APPDYNAMICS

An AppDynamics Business White Paper

Architecting for the cloud designing for scalability in cloud-based applications

The biggest difference between cloud-based applications and the applications running in your data center is scalability. The cloud offers scalability on demand, allowing you to expand and contract your application as load fluctuates. This scalability is what makes the cloud appealing, but it can't be achieved by simply lifting your existing application to the cloud. In order to take advantage of what the cloud has to offer, you need to re-architect your application around scalability. In this paper we'll look at what a highly scalable cloud-based application might look like and what strategies you can use to design an application for the cloud.

Sample architecture of a cloud-based application

Designing an application for the cloud often requires re-architecting your application around scalability. The figure below shows what the architecture of a highly scalable cloud-based application might look like.

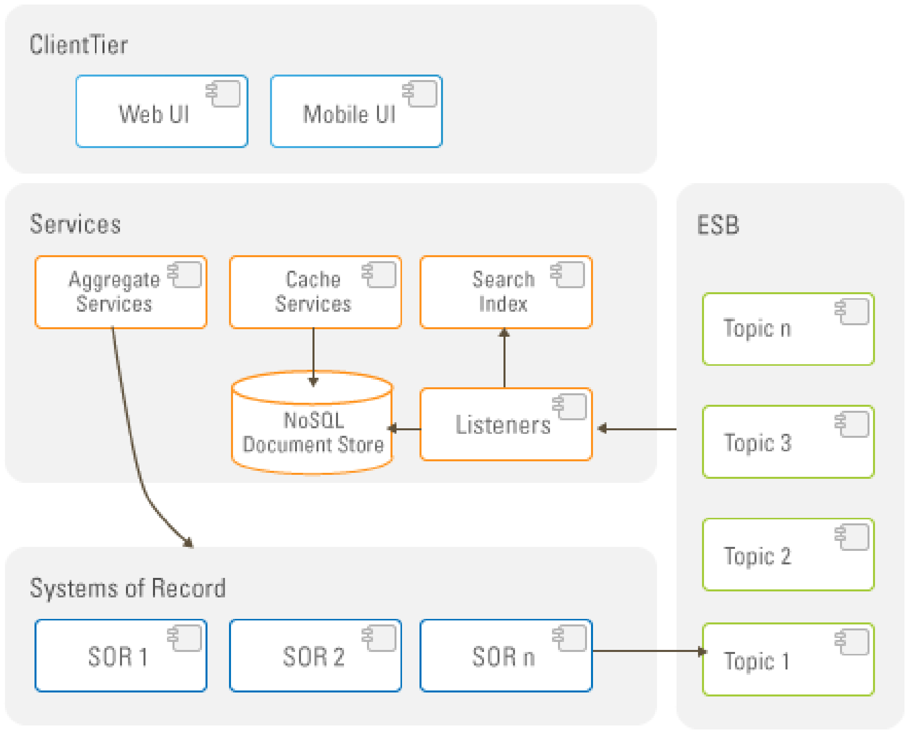

Figure 1. Sample Cloud-Based Architecture

The Client Tier: The client tier contains user interfaces for your target platforms, which may include a web-based user interface, a mobile user interface, or even a thick client user interface. There will typically be a web application that performs actions such as user management, session management, and page construction. But for the rest of the interactions the client makes RESTful service calls into the server.

Services: The server is composed of both caching services, from which the clients read data, that host the most recently known good state of all of the systems of record, and aggregate services that interact directly with the systems of record for destructive operations (operations that change the state of the systems of record).

Systems of Record: The systems of record are your domain-specific servers that drive your business functions. These may include user management CRM systems, purchasing systems, reservation systems, and so forth. While these can be new systems in the application you're building, they are most likely legacy systems with which your application needs to interact. The aggregate services are responsible for abstracting your application from the peculiarities of the systems of record and providing a consistent front-end for your application.

ESB: When systems of record change data, such as by creating a new purchase order, a user "liking" an item, or a user purchasing an airline ticket, the system of record raises an event to a topic. This is where the idea of an event-driven architecture (EDA) comes to the forefront of your application: when the system of record makes a change that other systems may be interested in, it raises an event, and any system interested in that system of record listens for changes and responds accordingly. This is also the reason for using topics rather than using queues: queues support point-to-point messaging whereas topics support publish-subscribe messaging/eventing. If you don't know who all of your subscribers are when building your application (which you shouldn't, according to EDA) then publishing to a topic means that anyone can later integrate with your application by subscribing to your topic.

Whenever interfacing with legacy systems, it is desirable to shield the legacy system from load. Therefore, we implement a caching system that maintains the currently known good state of all of the systems of record. And this caching system utilizes the EDA paradigm to listen to changes in the systems of record and update the versions of the data it hosts to match the data in the systems of record. This is a powerful strategy, but it also changes the consistency model from being consistent to being eventually consistent. To illustrate what this means, consider posting an update on your favorite social media site: you may see it immediately, but it may take a few seconds or even a couple minutes before your friends see it. The data will eventually be consistent, but there will be times when the data you see and the data your friends see doesn't match. If you can tolerate this type consistency then you can reap huge scalability benefits.

NoSQL: Finally, there are many storage options available, but if your application needs to store a huge amount of data it is far easier to scale by using a NoSQL document store. There are various NoSQL document stores, and the one you choose will match the nature of your data. For example, MongoDB is good for storing searchable documents, Neo4J is good at storing highly inter-related data, and Cassandra is good at storing key/value pairs. I typically also recommend some form of search index, such as Solr, to accelerate queries to frequently accessed data.

Let's begin our deep-dive investigation into this architecture by reviewing service-oriented architectures and REST.

REpresentational State Transfer (REST)

The best pattern for dividing an application into tiers is to use a service-oriented architecture (SOA). There are two main options for this, SOAP and REST. There are many reasons to use each protocol that I won't go into here, but for our purposes REST is the better choice because it is more scalable.

REST was defined in 2000 by Roy Fielding in his doctoral dissertation and is an architectural style that models elements as a distributed hypermedia system that rides on top of HTTP. Rather than thinking about services and service interfaces, REST defines its interface in terms of resources, and services define how we interact with these resources. HTTP serves as the foundation for RESTful interactions and RESTful services use the HTTP verbs to interact with resources, which are summarized as follows:

– GET: retrieve a resource
– POST: create a resource
– PUT: update a resource
– PATCH: partially update a resource
– DELETE: delete a resource
– HEAD: does this resource exist OR has it changed?
– OPTIONS: what HTTP verbs can I use with this resource

For example, I might create an Order using a POST, retrieve an Order using a GET, change the product type of the Order using a PATCH, replace the entire Order using a PUT, delete an Order using a DELETE, send a version (passing the version as an Entity Tag or eTag) to see if an Order has changed using a HEAD, and discover permissible Order operations using OPTIONS. The point is that the Order resource is well defined and then the HTTP verbs are used to manipulate that resource.

In addition to keeping application resources and interactions clean, using the HTTP verbs can greatly enhance performance. Specifically, if you define a time-to-live (TTL) on your resources, then HTTP GETs can be cached by the client or by an HTTP cache, which offloads the server from constantly rebuilding the same resource.

REST defines three maturity levels, affectionately known as the Richardson Maturity Model (because it was developed by Leonard Richardson):

1. Define resources

2. Properly use the HTTP verbs

3. Hypermedia Controls

Thus far we have reviewed levels 1 and 2, but what really makes REST powerful is level 3. Hypermedia controls allow resources to define business-specific operations or “next states” for resources. So, as a consumer of a service, you can automatically discover what you can do with the resources. Making resources self-documenting enables you to more easily partition your application into reusable components (and hence makes it easier to divide your application into tiers).

Sideline: you may have heard the acronym HATEOAS, which stands for Hypermedia as the Engine of Application State. HATEOAS is the principle that clients can interact with an application entirely through the hypermedia links that the application provides. This is essentially the formalization of level 3 of the Richardson Maturity Model.

RESTful resources maintain their own state so RESTful web services (the operations that manipulate RESTful resources) can remain stateless. Stateless-ness is a core requirement of scalability because it means that any service instance can respond to any request. Thus, if you need more capacity on any service tier, you can add additional virtual machines to that tier to distribute the load. To illustrate why this is important, let's consider a counter-example: the behavior of stateful servers. When a server is stateful then it maintains some client state, which means that subsequent requests by a client to that server need to be sent to that specific server instance. If that tier becomes overloaded then adding new server instances to the tier may help new client requests, but will not help existing client requests because the load cannot be easily redistributed.

Furthermore, the resiliency requirements of stateful servers hinder scalability because of fail-over options. What happens if the server to which your client is connected goes down? As an application architect, you want to ensure that client state is not lost, so how to we gracefully fail-over to another server instance? The answer is that we need to replicate client state across multiple server instances (or at least one other instance) and then define a fail-over strategy so that the application automatically redirects client traffic to the failed-over server. The replication overhead and network chatter between replicated servers means that no matter how optimal the implementation, scalability can never be linear with this approach.

Stateless servers do not suffer from this limitation, which is another benefit to embracing a RESTful architecture. REST is the first step in defining a cloud-based scalable architecture. The next step is creating an event-driven architecture.

Event-Driven Architecture

An event-Driven Architecture (EDA) is one of the keys to scalability. The core concept underlying EDA is the idea that systems notify other systems of changes using events and those events are delivered asynchronously. EDA promotes loose coupling because the producer of an event does not need to know anything about its various subscribers – it defines the structure of its events and publishes them at appropriate times. Consumers can be added at any point: they subscribe to receive notifications and process them when they arrive. This is illustrated in figure 2.

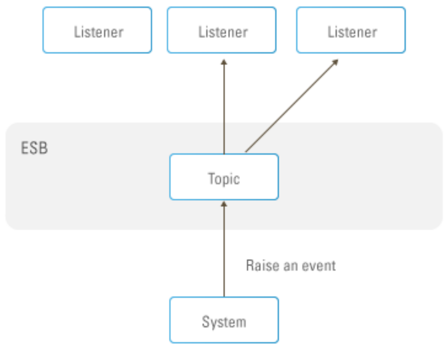

Figure 2. Event-Driven Architecture

Aside from being a good strategy for loosely coupling systems, a key factor that makes EDA scalable is its asynchronous communication paradigm. Asynchronous communication means that consumers process events as they are able to: if an application becomes saturated by excessive load, an asynchronous application may slow down as events back up, but it will not go down. Consider processing status updates on a social media site: as load increases the events may back up, but the consumers only consume them at the rate they can. So under heavy load you may have to wait five minutes to learn that I'm at the laundromat, but that's hardly the end of the world.

Additionally, EDA is a poster child for the cloud because as events back up, it is easy to start up additional event listeners to process the backed up events. Event listeners can leverage the elasticity that the cloud provides.

EDA is the architecture, but it does not dictate the implementation. Most often EDA is implemented on top of an enterprise service bus (ESB) and uses topics as the means of communication. As mentioned earlier, topics are preferred over queues because they operate in a publish-subscribe manner. A message producer publishes an event and anyone interested in that event can process it. If we used queues then every time a new subscriber was added, the producer would need to be updated to publish to the new consumer – this is an example of tight coupling, which we want to avoid.

ESBs are the most common choice to implement EDA, but they are not the only choice. In practice, while ESBs are advanced and have been developed by smart programmers, they are often difficult to maintain and can cause performance bottlenecks, so some architects have searched back through Fieldings' RESTful dissertation to discover the potential for using Atom feeds for event publishing.
In this model, when something meaningful happens in a component, instead of publishing a message to a topic, the component publishes the message to an Atom feed. Atom is an HTTP-based replacement for RSS (Really Simple Syndication) designed to publish things like newsfeeds, but because of its HTTP-based implementation it is very RESTful in nature. When using Atom as a replacement for an ESB, instead of subscribing to topics, a consumer instead polls the component's Atom feed. It then processes messages at its own rate and returns for more messages when it is ready.

Because there are typically not too many subscribers to a single component's Atom feed, the resultant load is not significant and each consumer controls its own capacity. Additionally, if a consumer goes down there is no concern that it has lost messages because it simply continues from where it left off. Finally, Atom feeds provide a source of history for a component that would have otherwise required quite a bit of effort to implement.

Atom is not a panacea, however. If your application is sensitive to latency then Atom is not a good choice. An ESB delivers a message as soon as it is ready but the Atom feed is subject to the polling interval of its consumers. Additionally, when there are components that do not generate many messages, there is the wasted overhead of continually polling the component. If latency is not an issue for your application, using an Atom feed to facilitate EDA is a simple, elegant, and fault-tolerant solution.

Figure 3 shows a graphical comparison of using an ESB and using an Atom Feed to implement an EDA application.

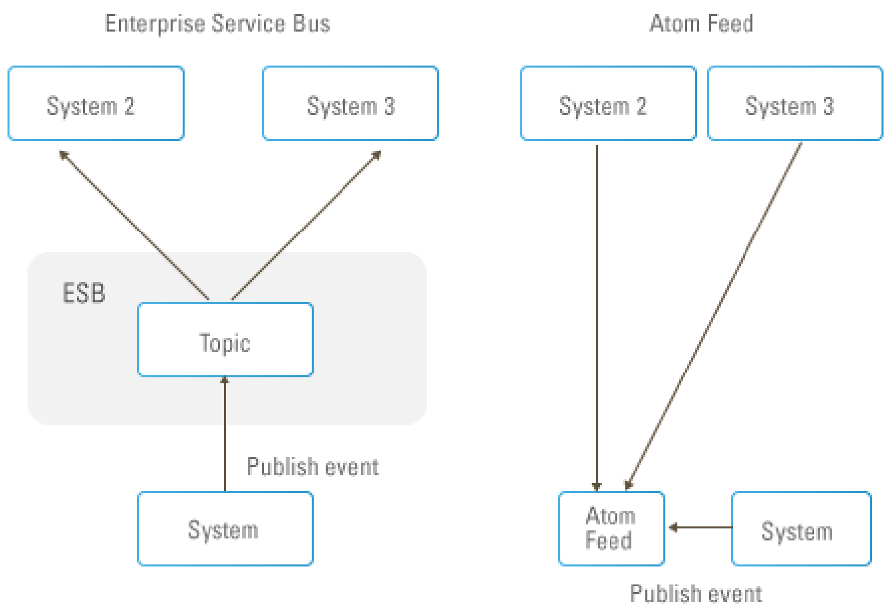

Figure 3. ESB versus Atom as an EDA Implementation

Deploying to the cloud

This paper has presented an overview of a cloud-based architecture and provided a cursory look at REST and EDA. Now let's review how such an application can be deployed to and leverage the power of the cloud.

Deploying RESTful services

RESTful web services, or the operations that manage RESTful resources, are deployed to a web container and should be placed in front of the data store that contains their data. These web services are themselves stateless and only reflect the state of the underlying data they expose, so you are able to use as many instances of these servers as you need. In a cloud-based deployment, start enough server instances to handle your normal load and then configure the elasticity of those services so that new server instances are added as these services become saturated and the number of server instances is reduced when load returns to normal. The best indicator of saturation is the response time of the services, although system resources such as CPU, physical memory, and VM memory are good indicators to monitor as well. As you are scaling these services, always be cognizant of the performance of the underlying data stores that the services are calling and do not bring those data stores to their knees.

Figure 4 shows that the services that interact with Document Store 1 can be deployed separately, and thus scaled independently, from the services that interact with Document Store 2. If Service Tier 1 needs more capacity then add more server instances to Service Tier 1 and then distribute load to the new servers.

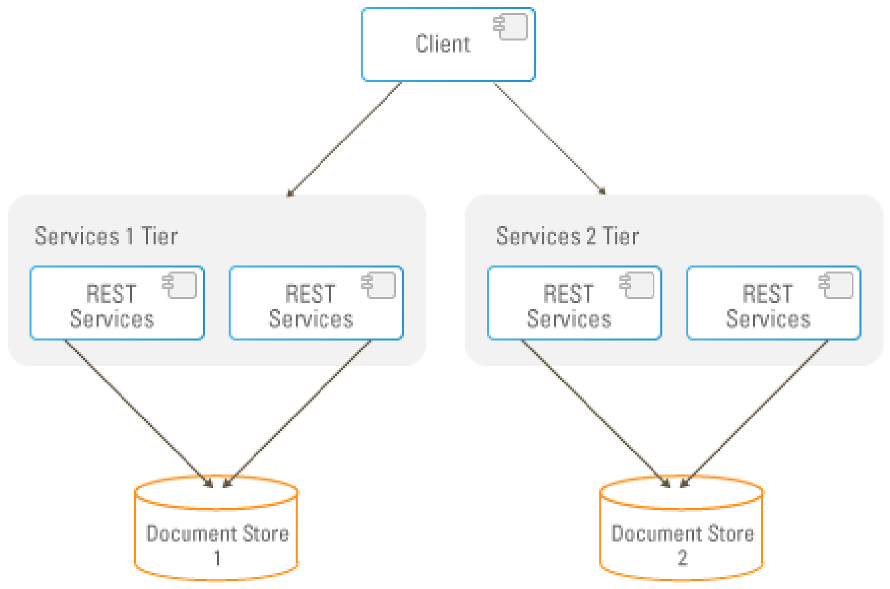

Figure 4. Scaling RESTful tiers

Deploying an ESB

The choice of whether or not to use an ESB will dictate the EDA requirements for your cloud-based deployment. If you do opt for an ESB, consider partitioning the ESB based on function so that excessive load on one segment does not take down other segments. This segmentation is shown in figure 5.

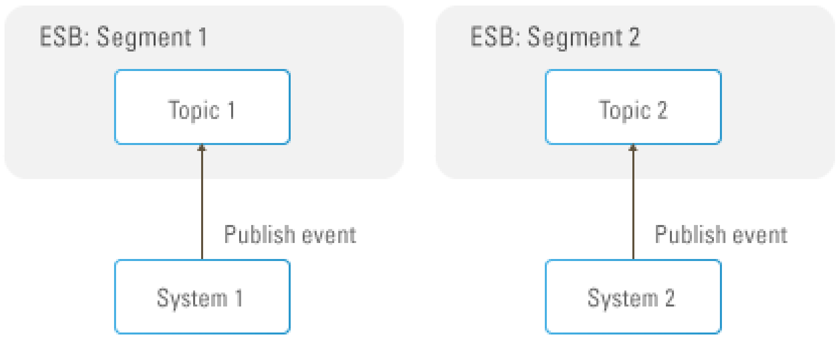

Figure 5. ESB Segmentation

The importance of segmentation is to isolate the load generated by System 1 from the load generated by System 2. Or stated another way, if System 1 generates enough load to slow down the ESB, it will slow down its own segment, but not System 2's segment, which is running on its own hardware. In our initial deployment we had all of our systems publishing to a single segment, which exhibited just this behavior! Additionally, with segmentations, you are able to scale each segment independently by adding multiple servers to that segment (if your ESB vendor supports this).

Conclusion

Cloud-based applications are different from traditional applications because they have different scalability requirements. Namely, cloud-based applications must be resilient enough to handle servers coming and going at will, must be loosely-coupled, must be as stateless as possible, must expect and plan for failure, and must be able to scale from a handful of server to tens of thousands of servers.

There is no single correct architecture for cloud-based applications, but this paper presented an architecture that has proven successful in practice making use of RESTful services and an event-driven architecture. While there is much, much more you can do with the architecture of your cloud application, REST and EDA are the basic tools you'll need to build a scalable application in the cloud.

Try it FREE at
www.appdynamics.com