoming messages will be served using this call.

As an optimistic concurrency technique, incoming messages can be marked using the last known version (or revision) of an aggregate (a materialized stream). By leveraging this technique, we can create idempotency when using a NoSQL solution. Our purpose here is to avoid a duplicate commit, but most NoSQL solutions don’t enforce secondary indexes. Hence we have structured the calling API to elegantly address this issue. When the calling application detects that a command has been issued against a previous version of an aggregate, the calling application can invoke the EventStore and request all commits for a particular stream where the revision is at least the version indicated on the command. The EventStore obediently responds by retrieving all commits. On each commit is found a value which uniquely identifies it. This is how we can detect and prevent duplicate commits.

When the application calls and receives a set of all commits since the message version, we are guaranteed that if the message has already been processed, the corresponding commit will have been retrieved from storage. As a result, the EventStore simply identifies and tracks all commits being retrieved. Then, if a write is attempted to the same stream using a commit that has been identified, the EventStore will throw an exception thus maintaining the integrity of the stream.

While portions of the API may not be completely intuitive at first glance, they have been designed to fully encapsulate any underlying storage engine infrastructure preventing the implementation from leaking through the abstraction.

**The Stream Document and Commit Sequence**

In a typical event store, the stream or “aggregate” document or database row, is used to maintain optimistic concurrency. Such is not the case with the EventStore. Instead each commit is given an application-assigned numerically increasing value. In many ways this value is similar to an aggregate version. This allows optimistic concurrency to be maintained regardless of storage implementation choices.

The problem with, **and** significant strength of, most NoSQL solutions is that multiple writes are considered to be separate commits. Because we are creating a commit as a single write, it takes an additional write to update the associated stream document to indicate the correct version of the stream. But this isn’t necessarily a problem because the stream document (or aggregate row) has become a heuristic that enables us to determine if a snapshot should be taken rather than to know the version of the most recent commit. As such, the stream document’s revision value can be updated asynchronously and perhaps even by another process altogether. And if, for some reason, the value isn’t updated upon every single commit, that’s fine too because it doesn’t affect our ability to process messages.

**Snapshots**

Snapshots are performance optimization and can often be ignored altogether except in the systems where latency is mission critical. Snapshots are a materialization of the stream at a certain revision. The snapshot can then be consumed by an aggregate to bring it back to a known state before applying all events which have occurred since the snapshot. Snapshots, if required, should be handled either by an outside process or, on a minimum, a different thread to avoid blocking main message processing.

The one trouble with snapshots is the ability to query for and retrieve the most recent snapshot along with all new commits since the snapshot. Relational databases have no trouble with this, but NoSQL solutions typically have only rudimentary querying capabilities. As such the structure of where snapshots are placed and how they are queried may vary from implementation to implementation. In a relational database, the EventStore places the serialized snapshot alongside the commit, as a separate column on the row to which it applies. This could be done in other ways, such as only having a single snapshot per stream on the Streams table or having a separate Snapshots table altogether.

When using a NoSQL solution and implementing a storage engine adapter for the EventStore, it may be necessary to issue two queries to the storage infrastructure in order to retrieve the most recent snapshot and all events since that snapshot. Fortunately, this may not be a large concern as snapshots can be cached in memory.

Another possibility for NoSQL would be to insert the snapshot between commits in a stream using a semi-alphabetic key. This may not possible or even desirable in a SQL solution where advanced querying exists, but is actually quite practical in NoSQL. To do this, the document key should be “StreamId:CommitSequence” as normal, but this time, we append some kind of non-numeric value such as “Snapshot” to the key. This would have the effect of inlining the snapshot into the result set when a query was performed without the need for an additional query.

**Eventual Consistency and Replication**

One other area where the EventStore has a distinct advantage is that of eventual consistency in the store itself. Because the storage engine is not dictated, an engine such as CouchDB, Cassandra, or Riak could be utilized which support multi-master replication. One potential issue this could have is that of allowing concurrency violations to occur because two or more processes could write to commit #5 for a given stream, for example.

In a fully consistent store, this would be a problem and would actually result in an exception. But when leveraging a store such as CouchDB or a Dynamo clone such as Riak, we can take advantage of the fact that commits are dispatched asynchronously to correct the problem. When the dispatcher goes to dispatch the commit, it could perform a consistent read from its peers and determine if anyone else has written again to that particular commit. If so, it would deterministically pick a commit to dispatch as the winner and it would then “revoke” the other commit by providing it back to the application. The application could then extract from the metadata the underlying command(s) that were issued and re-enqueue them to be processed again. In this way, no command messages are lost, and we maintain a fully consistent event store but still have eventual consistency.

**Conclusion**

While there are other event stores to choose from with varying degrees of community and maturity the EventStore as outlined herein aims to be a first-class choice due to its focus on simplicity and how it enables implementers to utilize technologies with which they are already comfortable and proficient rather than forcing and dictating the use of a relational database.