Event Sourcing

The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself. We can see in the diagram a general view of the idea, the user will cause an event whic will go through some processes changing its properties and and internal state, storing the object every time we apply one of these processes.

The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:
Complete Rebuild: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
Temporal Query: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).

Event Replay: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.
A common example of an application that uses Event Sourcing is a version control system. Such a system uses temporal queries quite often. Subversion uses complete rebuilds whenever you use dump and restore to move stuff between repository files. Enterprise applications that use Event Sourcing are rarer though.

Benefits of event sourcing:

- Increased data quality: as you're using the same events for analytics and for feature teams, the quality of data is improved for machine learning.
- Relatable architecture: the design more closely matches what occurs in the real world.
- Quick reaction: reaction time is really fast (~2 seconds).
- Scalability: easily scalable to thousands of events per second and addition of event consumers, if the base architechture is well designed.
- Service provisioning: strong decoupling of services which reduces time to market new services or changes to old ones. This lets you easily change to a new service and discard the old one.
- Historical record: Usually used for an Inventory or catalog, so all transactions are recorded in order.
- Example of event sourcing: a game of chess, every move is recorded to a log with the information necessary. As they are recorded in such a way, you could easily go backwards to movement 2 or forward to the last move.

So now we are going to do a brief summary about the experiences of a software developer with event sourcing, because I think that the best way to learn is with the example.

From the outset, the first thing he tells us is that event sourcing is difficult, you need some knowledge about, because it has a point.

Seeks to get into the transmission of events and return the data you need.

In practice, it is extremely coupled. The idea of a keeping a central log which multiple services can subscribe and publish is insane. You wouldn't let two separate services reach directly into each other's data storage. If we don’t use the event sourcing approach, we would pump them through a layer of abstraction to avoid breaking every consumer of your service when it needs to change its data – However, with the event sourcing, we pretend this isn't the case.

If you fix it, you are going to have another issue, the opacity. Now, you have like a Observer pattern were control becomes inverted in a way that makes it difficult to know how actually data flows through the system

It is a fairly slow process, it is created from scratch and it will have several implementations since you are learning with the error and thus to be able to finish with a sufficient base where to start.

Then another problem is finding a full development team and disagreements will grow between all, it does not matter their field, since they try to discover the best way to make the system maintainable.

And finally, the important point is that each project is something new, all is heterogeneous so probably, things that you implement in some place, you could not use it in another implementation.

Alexander López, Francisco Manuel Fernández and Marcos Fernández