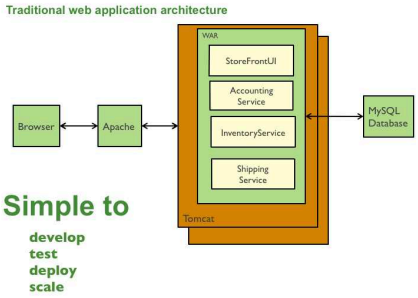
Decomposing applications for deployability and scalability

Part 1 - The (sometimes evil) monolith

Chris Richardson @crichardson

In April 2012, I started giving a talk called "*Decomposing applications for deployability and scalability*" [1]. It begins by describing how even though monolithic applications are initially simple to develop, deploy and scale, as they grow more complex the monolithic architecture has several drawbacks. For example, development slows down and applications become much more difficult to redeploy frequently and to scale. Also, adopting newer technologies becomes challenging and you risk being locked into a particular platform. The talk then describes how the solution is to decompose a monolithic application into a collection of independently deployable and scalable services. This article is part 1 of a reimagined transcript of the talk. It describes the limitations of a monolithic architecture. Later articles explore a more modular approach.

Let's imagine that you are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. It's quite likely that you would build an application like the one shown on this slide. The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.



The application is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat. A Rails application consists of a single directory hierarchy deployed using either, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat.

This is an extremely common way to architect an application. Most applications that I've developed have looked like this. That's because monolithic applications are simple to develop and test. They are also simple to deploy since there is just one

---

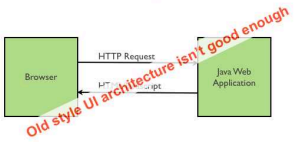
[1] http://slidesha.re/decomapps

deployment unit. Moreover, it's usually easy to scale this kind of application by running multiple copies behind a load balancer.

Unfortunately, this simplistic architecture has a number of limitations. For example, it constrains your choice of technologies. It also can impede development especially as the application and engineering team increase in size. Let's look at these problems.

But there are problems

Users expect a rich, dynamic and interactive experience on mobile devices and desktop



Real-time web ≅ NodeJS

The first problem is that users expect a rich, interactive user experience. It's no longer adequate for an application to serve up static HTML. Instead, applications need to push events to the browser. Today, the easiest way to build a real-time web application is with HTML5 and NodeJS, which is an event-driven, JavaScript-based framework. This means that you need to add two new components to your application: an HTML5 client application and a NodeJS-based front-end server.

The second problem with a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can't scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic.

One dimensional scalability

- Scales to handle transaction volumes
- Higher transaction rate required ⇒ run more instances
- Some clouds will do this automatically, others easily
- No code changes required

BUT

- Does not scale with increasing data volumes
- Caching is less effective
- Requires more memory
- Requires more I/O

Components that don't cluster

- Most application components are easily clustered:
  - Most services are stateless
  - Server-side session state: sticky session routing/state server

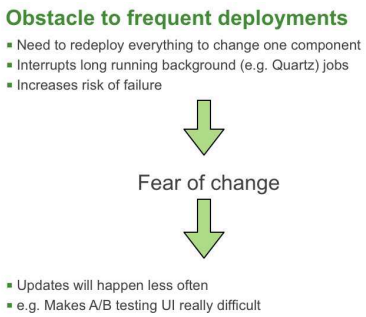
BUT there are exceptions

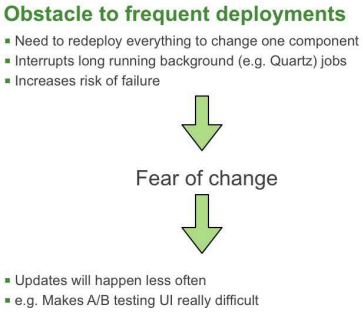
- Singletons that maintain global application state:
  - e.g. CEP engine analyzing event stream over HTTP
- Commercial software with a per-server license fee
- If one application component can't be clustered ⇒ you can't cluster the application!

The third problem with a monolithic architecture is that if one application component can't be clustered then you cannot cluster the application. Most application components are stateless and can be clustered. Components that maintain session state can be clustered either by using sticky sessions or an external state server. The problem is with components that store mutable, global application state.

This kind of stateful component is rare but sometimes necessary. For example, I once developed an application that used Esper[2], which is a Complex Event Processing (CEP) engine, to analyze a stream of events that were delivered over HTTP. Esper, like other CEP engines, is an inherently stateful component that can't be easily clustered. As a result, we could only run a single instance of the application, which reduced availability and made it much more difficult to get a good night's sleep.

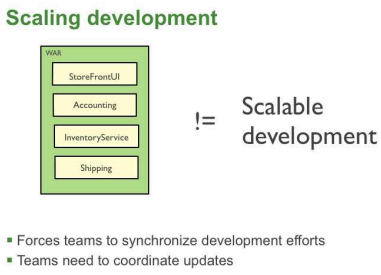
A monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven't been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.





A large monolithic application also slows down development. As the application gets larger, the IDE will get slower and slower. The application also takes longer to start. For example, I recently had lunch with the developer of a consumer web site who told me that the startup time for his application was over three minutes. Slow IDEs and long startup times lengthen the edit-compile-debug cycle and reduce developer productivity.

A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.



---

[2] http://esper.codehaus.org/

Another big problem with the monolithic architecture is that you are married to a particular technology stack and in some cases, to a particular version of that technology. With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let's imagine that you choose the JVM. You have some language choices since as well as Java you can use other JVM languages that interoperate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture.

Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It's possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

Summary

As you can see, a monolithic application is easy to develop, test, deploy and scale but has numerous limitations. In part 2, we'll look at more modular approach that overcomes those limitations.

