Tagged Architecture - Scaling to 100 Million Users, 1000 Servers, and 5 Billion Page Views

Monday, August 8, 2011 at 9:20AM

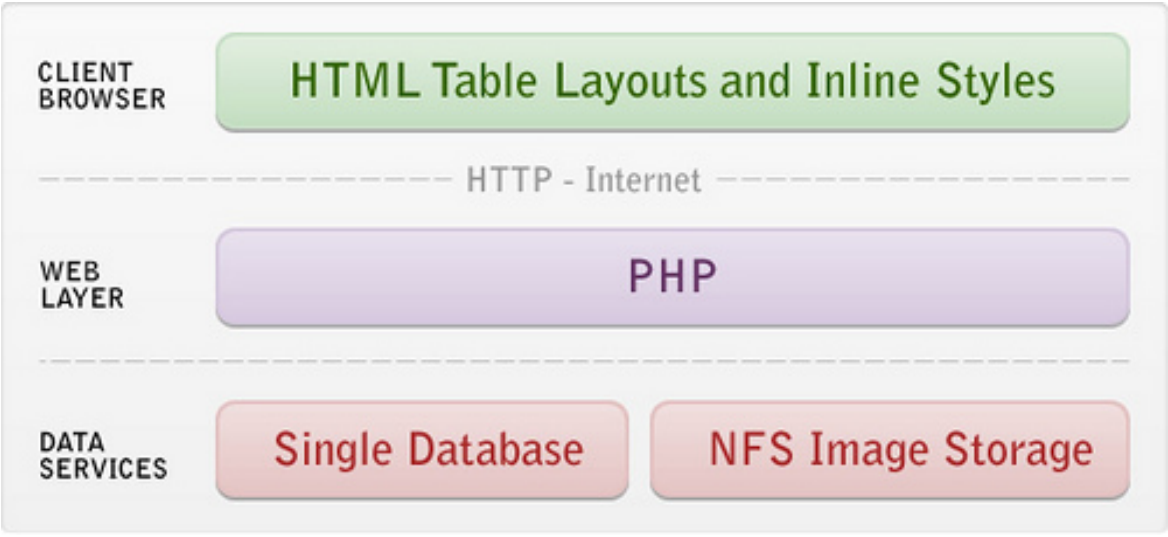
Todd Hoff in Example

This is a guest post by Johann Schleier-Smith, CTO & co-founder, Tagged.

Five snapshots on how Tagged scaled to more than 1,000 servers

Since 2004, Tagged has grown from a tiny social experiment to one of the largest social networks, delivering five billion pages per month to many millions of members who visit to meet and socialize with new people. One step at a time, this evolution forced us to evolve our architecture, eventually arriving at an enormously capable platform.

V1: PHP webapp, 100k users, 15 servers, 2004



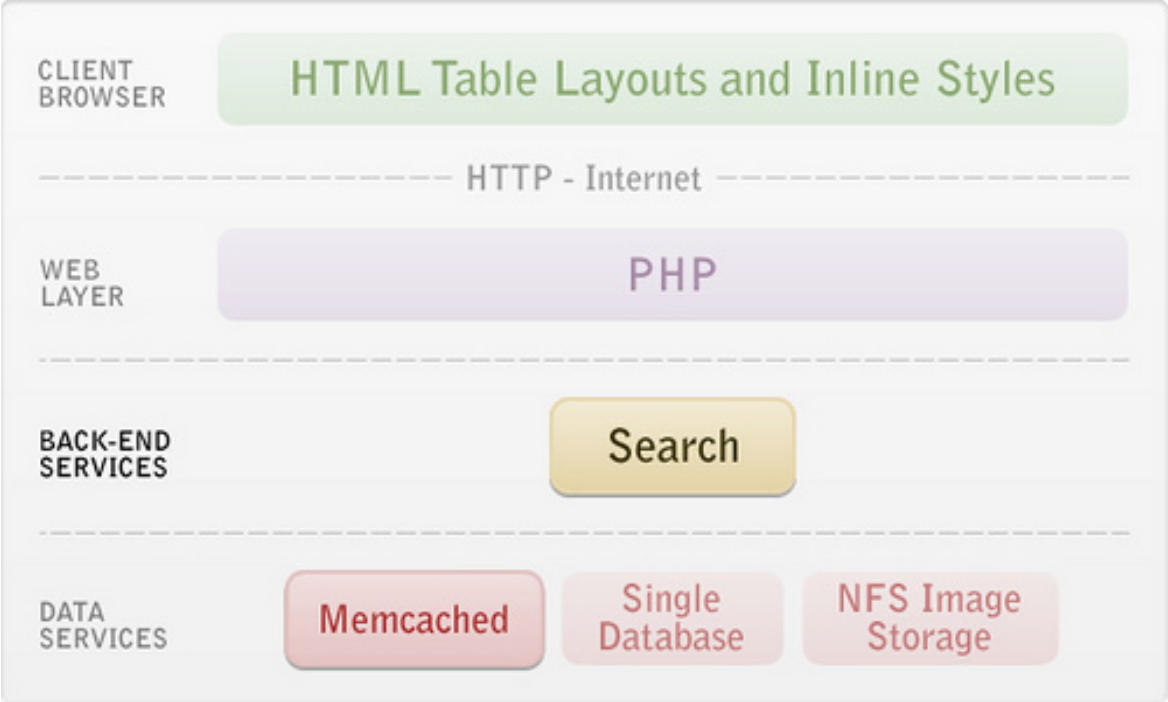
Tagged was born in the rapid-prototyping culture of an incubator that usually launched two new concepts each year in search of the big winner. LAMP was the natural choice for this style of work, which emphasized flexibility and quick turnaround at a time when Java development was mostly oriented towards development at large enterprises, Python attracted too few programmers, and Perl brought the wrong sort. Also, we

knew that Yahoo was a big proponent of PHP, so it would be possible to scale the business when the need arose.

Significant experience running MySQL on previous projects had left me with a love-hate relationship with the technology. In the spirit of experimentation we purchased a few entry-level Oracle licenses for Tagged to see whether that would work better.

Remarkably, many smaller web sites are still built just like the original Tagged. There is beauty in simplicity, and the two-way division between stateless PHP and stateful Oracle concentrates the trickiest bits in a single server, while extra page-rendering compute-power is easy to add.

V2: Cached PHP webapp, 1m users, 20 servers, 2005



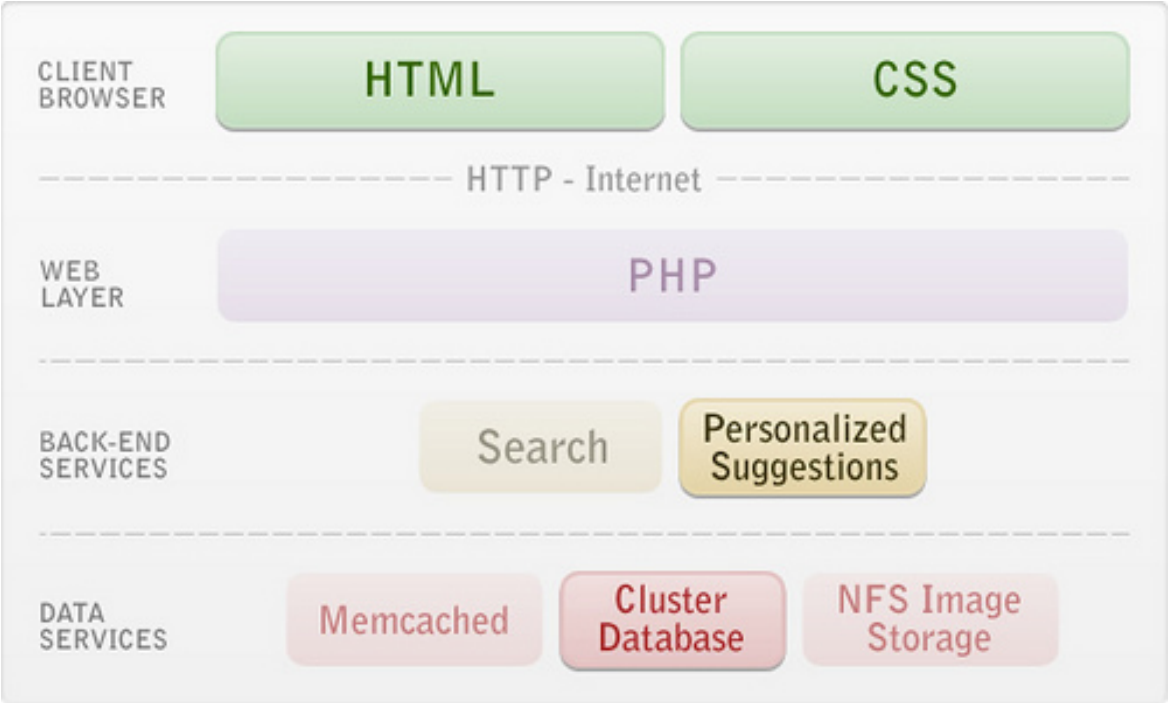
Even at eight servers Tagged had more web traffic than most of us had known. Fortunately, memcached brought dual advantages, removing over 90% of database reads, and ensuing that social networking pages packed with diverse information would render quickly.

From the start, our object caching emphasized explicit cache updates in favor of simpler techniques such as deleting invalid keys, or expiring stale data based on timers. At the cost of more complex code, this reduces database load substantially and keeps

the site fast, particularly when frequently-updated objects are involved.

Our site continued to evolve in complexity beyond standard social networking features (friends, profiles, messages) with the addition of search and social discovery functions. My team talked me into using Java to build search so that we could benefit from the Lucene libraries. I was relieved when we learned to run it well, and my reluctance born of early experiences with JDK 1.0 was transformed to enthusiasm for the platform.

V3: Databases scaling, 10m users, 100 servers, 2006

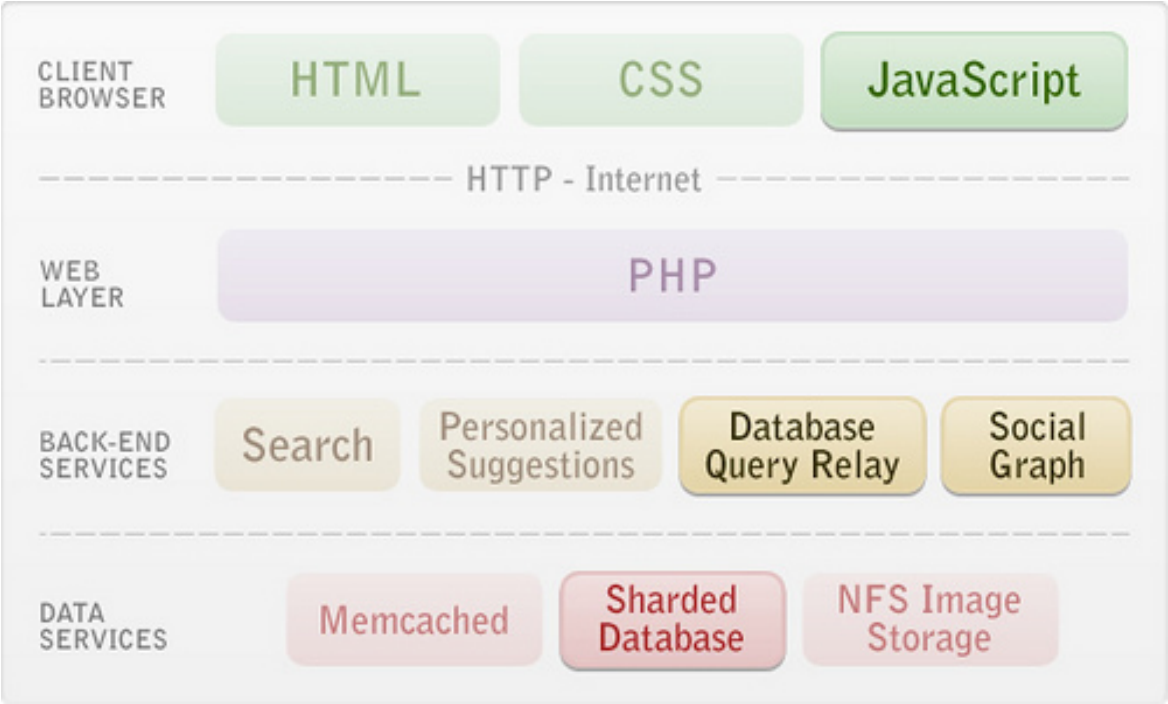


With 10 million registered users and thousands online at any moment we approached the challenge that I had been dreading. We had just raised capital and were working hard on growth, but the database was bursting for capacity. We scrambled to release one caching or SQL tuning optimization after the other, but the CPU our servers would time and again trend towards the 100% mark.

The idea of scaling up offered a quick fix, but the multi-socket server hardware could cost millions, so we opted for Oracle RAC, which let us use standard networking to hook up lots of several commodity Linux hosts to build one big database. When joined with the advantages of the latest CPUs, Oracle RAC delivered a crucial 20-fold capacity increase over our first database server, and allowed application developers to stay focused on building new features.

Java edged further into the environment when Tagged began to offer personalized people-matching recommendations by sewing together statistics from a large in-memory data set, something entirely impractical to do with PHP.

V4: Database sharding, 50m users, 500 servers, 2007



Sharding the database was without a doubt the most challenging, but also the most rewarding episode in scaling Tagged. By splitting up users among multiple databases we finally had a design that at all places allowed us to scale just by adding hardware.

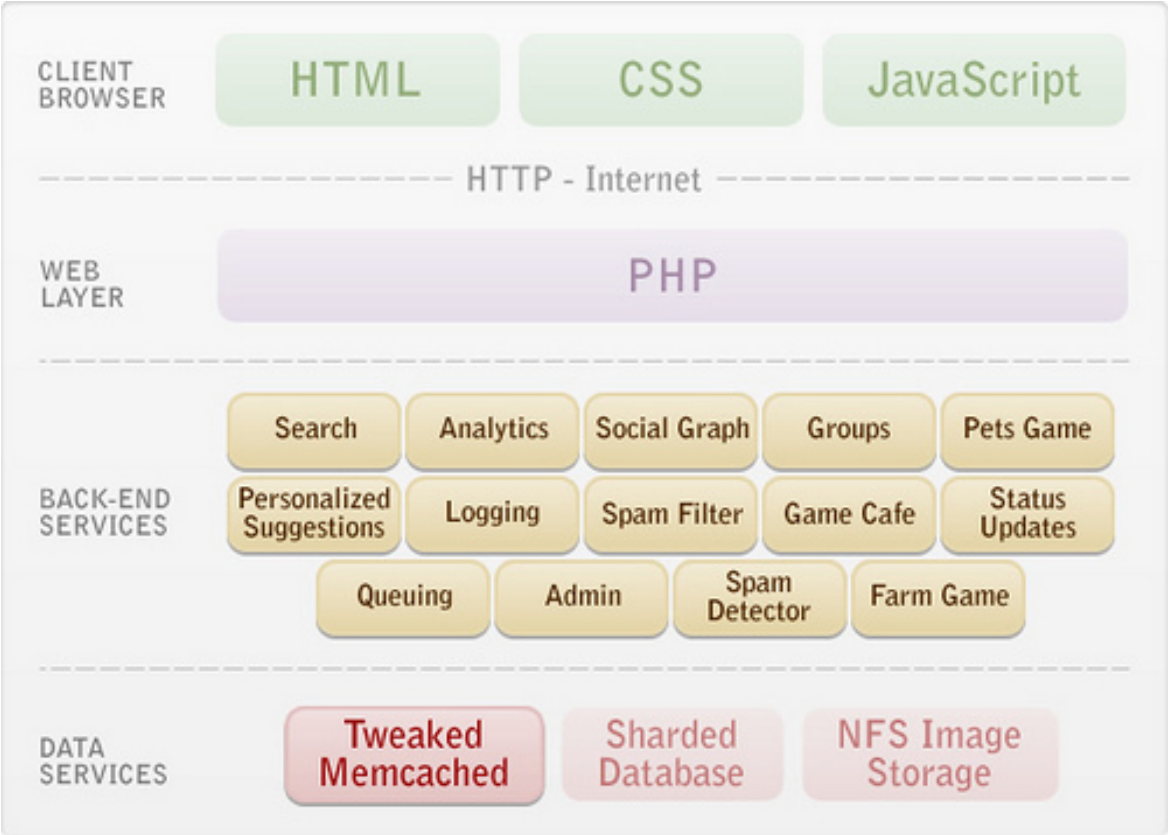
Our rule at Tagged is to shard each table across 64 partitions, and we hold firm to this default unless there is a very compelling reason to make an exception. Only certain games that benefit from high-performance protected transactions between players are vertically partitioned in a separate database.

Sharding existing data represented a complex transformation across several terabytes. At first we attacked features one-at-a-time, relying on application code to replace joins, but eventually we encountered a bundle of tables at the core of the application too closely linked for this approach. Writing migration software to generate SQL, we exported, transformed, and reloaded hundreds of millions of rows, using triggers to track changes on source system and updating targets incrementally so that the final

sync involved an outage of less than 30 minutes.

Having many databases means having many database connections. Especially as we added more "social discovery" functions like Meet Me, our first dating feature, sharding would have overwhelmed PHP, which lacked Oracle connection pooling. To cope, we built a Java application that exposes a web service for running queries, one which also continues to provide a very convenient monitoring point and allows graceful handling database failures.

V5: Refinements and extensions, 80m users, 1,000 servers, 2010



Here we jump ahead several years. With the crux database scalability problems solved, we found it straightforward to support expansion by adding hardware. PHP and memcached continued to serve us well, supporting rapid feature development.

During this time, scalability considerations shifted towards mitigating failures, addressing the threat of an increasing number of breakable parts. Protecting the web layer from problems at its dependencies was achieved through load balancer health checks and automatic shutdown of unresponsive services. We also engineered core

components for resilience, e.g., if memcached becomes overloaded with connections it must recover immediately once that burden is removed.

Java achieved a much more prominent role, in part due to increasing acceptance and expertise, but also because of increasing challenges. To combat spam and other abuses, our algorithms take advantage of large shared memory spaces, as well as compute-intensive techniques. Social games also benefited from the performance and concurrency control of Java, but there has been a cost in complexity; we now need to manage many more distinct pools of applications than before.

The future

Today, Tagged delivers five billion page views each month to its millions of members. Since we've arrived at scalable design, we can spend most of our energy on creating features that serve users better. We have effective tools for creating scalable software, but we can imagine much better ones, so current investments focus on software libraries, improving programmer effectiveness and productivity, and Stig, our upcoming open source, graph based database project designed for large scale social networks, real-time services and cloud applications.

Related Articles

- Johann Schleier-Smith Interview with theCube
