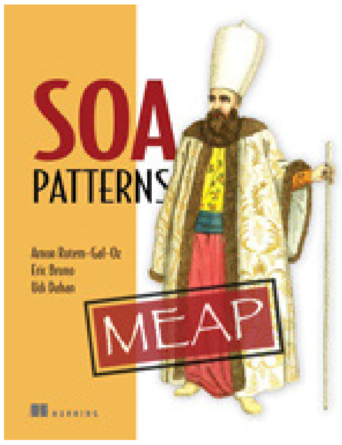

SOA Patterns
By Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan

Integration of services helps to achieve goals that are beyond the goals of a single service, for instance, the collaboration of several services to create a complete business process or creating a report on information from multiple services. This article based on chapter 7 of SOA Patterns takes a look at how to get an integrated view of the data needed for reporting when SOA encourages each service to hold its own data internally.

Aggregated Reporting

Getting an SOA right is hard, not so much because of the technical problems but because it is very hard to understand the business and figure out how to effectively partition it into services. Let's assume we, somehow, managed that and we have our business logic neatly divided into services. We then develop our business logic and business processes and all is almost done. All that is left is to produce a few reports. Well, maybe more than a few—perhaps dozens and dozens of reports. Assuming we did a good job partitioning our business into services, many of these reports will fall within the boundaries of our services. However, at least some of the reports will require data from several services.

The problem

Let's try to visualize the problem. On a project I worked on, we built an analytics platform for call centers. The real-life system had a lot of services but, to illustrate the problem, we'll examine just five of them (see figure 1):

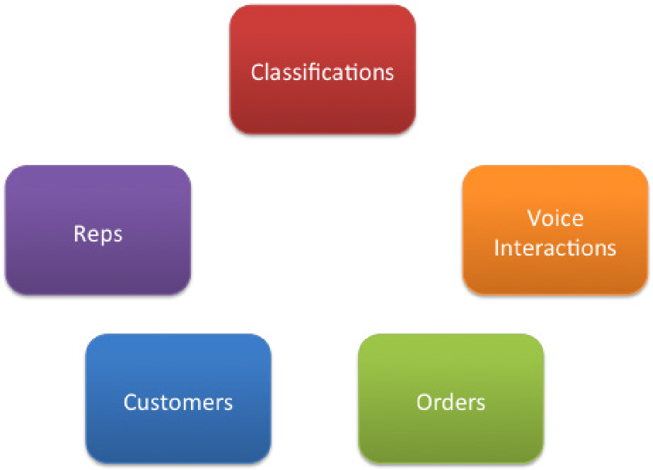

Figure 1 Services in a call center system. Customers make orders and then call a call center to complain/solve problems. The customer's interactions with call center representatives are recorders and analyzed.

- *Voice interactions*—All of the past/current calls a person made to the call center. The service encapsulates all of the metadata.
- *Customer*—Everything we want to know about the customer. Most of it is data imported from other systems like the CRM system and so on. Customer service also does identity resolution (for example, finds the user ID based on a cell number).
- *Reps*—Call center representatives. Data comes here from the operational system that the reps use when they interact with customers.
- *Orders*—Customer's orders.
- *Classifications*—Classify calls according to business-driven criteria, for example, calls by VIP customers that cancelled their service. Categories work both in batch mode and in real time (on incoming calls). It is task-driven—any knowledge it has on customers or interactions is transient; it only stores category definitions.

Now, management wants to understand if there are any correlations between the reps' performance and loss of business by customers, in general, and VIP customers, specifically.

All of the information we need is in the system. The interactions contain all their classifications as well as the customer and rep ids—we can find which representatives handled which customers from there. Customers has the information on which customers are VIP customers and which aren't, plus we can access orders to find out the business generated by each customer

Now all that's left to do is build the SQL query that takes all this data and produces the desired report. But, not so fast—there are two problems with this approach:

- One is the assumptions that all services use an RDBMS as a persistent storage. A few years ago, that might have been a reasonable assumption, but today, with the rise of NoSQL databases, that may not be the case.
- The other, more important, problem with this approach is that it introduces a lot of coupling between the different services. Using a single SQL query to solve the problem, we need to know and understand the internal structures of each service. We constructed services with API level integration to escape this very problem.

The question then is how to generate reports in a way that does not violate SOA principles and produces the reports efficiently:

> How do you get efficient business intelligence and summary reports spanning across the business when the data is scattered and isolated in autonomous services?

One possible solution would be to create the report at the consuming end (the user interface). The consumer would call each service to get its part of the data and then perform all the grouping, crosscuts, and so on. This solution is rarely a good idea. It puts the burden of understanding the data and optimizing the query on the shoulders of each report consumer. If we consider the example above, it is probably obvious that we need to go to customers first to see which of them are VIPs and only then get their total orders, but how do we connect that data to agent's performance? Which will yield a smaller set? And that's just a single report.

Another option is the one we've already mentioned above—go straight to the data and create an SQL query that will go into all the services' databases, join and get all the relevant bits. We've also seen why that's not a good idea.

Maybe the answer is *aggregation services*. This a notion that appeared in the early days of SOA and the idea is that when the granularity is such that the view of a entity is spread over multiple services (i.e. the granularity was wrong), we create a single service that creates a holistic view of that entity. The same idea can be applied toward creating an aggregated entity for the purpose of each report type, and we can copy over some data from all of the relevant services (so we won't have the problem in mentioned in the first option). Well, entity aggregation was a bad idea for its original purpose and it isn't a great idea here as well. For instance, who's the master of the data? Does each aggregate have its own copy of the data? Is the data federated from each service? What do we do when data changes? And, if we can make it work, how many of these will we need to properly provide reporting capabilities?

Well, the answer is that we can have one aggregated, and to make it work, it should follow some specific guidelines. I call this the *aggregated reporting service*.

The solution

Create an aggregated reporting service that gathers immutable copies of data from multiple services for reporting purposes.

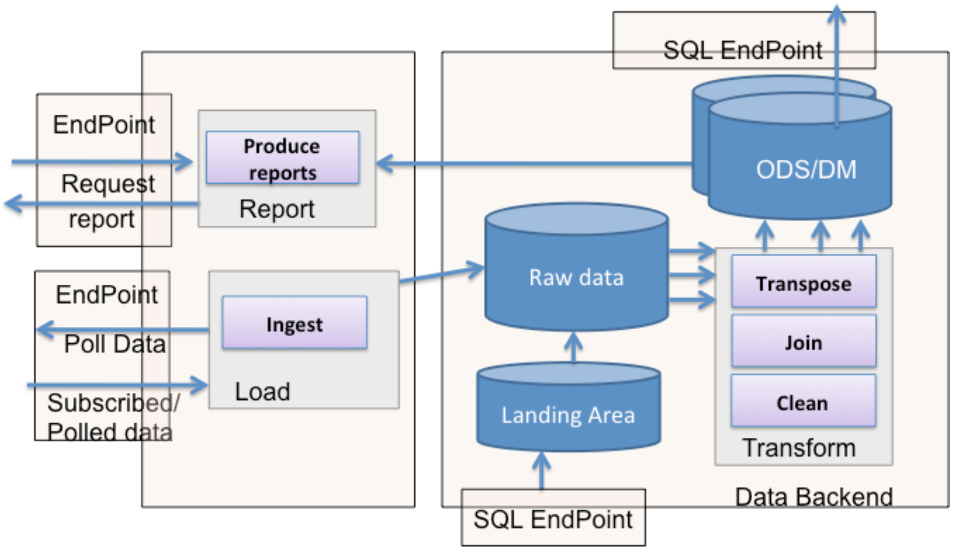

Figure 2 Aggregated reporting pattern

Before we delve into the differences between aggregated reporting and entity aggregation and why aggregated reporting is a good idea, let's first understand exactly what this pattern is. Unsurprisingly, the aggregated reporting is about aggregating data from the services and then providing reporting facilities above the data to make it useful.

The pattern is made of two main components: a service and a data backend.

The service component is the SOA endpoint of the aggregated reporting pattern, where by SOA I mean it utilizes standard SOA technologies like web services or messaging. The service exposes two types of endpoints. The first type is an output endpoint, which provides reports/queries that other services and service consumers can use. The second type of endpoint is an ingestion one. The ingestion endpoint allows collecting data from other service either by subscribing to their events or by allowing other services the means to push data.

The second component, the data backend in figure 2, is the core component of the aggregated reporting pattern. The component has three data stores. A landing area where some data from external interfaces is temporarily stored is used mainly for security purposes and to isolate the SQL ingestion endpoint from the raw data. Another data store is the raw data store. This can either be a temporary storage to coordinate data that arrives asynchronously or a long-term data store that can be the basis for advanced analytics (answering questions we don't know yet we need to ask). The last data store is the reporting data store where data is kept in a reporting-friendly structure, most likely an RDBMS.

The main active functionality of the data backend component is transformation service (which you can think of as an implementation of active service). The transformation service responsibility is to rearrange all the incoming data in a way that would be useful for reporting over it. An aggregated reporting implementation would have extensive transformation components that would sift through the raw data, clean it, aggregate it, and build useful representations of it.

Last are the two endpoints of data backend—an ingestion endpoint for importing data and a reporting endpoint that allows querying data. These endpoints are unique because they use SQL and not technologies usually associated with service orientation. The main reason for that is that standard tools for both importing data and for business intelligence and reporting over data have been built on top of SQL for decades and forcing their hand to other technologies is usually not practical (in terms of effort vs. benefit). We still have to treat these endpoints as bona fide SOA endpoints though—isolate them from internal data structures, provide contracts, and so on.

There are a lot of components that play together here, so let's take an additional look at the pattern from another perspective. Figure 3 below illustrates how data can flow into the aggregated reporting implementation.

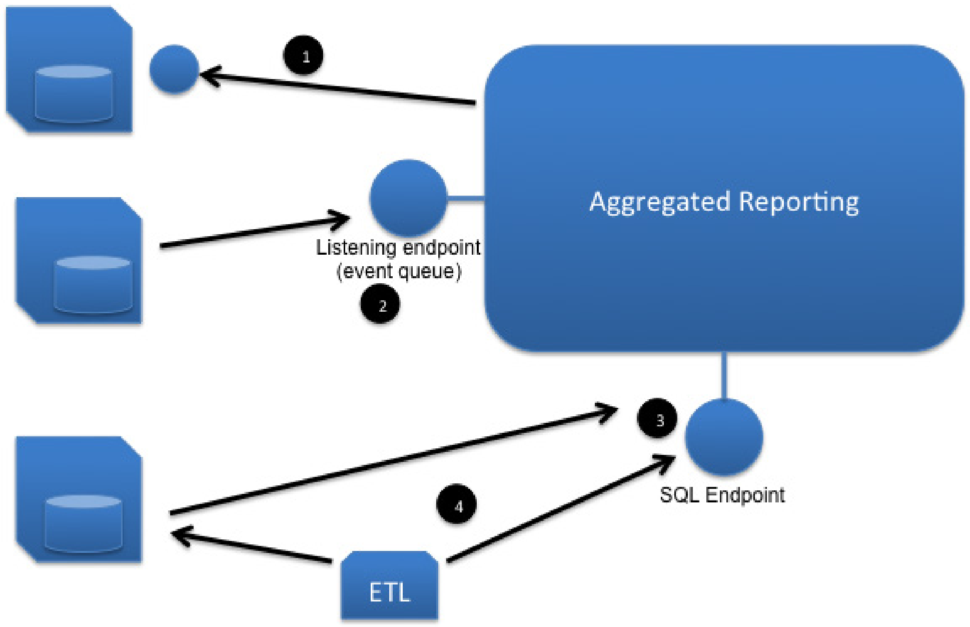

Figure 3 Data sources for an aggregated reporting implementation. The illustration shows four ways to get data into an aggregated reporting implementation: actively going to the data (1), listening to events (2), SQL push by other services (3), or SQL push by ETL tools (4).

Essentially there are four ways:

1. *Actively calling other services*—Here, the aggregated reporting will use other services' contracts to occasionally sample them for new data. This is probably the worst way to go about it because the aggregated reporting implementation has to know about all the other services to be able to do that and the contracts should be expressive enough to export all the needed data.
2. *Passively getting data from services*—There are actually two subtypes here: one where services call the aggregated reporting server with data they wish to expose to reports, for example, by submitting a CSV file with exported data to the aggregated reporting service API. The second variant is to have the aggregated reporting implementation subscribe to events published by other services.
3. *Service SQL Push*—Services export a view of internal data, establish a connection to the aggregated reporting landing database, create their own tables, and save data for reporting there.
4. *ETL SQL push*—Similar to 3, where the responsibility of getting data from services and getting it to the aggregated reporting is on an external tool. This isn't recommended, as mentioned above, because the ETL is likely to violate the services' autonomy to get the data. From the aggregated reporting side, it is still OK because the ETL does not know the internal implementation/representation of data within the service.

Ok, so we have the data in. What happens now? Figure 4 illustrates the process that data goes through once it arrives at the aggregated reporting service. In essence, what happens now is transform-and0load process. Step one is the service side of what we've just explained. It is recommended to use a landing database which is different from the raw storage store when providing security buffer for the SQL endpoint.

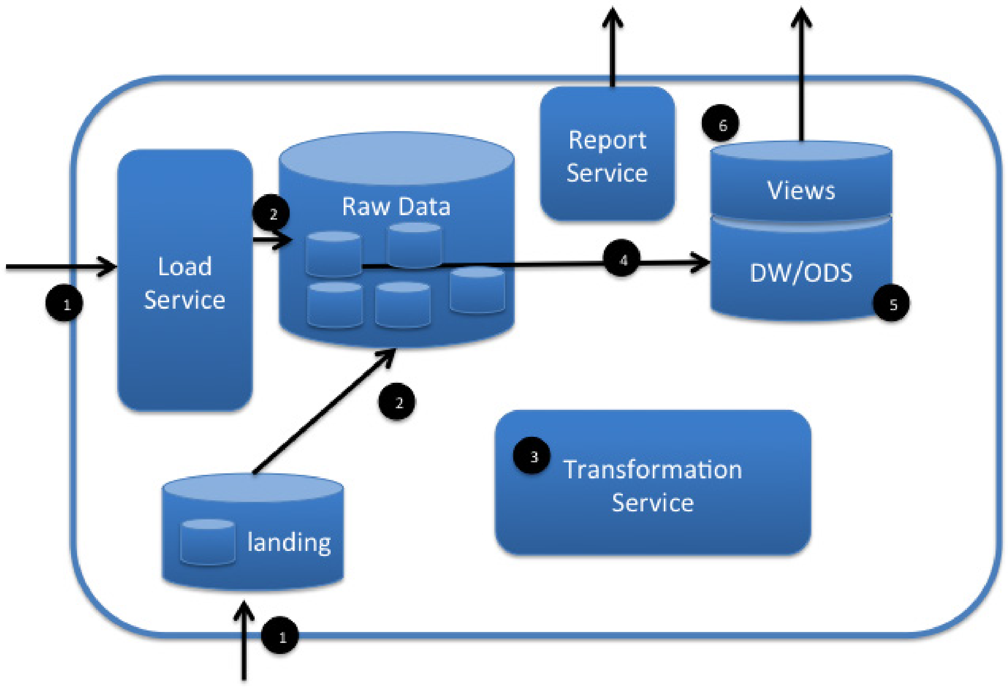

Figure 4 Data processing within the aggregated reporting pattern. We accept the data from external sources (1), save it to a raw data store (2). Process it (3) and prepare it for reporting (4). The data is then saved into a Datamart or and ODS (6) and exposed (6) via a reporting interface and/or views .

Step two is to get the data into the raw data store. There are two options for this. One is a transient store, where the idea is to synchronize different data sources that arrive unsynchronized (each service can send its data independently). The other is to keep the raw data and build a big data store of data over time.

The next step is to process data, cleanse, aggregate, and prepare the data for reporting, for example, create star schemas, build cubes, and so on. Again, as in the previous stage, we can make a choice between longer term and shorter term solutions. If we opt for a shorter term solution, we can set the reporting database as an operational data store (ODS)—a database that is structured like a datamart (denormalized data) but with short retention. The second option is to create a datamart for reporting. It is more common to store the raw data long-term when choosing to use a datamart for reporting.

The last step is to expose the reports. This is done both via service interface for queries and predefined reports and SQL endpoints.

Now that we understand better what aggregated reporting is, there are still a few open questions we need to address.

- How is aggregated reporting SOA friendly?
- How is it better than SQL directly to each service?
- How is it different from entity aggregation?

How is aggregated reporting SOA friendly?

How can the aggregated reporting get data from multiple if not all the services and not violate SOA principles? Well, what makes aggregated reporting a service is that the data it holds is immutable and the aggregated reporting is not the owner of changes in the data. It holds a representation of unchanging data for use in the service it provides (reporting). It is recommended, in this regard, that data kept by the aggregated reporting service would be idempotent (versioned) so that the relations it expresses will always be true (for the versions involved). In any event, the source of “truth” is the original services whose data is mirrored. On the structural level, the aggregated reporting service is SOA compatible because it externalizes its capabilities via well-defined interfaces. The incoming SQL endpoint needs to be configurable via the regular service API—a service should contact the service API to request an allocation of space. It will then be guaranteed connection credentials to its own landing zone. The implementation specifics can be different, but the idea that the interaction with the incoming SQL endpoint is to be controlled via a contract should hold. As for the output SQL endpoint, here, the separation of internal data structures and notion of contract should be implemented by a layer of views. The views represent the external agreement and the internal implementation don’t have to match and can vary from the external one.

How does aggregated reporting defer from direct access to each service internal database?

First, as already mentioned, the internal structure of a service might not be an RDBMS. Second, exposing SQL at each and every service increases the risk this will not be done right, either by exposing internal data structures, mangling up security, and so on. The real benefit, however, is that the aggregated reporting internal structure is built for reporting and as such will, most likely, provide much better performance than accessing even a single service directly because the service’s internal data stores are transaction (OLTP) oriented and not reporting oriented. That is even truer for reports that need to be cross services.

How is aggregated reporting different from entity aggregation?

I included this question because it sounds they are similar as both have the word aggregation in their name. However the similarity ends there. Aggregated reporting keeps data ownership at the different services, is not focused on a single entity, is built on immutable data, and is geared only towards reporting.

What are the drawbacks of using aggregated reporting?

I personally believe aggregated reporting is the best way to handle reporting in service oriented architecture. However, like every design decision, it does come with its own tradeoffs. The main tradeoffs here are the relative complexity of the solution (vs. reporting going to each service), which translates to higher time to market, increased latency in terms of freshness of data (data has to be processed before it is available), and increased storage costs form duplication of data. The benefits, as we’ve already mentioned are high performance of reports, cohesive view of the data, separation of responsibilities, and keeping SOA’s flexibility benefits. An additional non-SOA benefit of the aggregated reporting pattern is the promotion of concepts such as command-query responsibility separation (CQRS) and master data management.

Technology mapping

We’ve explored the structure of the aggregated reporting pattern and saw that it is has a lot of functionality, which means that there are plenty of ways to implement it and plenty of technologies that can play the various parts. As usual, the point of covering technology mapping is not to provide an exhaustive list of implementation options but rather to provide a taste of what’s possible. One of the interesting options, which has grown in popularity in recent years is to implement aggregated reporting as a big data store.

Recent systems I worked on used Hadoop and conventional datamart together as the aggregated reporting implementation. The Hadoop system was used as a datawarehouse with the long term, never-delete storage of historic data coming from all the services. The data from each of the services was saved in almost raw form in Hadoop’s distributed file system (HDFS) as it arrived and processed at later intervals to provide data useful for reporting. An ETL process took recent (last few months) of that data and exported it to a star schema in a conventional datamart (Oracle).

An interesting scenario in this regard is the case when the data exported to the datamart is summary data and not the detailed data, for instance, if we have the reps’ monthly performance data in the datamart and their call-by-call performance data stays in the datawarehouse.

Figure 5 illustrates how a drill-through from datamart data to Hadoop data can occur. The first step occurs when the summary data is calculated (step 1). TheMapReduce job that calculates the summary and exports the data to the datamart also saves the origin of each calculation in HBase. (HBase is a Hadoop-based NoSql solution that supports high-throughput random read/write.)

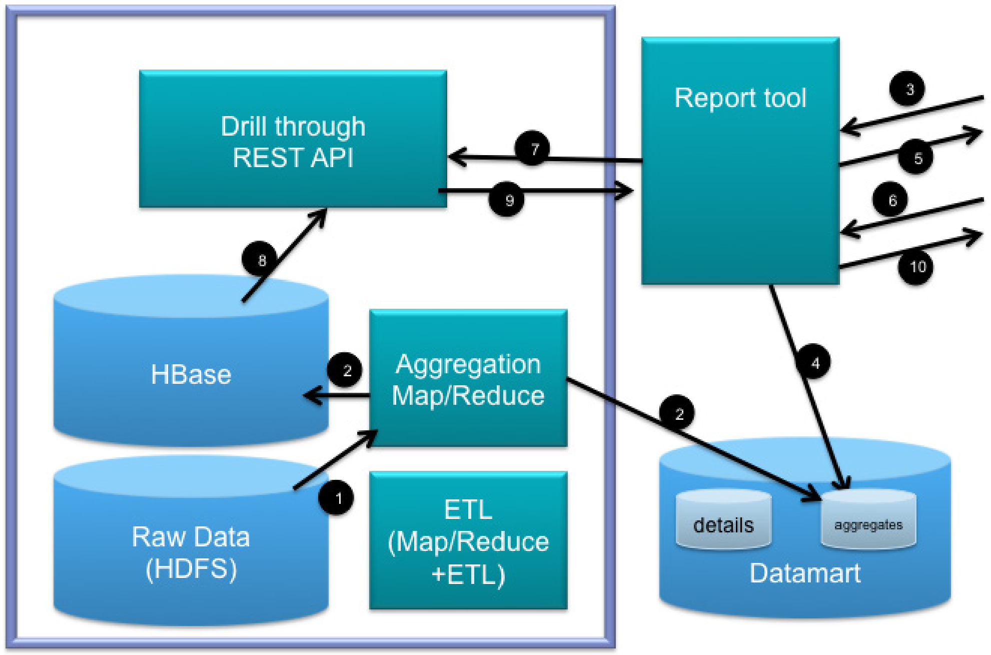

Figure 5 Example of a drill-through from summary data in a data mart to a Hadoop-based data warehouse. When calculating summary data (1), the job writes the summary into the data mart and the source data to HBase (2). When a report is processed, it runs against the datamart, (3, 4, and 5). When a user requests to see how the summary data was calculated (6), the reporting tool makes a REST call (7) to a service, which gets the details from HBase (8) and provides it (9) to the reporting tool, which in turn notifies the user (10).

When a user runs a report on the aggregated data (steps 3, 4, and 5 in the diagram) she can see the data as it presented from the datamart (our sql endpoint in the aggregated reporting pattern) when the user wants to see the drill-through into each call that produced the score (step 5). The reporting tool then makes a call to the service endpoint of the aggregated reporting, providing the key of the summary data (step 7). The service then accesses the list of sources generated when the summary data was generated (steps 8 and 9) and the data is presented to the end user (step 10).

Again, this is just one possible implementation. For instance, in another, smaller project we just used an operational data store to hold the latest data in a start schema without retaining long-term historic view of the data. The details change but the architectural principles stay the same.

The last thing to discuss about the aggregated reporting pattern is quality attributes where things are a little different from other patterns.

Quality attributes

The aggregated reporting pattern is probably the only architectural pattern whose main drivers are functional requirements rather than architectural qualities. The reason the aggregated reporting pattern is still architectural is that its implications are solution-/system-wide and not local. As mentioned above, the aggregated reporting pattern provides a functional solution that still retains SOAs architectural benefits and that's its strength.

Note that the aggregated reporting pattern does promote desirable quality attributes like flexibility and maintainability but it isn't driven by them – its motivation, as mentioned above is functional and not non-functional.

Summary

Integration patterns enable services to work together and become a system rather than a bunch of services or a knot of unmaintainable mess. We covered aggregated reporting, which provides a SOA-friendly way to solve the reporting conundrum.

Last updated: March 26, 2012