SOA PATTERNS

Arnon Rotem-Gal-Oz

SAMPLE CHAPTER

MANNING

SOA Patterns

by Arnon Rotem-Gal-Oz

Chapter 10

brief contents

10 SOA vs. the world

In this chapter

- Constraints of REST
- SOA and the cloud
- SOA and big data

In this part of the book, we've looked at antipatterns, discussing some of the things that can go wrong, and we've looked at a case study, exploring how different patterns can interact with and complement each other. This chapter takes a look at the impact of other architectural styles and trends on SOA. We're going to cover:

- *REST*—What is the relationship between REST and SOA? Are they friends? Foes? Can they work together?
- *Cloud*—Is SOA a good fit for cloud-based deployments? How does the cloud affect SOA?
- *Big data*—NoSQL is starting to mature, with offerings from the big vendors both in the advanced analytics front (IBM and EMC offer distributions of Hadoop; Microsoft, Oracle, and others provide Hadoop integration) as well as solutions for big data in real time (such as IBM InfoSphere Streams and SAP HANA). How does SOA fit in?

Let's start by looking at the REST architectural style, which many see as an alternative to SOA.

10.1 REST vs. SOA

In recent years, the REST architectural style has become very popular, with a lot of companies building RESTful APIs (such as Twitter and Facebook) and a lot of other companies building value-added services, called mashups, by using these APIs.

Wikipedia defines mashups as:

> In *Web development*, a *mashup* is a *Web page* or application that uses and combines data, presentation or functionality from two or more sources to create new services. The term implies easy, fast integration, frequently using *open APIs* and data sources to produce enriched results that were not necessarily the original reason for producing the raw source data.
>
> The main characteristics of the mashup are combination, visualization, and aggregation. It is important to make existing data more useful, moreover for personal and professional use.[1]

This makes mashups sound a little like SOA, so to help clarify things I'll explain the differences between REST and SOA and what a RESTful SOA is. But first, let's look at what exactly REST is.

10.1.1 What is REST anyway?

REST is short for REpresentational State Transfer, and it's an architectural style defined by Roy T. Fielding in 2000 to describe the architectural style of the web. REST's basic component is the *resource*, which is addressable at an endpoint called a *URI*. Figure 10.1 illustrates the constraints the REST style defines.

Let's look at the constraints one by one:

- *Layered system*—The layered architectural style defines a hierarchy of components (layers) so that each layer can only know one level down. This promotes simplicity and the ability to enhance capabilities by adding middle layers (such as a firewall for added security).

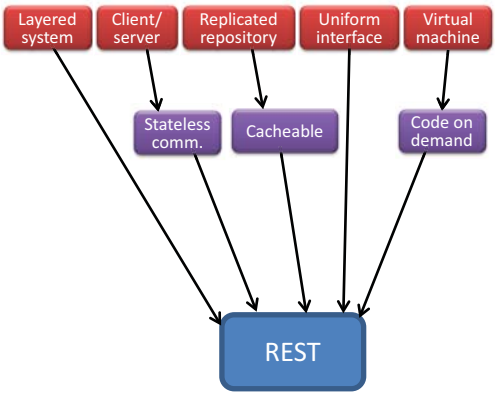

Figure 10.1 The REST architectural style is derived from five base architectural styles: layered system, client/server, replicated repository, uniform interface, and virtual machine

[1] Wikipedia mashup definition: http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid).

- *Client/server*—The client/server architectural style introduces a separation of concerns between consumers and the providers.
- *Stateless communications*—This constraint means that each request made from the client to the server should have enough context (state) for the server to figure out what to do with it. This is why there are cookies that carry the session state from browser to server.
- *Replicated repository*—The idea behind this constraint is that it is OK to have more than one process provide a particular service in order to achieve scalability and availability of data.
- *Cacheable*—The cacheable constraint means that messages can specify whether it is OK to cache them and for how long. This constraint is an application of the replicated repository constraint to the message level, and it helps save on server round-trips, improves performance, and decreases server loads.
- *Uniform interface*—Probably the most distinct characteristic of REST is the use of a limited vocabulary. HTTP, the most prevalent REST implementation, offers just eight methods (`GET`, `POST`, `PUT`, `DELETE`, and the lesser known `OPTIONS`, `HEAD`, `TRACE`, and `CONNECT`). The uniform interface makes it relatively easy to integrate with RESTful services, and it also has a lot of impact on how you model RESTful services (as compared to non-RESTful services).
- *Virtual machine*—Virtual machine or interpreter is the ability to run scripted code. This is a prerequisite to the next constraint, "code on demand."
- *Code on demand*—This is an optional constraint that allows you to download code to the client for execution (such as JavaScript that runs in a browser). Code on demand makes integration easier, because clients can get code to handle the data they need instead of having to write code to handle the data themselves.

Another important aspect of REST is the use of Hypermedia as the Engine of Application State (HATEOAS). HATEOAS means that replies from a REST service should provide links (URIs) to the available options, which are based on the server's state, for moving forward from the current point. If a request to place an order was made, the reply can contain a URI for tracking the order, a URI for canceling the order, a URI for paying for it, and so on. HATEOAS is an outcome of using a uniform interface, and provides a map of the way to fulfill business goals when working with REST.

That's a view of REST from 50,000 feet, but even so, we can see some similarities to and differences from SOA.

10.1.2 How REST and SOA are different

REST shares a couple of constraints and components with SOA. Client/server and the notion of a layered system are basic building blocks of SOA, as they are for REST. On the other hand, constraints like uniform interface and virtual machine are very foreign to SOA.

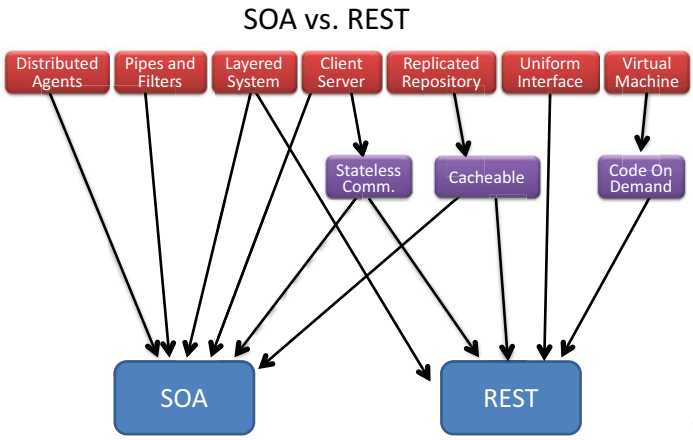

Figure 10.2 A comparison of REST and SOA architectural constraints

You can see the whole picture in figure 10.2, which illustrates SOA's influences as compared to REST's.

In addition to the layered system and client/server constraints, you can see two other REST constraints that are optional in SOA: stateless communications and cacheable. One of the optional constraints in SOA is the cacheable style.

In terms of the latter, we talked in chapter 5 about message exchange patterns and the benefits of sending immutable (versioned) data in messages. Immutable messages are SOA's way to specify cacheable messages; explicitly specifying cacheabilty, like in REST, is also an option.

The Service Instance pattern from chapter 3 is supportive of the replicated repository constraint. Similarly, while stateless communication is not a must in SOA, it is highly recommended (see the discussion on document-centric messaging in the Request-Reply pattern in chapter 5).

SOA's benefits over REST include governance and planned reuse as well as high security standards and a wealth of supporting components and message patterns (such as publish/subscribe). REST's advantages (especially REST over HTTP) include the ubiquity of the browser and the serendipity of reuse.[2]

The virtual machine constraint is very foreign to SOA, and fortunately it and its derived constraint (code on demand) are optional for REST. This means you can combine REST and SOA to enhance SOAs reuse with REST reuse serendipity.

10.1.3 RESTful SOA

I find that RESTful SOA is beneficial when you want to have a dual API. In most other cases, it's usually better to choose either SOA or REST (based on your specific needs) and stick with it.

---

[2] Steve Vinoski, "Serendipitous Reuse," *IEEE Internet Computing*, volume 12, issue 1 (January 2008), 84–87, http://steve.vinoski.net/pdf/IEEE-Serendipitous_Reuse.pdf.

How can you enrich SOA with REST? There are basically two approaches:

- Build a RESTful service and extend it to be an SOA one
- Take an SOA service and extend it to be a RESTful one

I recommend the latter approach, because SOA offers more flexible ways to connect services and has better tooling support. Also, it's likely that in enterprise environments SOA-related APIs will be more prevalent. That said, you'll often want to add REST to allow third-party integration and to allow mobile clients to interact and consume services directly (somewhat like the Composite Front End pattern in chapter 6).

> **NOTE** The Edge Component pattern (discussed in chapter 2) is a good approach for adding a REST API on top of, or in addition to, an existing SOA API. You can even use technologies like Apache Camel, which enable flexible routing from external interfaces to internal ones.

The REST and SOA APIs will look radically different. REST comes with a hierarchical noun-oriented API, and SOA has a shallow verb-oriented API (both for event-oriented and web service-oriented APIs). Nevertheless, I find that mapping between the two is more straightforward than you might expect.

Mapping REST to SOA

Mapping REST to SOA is not an automatic task. But while you will have to put some thought into it, it's more than doable. The following list contains a few tips or things to remember when building a REST-to-SOA mapping:

- Different resources can map to a single service. If you have Order and Product resources, the Order resource may have a `GET /orders/<order id>` URI to see order details, and a `GET /products/orders/` URI to see the different orders a product participates in. Both might be mapped to an Order service with two messages in its contract, such as `ListOrderDetails` and `GetProductOrders`.
- Different REST URIs can point to the same message in a service. Both `POST /orders/`, which creates an order where the server allocated the key, and `PUT /orders/<order id>`, which creates an order where the client sets the order, ID can map to the same `CreateOrder` message, which accepts an XML message that may or may not have an order ID.
- As REST is new to most SOA practitioners, it is important to avoid common REST mistakes, like forgetting all the HTTP verbs and building a GETful architecture (where only the `GET` method is used), neglecting to use hypermedia, the error of using verbs as URIs (such as `/createOrder/`), and so on.
- If you have a proper REST API that utilizes HATEOAS and properly implements the `OPTIONS` verb to allow checking for next steps, a contract for the REST API isn't needed. Remember that the SOA API already has a more formal contract (event list or WSDL) and that the REST API is supplementary.

SOA and REST can be made to work together, and this combination can be beneficial, especially if you plan to expose an API for consumption by UI applications directly, and not limit it to being consumed by other applications. If you build your services properly and employ REST practices, using stateless communication and making results cacheable, you can add REST as an additional API (or as the only API for new services) and still get SOA's benefits.

That's enough about REST. Let's see how SOA matches up with another hot trend—the cloud.

10.2 SOA and the cloud

Cloud computing is an important IT trend, taking virtualization to the next level by using a large pool of virtualized hardware to provide utility computing capabilities. It provides an electricity-like model, where computational resources are available on demand (usually with pay-as-you-go billing) and with the ability to elastically grow and shrink your resource use as needed.

We'll take a look at how this relatively new playground affects SOA, but let's first try to make sense of the different cloud-related terms out there.

10.2.1 The cloud terminology soup

Cloud computing sounds a lot like many other virtualization and hosting solutions that have come around before. But while cloud technologies share concepts with previous solutions, there are several characteristics that differentiate cloud computing.

The U.S. National Institute of Standards and Technology published a formal definition of cloud computing (see the further reading section) in which it defined five essential characteristics:

- *On-demand self service*—The ability for cloud users to add capabilities (such as virtual machine instances or storage, and so on).
- *Rapid elasticity*—The ability to add or remove resources on demand.
- *Measured service*—The cloud service provider collects, controls, reports on, and optimizes resources (bandwidth, CPU usage, and so on). Users' consumption of these resources is usually the basis for service charges.
- *Resource pooling*—Resources are shared by multiple consumers transparently. Users do not know where the resources are located or what other tenants may be using them.
- *Ubiquitous network access*—Capabilities are accessed via heterogeneous networks.[3]

Cloud computing can be delivered as a "public cloud" where anyone can register and use the resources. Examples include Amazon Web Services (AWS) and Windows Azure. There are pros and cons to public cloud computing:

[3] NIST, *The NIST Definition of Cloud Computing*, Special Publication 800-145, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

Pros	Cons
■ Low barrier to entry Increased latency ■ No up-front investment ■ A convenient pay-as-you-go model ■ Virtually infinite scalability	■ Increased latency ■ Can be costly for steady-state usage ■ Vendor lock-in (though this might be a temporary issue)

An alternative to public clouds is the "private cloud," which involves deploying a cloud onsite for internal use by a single company. This can be done by building a solution based on OpenStack or using VMware vFabric. The pros of this approach include improved performance and latency, familiarity of tools and technologies (for the cluster managers), and privacy and security. The cons include greater up-front investment, limited resources, and reduced scalability.

There's also the option of "hybrid clouds"—using both a public and private cloud as a single solution. Hybrid clouds have the advantage of providing a good balance between flexibility and performance. On the other hand, hybrid clouds mean more complexity and security challenges, and the costs savings are there only if you optimize the cloud usage; otherwise it can prove to be more costly than the other options.

Cloud capabilities are delivered over the network "as a service." There are three main types of service delivery:

- *Infrastructure as a Service (IaaS)*—This type of service is usually provided by companies such as Amazon (AWS). The cloud capabilities are basic building blocks like virtual machines, storage, network bandwidth, and so on.
- *Platform as a Service (PaaS)*—In this type of cloud computing service, the provider delivers infrastructure software components such as databases, queues, and monitoring. Windows Azure is an example of this type of service.
- *Software as a Service (SaaS)*—These services are usually provided by smaller companies that deliver complete business capabilities. An example is Salesforce.com, which delivers a CRM solution as a service.

Now that we've got the vocabulary sorted out, let's take a look at the architectural implications of the cloud.

10.2.2 The cloud and the fallacies of distributed computing

I mentioned Peter Deutsch's fallacies of distributed computing several times in this book, and for a good reason. The fallacies are base architectural requirements that you have to account for when designing distributed systems. The cloud does not get a free ticket here.

Table 10.1 shows that cloud computing doesn't solve distributed computing problems, but it helps in making some of the fallacies more apparent, so you're less likely to assume they're not there.

Table 10.1 Fallacies of distributed computing and their relevance in cloud setups

Fallacy	What does it mean in the cloud
The network is reliable	No change—this is still a problem, especially in hybrid cloud solutions. If you have a real mission-critical app, you still need a disaster recovery plan (a backup in a secondary cloud provider).
Latency is zero	Latency has not decreased in the cloud, but by deploying in data centers near your end users, you can lower it. The cloud introduces another latency-related problem.
Bandwidth is infinite	In private clouds, this hasn't changed from traditional systems. In public clouds, it depends. For internal communications between deployed servers, bandwidth has been transformed into a cost problem. For clients connecting to your cloud application, bandwidth is same old problem.
Topology doesn't change	If you assume this in a cloud solution, you'll have a real problem. The whole notion of elasticity means there's no way the topology stays the same.
There's one administrator	This is still a fallacy in the cloud—just one that it's hard to believe someone would make.
Transport cost is zero	Transport cost is still a problem. The dollar costs of moving data in and out of the cloud are more apparent than in noncloud environments because cloud services come with a price list. The additional costs (performance, latency) on transforming data structures, encryption, and so on, can still be hidden.
The network is homogeneous	The network is not homogenous, but you don't need to care as much because you can define the types of machines you need and get virtualized copies that match your needs.

The flip side is that the cloud brings with it a couple of new fallacies to watch out for:

- *Nodes are fixed*—This point builds on the "topology doesn't change" fallacy, and it means you can't assume too much about the node you are running on. Not its IP address, not that items you copied to it will be there on the next boot, and so on. Don't assume anything. Any meaningful state should be persisted elsewhere on attached or connected storage.
- *Latency is constant*—This point builds on the "latency is zero" fallacy. The fact that latency isn't constant means that if you send messages asynchronously, you can't assume they'll arrive in order. If you connect with UIs, you need to understand the variance and plan for it so that users will get an appropriate experience. For instance, in the visual search service mentioned in chapter 9, we sometimes saw 5 to 15 seconds of latency when establishing communications with the server. To get a reasonable identification time, we had to think about sending images and videos in the background, before the user chose which image to identify.

Fine, but how does all this relate to SOA?

Nodes are fixed? A real-world example

On one project I worked on, we had a service hosted in Windows Azure in two distinct setups: staging and production. We used a Windows Azure feature that allows you to do a virtual IP switch to move the staging servers to production and it worked great—except the new production (former staging) service was still pointing to the staging data store and using the staging certificate store.

We solved this by orchestrating the switch from another service that also sent events to synchronize the whole move. But we learned our lesson: in the cloud, nodes aren't fixed and you can't assume anything.

10.2.3 The cloud and SOA

SOA is probably the best architectural style to enable a transition to cloud computing, especially for hybrid and public cloud scenarios.[4] Table 10.2 shows SOA's traits and how they're a good fit for the cloud.

Table 10.2 SOA traits that are good fit for the cloud

SOA trait	How is good for the cloud
Partitioning of the enterprise/system into business components	A service is a good-sized unit to move to the cloud (as it is for moving to an external vendor). An SOA component presents a complete business function. Service boundaries already take into account the fallacies of distributed computing and already internalize the handling of messages.
Using standards-based message and contract communications	Encapsulating internal representations rather than relying on shared data means that services moved to the cloud will be able to operate in isolation from the rest of the world, communicating only via the messages defined in their contracts.
Treating service boundaries as trust boundaries	When you want to move functionality to a public cloud, it greatly helps if your software already assumes that anything foreign is hostile and should be authenticated, validated, and so on.
Keeping services autonomous	Autonomy better equips services to survive on their own. It also helps them to keep operating when other services go out.

A lot of the patterns in this book are very relevant to cloud deployments and even more so for the transition to the cloud:

[4] See the following articles: Andrew Oliver, "Long Live SOA in the Cloud Era", *InfoWorld* (June 2012), www.infoworld.com/t/application-development/long-live-soa-in-the-cloud-era-196053; Joe McKendrick, "SOA, Cloud: It's the Architecture that Matters," *ZDNet* (Oct. 2011), www.zdnet.com/blog/service-oriented/soa-cloud-its-the-architecture-that-matters/7908; and David Rubinstein, "SOA (the Term) is Dead, but SOA (the Architecture) Lives On," *SD Times* (April 2012), www.sdtimes.com/content/article.aspx?ArticleID=36566&page=3, (see particularly the "Without SOA, There Is No Cloud" section).

- *Service Bus (chapter 7)*—Helps in providing location transparency and service registration (so services will know where to find other services). Location transparency is very beneficial in the cloud because new services might be spawned in a new node with new IP address or be consolidated to a single node based on load.
- *Identity Provider (chapter 4)*—An identity provider is a crucial component when services are spread across the enterprise and a cloud, and users expect a single sign-on experience. This is even more important if you add REST to the mix, and you need to interleave WS-Trust and OAuth services.
- *Request/Reaction and Inversion of Communications (chapter 5)*—Asynchronous communication is more resilient than plain RPC, and that's a big plus in hybrid cloud setups.
- *Service Monitor and Service Watchdog (chapters 4 and 3 respectively)*—These patterns are always relevant, but they're even more important when you don't control the hardware.
- *Service Instance (chapter 3)*—This is another pattern that can help with elasticity and scaling out.
- *Virtual Endpoint (chapter 3)*—When running in the cloud, the endpoint in which services are delivered will most likely be a virtual endpoint, whether or not you like it.

In summary, SOA principles and patterns are a very good match for the cloud. The division of business capabilities into autonomous components fits well both for gradual transitioning to public clouds as well to hybrid cloud setups.

10.3 SOA and big data

There's an interesting video called "Shift Happens" (or sometimes "Did You Know?") that includes all sorts of interesting trivia on the rate at which the world is changing in the digital age.[5] Version 6 of this video includes an estimation that 40 exabytes (4.0 * 10^19) of unique information will be generated in 2012 (which is more than in the previous 5000 years combined). Most of us don't have to deal with these amounts of data, but there's no denying that the amount of data enterprises have to process and amass every year continuously grows. A TDWI research report from September 2011 states that a third of the organizations surveyed had more than 10 terabytes of data and that the number of larger sets (100s of terabytes) will triple in 2012.[6]

---

[5] Karl Fisch, Scott McLeod, and Jeff Brenman, *Shift Happens 3.0*, www.youtube.com/watch?v=cL9Wu2kWwSY. For more information on versions of the video, see the *shifthappens* web page: http://shifthappens.wikispaces.com/.

[6] Phillip Russom "Big data analytics, Fourth Quarter 2011," TDWI Research, http://tdwi.org/research/2011/09/best-practices-report-q4-big-data-analytics.aspx.

Most research organizations (like TDWI or Forrester Research) agree that big data evolves around different Vs, like velocity, volume, variety, and variability. Personally, I think the major drivers are just the first two Vs—the velocity at which you have to ingest data, along with the latency until it's usable, and the total volume of data you have to store and do something with. If you have a high peak load of messages for a couple of hours a day, and you don't need to see that data until a day later—that's not a big data problem. The same goes for terabytes of archival data you don't need to analyze, and are just storing for some regulatory reason.

Big data has a lot of implications, starting with changing the way we think about data and producing new professions like data science. It also has technical implications, which is what we'll take a look at next.

10.3.1 The big data technology mix

According to Gil Press, the first big data problem occurred in the 1880s (yes, you read that right).[7] In the late 1800s, the processing of the U.S. census was beginning to take close to 10 years. Crossing this mark was meaningful, as the census runs every 10 years and the population, and thus the amount of information, was increasing—the outlook wasn't very good. In 1886, Herman Hollerith started a business to rent machines that could read and tabulate census data on punch cards. The 1890 census took less than 2 years to complete and handled both a larger population (62 million people) and more data points than the 1880 census. (Hollerith's business merged with three others to form what became IBM.)

Today we find ourselves in a similar position when we try to solve big data problems with the traditional tools we have at hand, like our trusty RDBMSs or OLAP cubes. Those tools aren't going away, but we need additional tools—our own Hollerith machines to cope with the scale. The good news is that a lot of these new tools are emerging. The bad news is that a lot of these new tools are emerging.

Figure 10.3 shows some of the main categories of solutions for big data storage that have emerged in the market, and a few examples of tools in each category. For instance, there's the relational category, which is divided between NewSQL solutions (sharding solutions over regular RDBMSs) and massively parallel solutions. The massively parallel solutions are then divided into column-oriented solutions and row-oriented ones. On the other side of things are key-value stores, which are divided between in-memory and column-oriented solutions. The diagram is not exhaustive, but it does demonstrate the wide range of options and suboptions available. It also indicates that there's no single good solution—otherwise there'd be fewer options and everyone would standardize around the best solutions (as happened with RDBMSs 30 years ago).

[7] Gil Press, "The Birth of Big Data," *The Story of Information* (June 15, 2011), http://infostory.wordpress.com/2011/06/15/the-birth-of-big-data/.

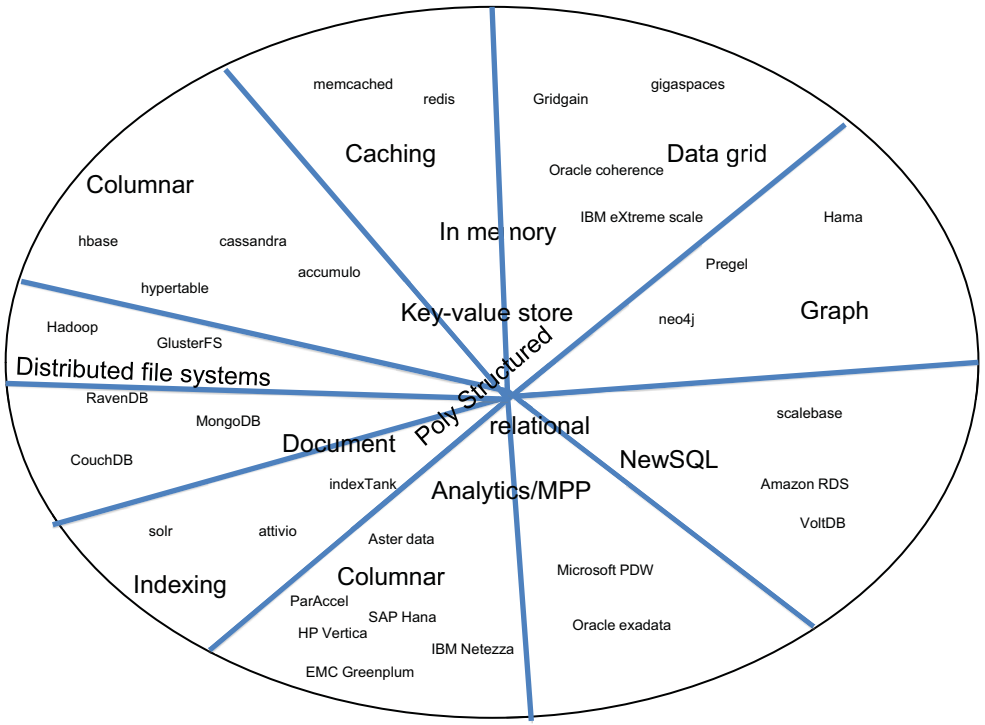

Figure 10.3 The big data storage space. There are several classes of solutions; some based on the relational paradigm and others remove database capabilities to get massive scale at cheap prices.

With this almost endless list of options to choose from, we need selection criteria in order to pick the best solution for a given project. Here are some of the criteria I find useful:

- *Type of organization*—Enterprises will likely be drawn to the more established vendors (for support, regulatory compliance, and so on). Startups will most likely gravitate toward the cheap, open source options.
- *Data access patterns*—Will you have mostly reads versus mostly writes, access based on the primary key or a lot of ad hoc queries. If you need to traverse relations back and forth (like walking a social graph), graph databases can be a good option.
- *Type of data stored*—Structured data is a good fit for relational models, semistructured data (XML/JSON) is a good fit for document and column stores, and unstructured data is good for file-based options like Hadoop.
- *Data schema change frequency*—Is your schema mostly fixed or constantly changing? Relational options are better with fixed schemas; document and name-value solutions handle open schemas better.
- *Required latency*—The faster you need the data, the more you'll want (or need) an in-memory solution.

Apache Hadoop

There are a lot of interesting technologies in the big data space, but one that stands out is Apache Hadoop. Hadoop is an open source implementation of Google's filesystem and map/reduce paradigm. Hadoop is interesting, not because it's necessarily the best solution for big data, but because it has gained massive backing from many of the major IT vendors. Oracle, IBM, EMC, Microsoft, and Amazon all offer a Hadoop distribution or service.

I've included a few sources about Apache Hadoop in the further reading section of this chapter. Or you can go straight to the Hadoop web page: http://hadoop.apache.org/.

At this point, you might be thinking that big data sounds interesting, but where's the place for SOA in all this. How can you fit SOA into all of this?

10.3.2 How SOA works with big data

How can SOA work with big data? If we accept the premise that more and more enterprises are finding that they need to handle big data, SOA should be able to work with big data, or it should be replaced with a more appropriate architecture.

One way to deal with big data within SOA is for services to use big data-related technologies within the services. A service that needs to handle semistructured data can use a document database store, and a service that needs to handle event data in near real time can use a data grid or an event-stream processing solution. Like with cloud technology, the advantage of SOA is the separation and isolation of the various services from one another. The isolation allows for gradual adoption in the enterprise, where only services that need these technologies adopt them, and other services can stay with their current technologies.

One related pattern here is the Gridable Service (discussed in chapter 3), which describes taking a computationally intensive task and dividing it between multiple services—something you can achieve with both data-grid solutions as well as big data stores that support map/reduce.

When it comes to the analysis of big data, we should distinguish between situations where the analysis can be made within the boundaries of the service and those where the analysis requires data from multiple services.

For the second type of big data analysis, where a cross-service view is needed, the ideas described in the Aggregated Reporting pattern (see chapter 7) still apply. You can get the data from all the services in a way that does not violate SOA principles as long as you make the data immutable and you know where the ownership lies. The processes that perform the actual analysis can sometimes be considered services themselves, such as a recommendation service for e-commerce solutions.

When the analysis can be handled within the boundaries of a specific service, the implementation is a matter of utilizing big data-related technologies as part of the service.

In a system I recently worked on, we had to categorize multichannel interactions (voice, email, chat, and so on). The categorization service had a subscription for incoming interactions, which arrived both in batches and in real time. The same business logic that categorized data in real time was also used in the batch. The real-time categorization had a web service and messaging endpoints, and the batch processing used map/reduce on top of Hadoop—two parts of the same business service using the same business logic to do their work. Figure 10.4 provides an illustration of this service.

In addition to the specifics around big data, you can see the application of some of the patterns described in this book within the illustration. An implementation of the Service Host pattern (chapter 2) hosts the service with its two endpoints, each of implements the Edge Component pattern (chapter 2). Note that one of the endpoints is a RESTful one, as discussed earlier in this chapter. Additionally, you can see the Service Watchdog pattern (chapter 3) in use, and that the service is deployed multiple times (the Service Instance pattern from chapter 3).

In summary, you've seen that services can be used with big data. Big data emphasizes the need to make services coarse-grained (see also the discussion of the Nanoservices

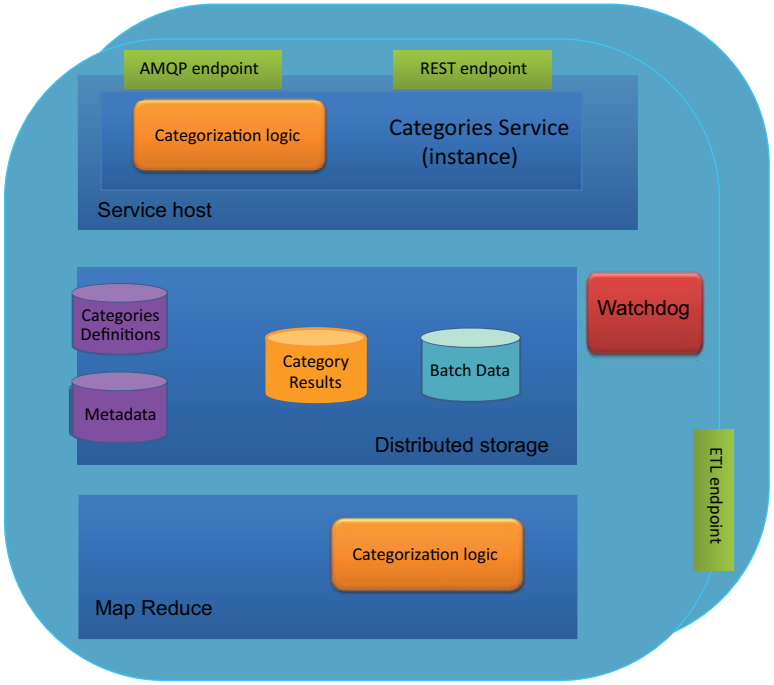

Figure 10.4 A categorization service that incorporates big data map/reduce handling with online handling. The service has three endpoints: an ETL endpoint that ingests large batches of updates, a REST endpoint that accepts small batches and online requests, and an AMQP endpoint for low-latency requests. The same categorization logic is used in the map/reduce batch processes and the online/real-time processes.

antipattern in chapter 8), and what you learned about building services is still applicable. Nevertheless, big data is changing the way enterprises handle and think about data. For SOA to stay relevant as an architectural style, it should, and can, adapt and utilize the new technologies that solve big data problems.

10.4 Summary

There are, of course, other architectural styles and technologies that are related to SOA. We discussed event-driven architecture (EDA) and SOA as part of the Inversion of Communications pattern in chapter 5. Another relevant style is domain-driven design, which isn't as popular as the three trends discussed in this chapter, but it can complement SOA as a way to design individual services.

These are the three styles we did cover in this chapter:

- *REST*—An alternative architectural style that can be merged with SOA. If you build RESTful SOA, you can benefit from both and use either SOA-style or REST-style APIs for your services (or both).
- *Cloud*—A complementary IT trend that shares its principles with SOA, and for which SOA is a very good fit.
- *Big data*—An increasingly common reality in a lot of enterprises, and to which SOA has to adapt.

Congratulations on finishing the book. You should now be able to understand the main challenges and common pitfalls of building distributed systems in general and service-oriented ones in particular. You should also have an arsenal of architectural concepts that will help you cope with these challenges and build solid systems.

The focus of this book, is on using SOA as a way to solve distributed systems challenges, so naturally this chapter's coverage of other architectural styles only scratched the surface. You can take a look at the next section for resources that will expand on the topics mentioned here.

10.5 Further reading

REST

Roy Thomas Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," (PhD thesis, 2000), www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.
Roy Fielding's dissertation is where the REST architectural style was defined, and it's still one of the best sources for learning about it.

Jim Webber, Savas Parastatidis, and Ian Robinson, "How to GET a Cup of Coffee," *InfoQ*, www.infoq.com/articles/webber-rest-workflow.
Jim, Savas, and Ian take a simple, down-to-earth example (ordering a coffee) and use that to provide a good explanation of REST principles, including HATEOAS.

Leonard Richardson and Sam Ruby, *RESTful web services* (O'Reilly, 2007).
This is probably the best book on REST.

THE CLOUD

Jothy Rosenberg and Arthur Mateos, *The Cloud at Your Service: The When, How, and Why of Enterprise Cloud Computing* (Manning, 2010).
Jothy's and Arthur's book provides a good all-round introduction to cloud concepts and technologies.

Peter Deutsch, "The Eight Fallacies of Distributed Computing," http://blogs.oracle.com/jag/resource/Fallacies.html.
James Gosling (the father of Java) concisely lists Peter Deutsch's eight fallacies on his blog.

Arnon Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," www.rgoarchitects.com/Files/fallacies.pdf.
This paper explains the fallacies in some detail.

OpenStack, http://openstack.org/.
OpenStack is an open source cloud implementation that's trying to provide an alternative to closed source implementations like Amazon's and Microsoft's.

ReaderWriterCloud, www.readwriteweb.com/cloud/.
ReadWriterWeb is a news and information site on internet-related technologies. ReaderWriterCloud is its channel dedicated to cloud computing.

BIG DATA

Alex Holmes, *Hadoop in Practice* (Manning, 2011).
This book provides a relatively up-to-date view of Hadoop and related technologies.

Lars George, *HBase: The Definitive Guide* (O'Reilly, 2011)
Lars is one of the contributors to HBase, and his book is currently the best one on HBase.

Phillip Russom, "Big data analytics, Fourth Quarter 2011," TDWI Research, http://tdwi.org/research/2011/09/best-practices-report-q4-big-data-analytics.aspx.
This is TDWI research group report and overview of the big data landscape.

NoSQL Databases, http://nosql-database.org/.
This site links to a lot of NoSQL databases (segmented by type). The site also provides links to articles related to NoSQL.

Curt Monash, DBMS2 (blog), www.dbms2.com/.
Curt Monash's site provides good information and insights on databases (SQL and NoSQL) and related technologies.

Alex Popescu, myNoSQL (blog), http://nosql.mypopescu.com/.
Alex's blog rounds up articles and news related to NoSQL.

Marco Seiriö, Marco on CEP (blog), http://rulecore.com/CEPblog/.
Marco on CEP is a good blog covering complex event processing technologies.

SOA/ENTERPRISE

SOA PATTERNS

Arnon Rotem-Gal-Oz

The idea of service-oriented architecture is an easy one to grasp and yet developers and enterprise architects often struggle with implementation issues. Here are some of them:

- How to get high availability and high performance?
- How to know a service has failed?
- How to create reports when data is scattered within multiple services?
- How to make loose coupling looser?
- How to solve authentication and authorization for service consumers?
- How to integrate SOA and the UI?

SOA Patterns provides detailed, technology-neutral solutions to these challenges, and many others, using plain language. You'll understand the design patterns that promote and enforce flexibility, availability, and scalability. Each of the 26 patterns uses the classic problem/solution format and a unique technology map to show where specific solutions fit into the general pattern.

Written for working developers and architects building services and service-oriented solutions. Knowledge of Java or C# is helpful but not required.

Arnon Rotem-Gal-Oz has over a decade of experience building SOA systems using Java and C#. He's a recognized authority in designing and architecting distributed systems in general and SOAs in particular.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SOAPatterns

"Documents a significant body of knowledge on SOA."

—From the Foreword by Gregor Hohpe, coauthor of *Enterprise Integration Patterns*

"An essential guide."

—Glenn Stokol
Oracle Corporation

"For people who actually have to ship code."

—Eric Farr
Marathon Data Systems

"Patterns are hard; this book makes them easy."

—Robin Anil, Google

"Brings structure to the wild SOA landscape."

—Rick Wagner, Red Hat

ISBN 13: 978-1-933988-26-9
ISBN 10: 1-933988-26-6
54999
9 781933 988269

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]