csc_253_unit_7

6

Iain McCulloch

Contents

1 CS253 Unit 7: Changing the World - Building a successful web application project

Contents

1.1 1. Introduction and Overview

This unit is basically a wrap-up unit, but it will contain some fun stuff. It is all about real-world issues. Things like what to look for in a web framework, how to get hosted and some of the decision making that should underpin these choices.

We will talk about how Reddit began on a single machine and how they grew and scaled, learning as they developed. This will include fascinating insights from guest speakers from Reddit and Udacity who can provide further real-life insights into some of the issues that we have discussed on this course.

There are no quizzes in this unit.

1.2 2. Code Organisation

Steve Huffman

A question that comes up a lot is “How do I organise code?”. You don’t want to keep all your code in one Python file, so what is the correct way to organise things?

Well, the first thing to point out is that there is no, one, *correct* answer. Whether you are writing web apps, or any other kind of software, how you organise your code is something that is personal to you, and will come from your own experience. Steve goes on to explain his approach:

When he starts out, Steve tends to begin with everything in a single file. The file will probably have sections for:

- handlers, which define what to do when a particular URL is hit
- URL mappings, which map URLs to the correct handlers
- db Models

Alongside these, Steve will have a number of separate files for static content, including things like css, JavaScript, images, and so forth. Another thing that Steve will almost always keep separate from the beginning are his template files.

Steve's main Python file is often called simply main.py.

The first thing that he does with main.py is to pull out the db models into a separate file. If you are not using Google App Engine, you may need to write an additional piece of code called the ORM (Object Relational Mapping), which maps your Python objects to the relational database. We haven't had to deal with that in this class because App Engine provides this functionality. Steve will generally have a separate file for each type of data, e.g. posts, art, users, etc. Class specific functions would go into the appropriate file, so, for example password hashing functions would go in the users file, and functions to get the top posts etc. would go into the posts file.

The next thing that Steve almost always has is a file called utils.py (which might actually be a series of files in a directory). This holds all the 'random stuff' and handy 'things'. Things like make_secure_val() or make_salt() or random_string(). In general, Steve likes to put as much into utils as he can, so it will include a lot of date-manipulation and string-manipulation functions and also escaping functions. This should be just a list of 'flat' functions – functions that have no dependencies on any other part of the project. It is really important that it is only a one-way inheritance. Handlers can import from utils. Database can import from utils. Anything can import from utils, but functions in utils never import from other parts of the project. In the Reddit source code there are about 100 functions that are used all over the rest of the code.

The final thing that Steve does (and this might be quite time-consuming) is to take the handlers out of main.py. There may be separate files for each type of handler.

So by now, main.py is just the URL mappings and a bunch of imports.

For the file structure, Steve uses a structure something like this:

- /main.py
- handlers.py
- blog.py
- ROT13.py
- etc.
    - /lib
    - utils.py
        - /db models
    - /templates
    - base.html front.html
    - etc.
    - /static
    - main.css etc.

At the top-level in the main directory there will be main.py and something like a handlers.py file that hold the generic handler for things that happen on every request, together with the specific files for the project (like blog.py or ROT13.py etc.). The bulk of the code is in these files. Below that directory there will probably be a directory called 'lib' which contains stuff like the utils file and, within that directory, a further sub-directory for the database stuff.

There will also be directories called 'templates' and 'static' which contain the template files and static files like CSS.

-

1.3 3. Hosting

Steve Huffman

There are a range of questions about hosting. We have been using Google App Engine, but what other choices are there? What issues does each potential solution bring with it? Let's see if we can't answer a few of these questions here.

1.3.1 3.1. Local Hosting

The first option is to run your website locally. Essentially, running your website from a machine that might be in your home or small office.

This works fine for a single user (the sort of thing we have been doing in this class) where the site doesn't have to be on all the time, but in almost every other type of use, this option really doesn't make sense. Some of the main drawbacks are:

- The site isn't always on.
- The site isn't always accessible.
- Your IP may change (or you may need to pay extra for a static IP address).

1.3.2 3.2. Co-location Hosting

The next thing that you can do is to 'co-locate'. Basically, this means buying space on a rack in a datacentre. You still buy your own machine (or machines) and then install them in the datacentre. This means that you still have control over the machines and just pay for rent, power and bandwidth.

The downside is that it involves a lot of work maintaining the system. You need to administer the machines, install software (and software upgrades), and maintain the machines.

Very few large websites don't control their own machines.

1.3.3 3.3. Managed Hosting

Another option is .managed hosting'. This industry is changing right now. Just a few years ago, you would rent the number of preconfigured machines that you needed from a datacentre. Now there is a move towards managed hosting in the cloud from companies like AWS, Rackspace and Linode. Reddit and Hipmunk both use AWS

You rent machines from the provider and pay an agreed amount for bandwidth. Although you can install the OS, in reality you are probably just going to be configuring the OS so you are reducing the number of sysadmin tasks that you need to carry out. From your perspective all the admin and maintenance tasks are carried out virtually.

Reddit progressed from local hosting, through co-location hosting and is now operating in a managed hosting environment with AWS.

1.3.4 3.4. Zero System Administration

At the next level up, we have things like Google App Engine and Heroku which are completely managed for you. You don't even need to think about machines or OS, you just upload your code and they run the databases, sharding etc. There is virtually no sysadmin. A lot of big websites have been built in just this environment – Udacity being one of them.

However, it can be very difficult to customise things. It can be difficult to do anything that the providers of the systems haven't already thought of beforehand.

There is no real penalty for choosing the 'wrong' hosting solution when you start out. You can just move to a different model. That doesn't mean that moving will necessarily be easy. When Reddit moved from co-location to AWS, they were already a fairly large website. They needed to run two infrastructures in parallel for a while as they slowly migrated from one to the other.

1.4 4. Web Frameworks

Steve Huffman

Another good question is about how to choose between web frameworks. The web framework is the application that interfaces between your program and the Internet. In this class we have been using the Google App Engine framework, webapp2. This handles HTTP, scheduling, parsing basic headers,, converting your response object into the appropriate HTTP to send to the browser, URL mapping and so forth. Different web frameworks give you more or less control and so require you to do more or less work.

Google App Engine operates at the level that Steve likes to work at. From his point of view, the important features are:

- Direct access to GET and POST

    - ♦ Understanding web apps at the method level is very important.
- Direct access to the request

In other words, you have access to the low-level stuff that you may want to modify, but it isn't so low that you're having to worry about HTTP versions or host headers (unless you really want to).

- A lot of frameworks provide features that Steve would consider unimportant, such as:
- Sessions
- Caching
- Automatic forms
- db - ORM

The problem with these tools is that they may not allow you to manage custom behaviours in quite the way that you would want to do. As a guide, avoid systems that seem too "magical". If you are so far away from the actual request that you don't actually understand what just happened, you are asking for trouble down the line.

1.4.1 4.1. Templates

Template languages come in all shapes and sizes. One that Steve is partial to is called Mako. Mako templates are written in Python. The template language that we have been using in this class is called jinja2, which Steve has also found to have worked well.

The key to using templates is having the discipline to separate code from templates. A lot of these template systems allow you to put arbitrary code into them, but you should try to keep the amount of code in your templates to an absolute minimum. You have a whole programming language (in our case Python) which lets you do all sorts of wonderful things. Why would you choose to use a broken subset of that in your templates?

1.5 5. How I Work

Steve Huffman

Here, Steve describes how he works when developing a web app.

When developing a web application you will generally have a browser window and your editor open together. Side-by-side is ideal, but not always practicable.

Something that people often neglect is to have the Terminal open nearby. Steve almost always runs the app server out of a terminal (even when using App Engine), so that he can see the logs. If you are developing without the ability to see your logs, there is a lot of stuff that you are going to miss. You can find many, many bugs by running the web site in the browser and watching the logs. In many cases, there might be an error that you aren't even aware of, but that could be a big deal in production, which you can spot and fix in 10 seconds if you are monitoring the logs.

- Another thing that Steve sees as a problem with App Engine right now is that if you are writing code that writes to a database, it is critical that the queries that you run are printed to the log. If you have a complicated setup with your database and your cache or memcache, you need to make sure that it is working. If you are scrolling through your site and see lots of database queries appearing in the logs, that is a sign that something is broken! In general, you will want to make sure that every page you run doesn't hit the database unless absolutely required.

Try to get into the habit that when you switch from your editor to your browser, you instead switch to your browser plus the Terminal so that you can see things working. If you get a sense of how the logs scroll when your website is working normally, when it deviates from this you will spot it and so pick up bugs more quickly.

1.6 6. Reddit

Steve Huffman

We have referred to Reddit quite a lot over the course of this class. We've looked at database examples and system architecture examples, and seen some of the mistakes that Steve made while developing Reddit. Let's now take a look at how Reddit was built, and how it grew over the years.

Reddit began in June 2005 with Steve Huffman and his co-founder (and college roommate) Alexis Ohanian. Steve was the engineer, and Alexis was the everything-else guy.

The first version of Reddit was written in Lisp. Everything was created in Lisp. This included the handlers, HTML CSS and JavaScript (Lisp is a language that lends itself to generating other languages and other pieces of code really nicely). This meant that the entire site was built from Lisp and Postgres. There was no memcache at that time, Steve just stored everything in an in-memory hash-table.

1.6.1 6.1. Reddit Architecture

The original Reddit architecture was something like this.

They had a single rented machine which ran Lisp and Postgres. The database structure had tables for:

- Links (score, title, URL)
- Votes (link-id, direction)
- Users (name, password)

(Reddit didn't include comments for a long while).

The database also had a lot of joins.

The initial setup lasted for about six months. At about that time, Reddit merged with another company called Infogami and added a couple of engineers, Aaron Swartz and Chris Slowe, to the team. The expanded team re-wrote Reddit from Lisp into Python, and also moved the database onto a second machine. Going from a single machine to two machines and separating the app server from the database gave about a four-times improvement in speed.

At this time, the major issue was downtime. Whether in Lisp or Python, sometimes the site would just crash and they would get a notification and one of the team would have to bring the site back up. In retrospect, Steve says that it makes him cringe to think about how much stress he went through worrying about whether their website was up or not. Particularly since there are so many easy ways to avoid that scenario. Shortly after that, they added a piece of software called Supervise which monitored the app server, and if the application crashed it simply restarted it.

Essentially, they were very lucky that they never lost an actual machine through hardware failure. Especially since losing the database machine would have meant losing all the data (they weren't doing good backups!). Hardware failure is a fact of life. If you run your machines 24 hours a day, 7 days a week, at high load you are going to get hardware failures including hard disks, memory, etc.

1.6.2 6.2. Reddit Architecture 2

The next step in improving the speed of the site was to add another database. This was using a piece of software called Slony to replicate the database. There was still only the one app server at this time, and the caching was relatively limited, so they were making a lot of database requests.

The app server was hitting both of the database machines, and this was when they first ran into the problem of database lag. Slony was pretty good at replicating the database, but it was possible for the slave system to lag behind the master database by up to 5 seconds. Their solution was to improve their caching, but since they had two python applications running on the app server, and two databases they also faced the problem of keeping their caches in sync. They did this using a library called Spread. Spread is a network library which, if you send a message to one host, ensures that it is sent to all hosts.

Spread was also used to keep the hash tables in sync when they added an additional physical machine as a second app server. Obviously, this solution wouldn't scale very well since as the number of servers increased there would be a huge increase in network traffic as Spread attempted to keep them all in sync. The solution would be to use memcache, but this wouldn't happen until after the database was re-written.

Shortly after adding the second app server machine, Reddit added comments. The first version of comments was just another database table. It had a link-id, contents, author-id. Nothing very complicated. But the database still had a lot of joins. It was about this time that Reddit changed to a more flexible database architecture.

The challenge that they faced was that every time they added a new feature, they might have to add a new table, or they might need to a new column to an existing table. Adding columns would take time, and making the changes might require a data migration. or updating indexes all of which would add further load to the system and hit performance. The rate at which they could add features was limited by the rate at which they could make changes to their database and do these large migrations. In Steve's words, "It was a total pain!".

1.6.3 6.3. Thing Db

The flexibility that they were looking for in the database was provided by something called ThingDb. If you find any references to TDB or TDB2 in the Reddit code, this is what it is referring to.

Instead of having a table that looked like this:

Link:

URL	title	date	score

They would instead have a table, called a thing table, with just a few properties on it like this:

Link

score	author	date

These are the properties that everything has. Whether it's a link, a comment, or anything else. Then they had a separate table for each datatype that held just 3 things: the thing-id, a key and a value:

Data:

thing-id	key	value

So a link might have a row in the thing table like this:

1	5	0	06/01/05

and then several rows in the data table:

1	URL	...
1	title	…

There is therefore a row in the data table for every property of a thing that isn't common across all of the things. These are now different database tables, and they don't even have to be located on the same machine.

This allowed them to add features much more quickly and easily. If they wanted to add a new datatype to links, the new links would just have extra rows in the data table. Then they could just say, in effect, if the link doesn't have a value for this new property, we'll just pretend it has some specified default value.

So now they didn't have to get all their data structures exactly right way in advance. Later on, when they added sub-Reddits – which allowed users to make their own categories (their own "Reddits"), these were just another 'thing' which made the whole development process a lot simper.

1.6.4 6.4. Scaling

So Reddit were at the point where they had a couple of app servers, running their own caches, and a couple of databases that were replicas of each other. They had also added a load balancer to improve the performance of their app servers.

The next thing that they added was the memcached layer. Now, instead of the app servers having their own in-memory cache, they would communicate via memcache. Now there was just the one cache shared among all of their app servers. With 20/20 hindsight, Steve now says that they should have added memcache from the beginning.

To improve the databases, they began by segmenting by type. Thus there would be one database for just links and another for just comments. This means that if you are only submitting a link, or if you are only submitting a comment, you only have to access one database. In fact, this is essentially the structure that Reddit has to this day.

Steve didn't include sharding from the beginning. Although he had it in mind when he re-wrote Thing db, it was left out in the rush to get into production. The lesson that he has learned from the experience is that when you are developing a large system, if you don't do the hard parts up-front, you may never get the chance.

he framework that Reddit used, at least since it has been using Python, is called web.py. The Google App Engine framework, webapp2, inherited many of its features (including the handler class with distinct get() and post()

functions from web.py).

1.6.5 6.5. Pre-computed Caching

One of the other big architecture pieces that Reddit added to help them scale was the notion of a pre-compute cache. They would find themselves having to run the queries needed to generate the hot page for Reddit over-and-over again. They may have been able to cache it for a minute, but when that minute expired they had to recalculate it once again.

Alongside this were all the user pages. Every user had lists of the things that they had submitted and liked together with their top-things. Every subReddit had a new page and a hot page and so forth.

So, Reddit started to pre-compute everything. The way that they did this was to use a whole new database stack which were essentially replicas of the links database. These could lag behind the main database a little bit without causing a problem. When a vote came in, the app server would generate e series of tasks. These included recomputing the Reddit front page, recompute this user's likes page, recompute the user's top page and so forth. These tasks were held in a queue which fed a couple of machines called the "pre-compute servers".

The PC servers would take tasks from the queue and run the jobs against the new replica database. As a result, these databases ran really hot, although no real-time requests from the Internet ever touched these pre-compute machines. When the jobs completed, the result was stored in memcached. Now, almost every page you access on Reddit comes directly from the cache. There a very few things that you can do on Reddit that will directly manipulate the database.

Once they had reached that point, scaling became a lot easier. The main comments and links databases are now really the 'last-resort' primary data source. Any data you can access in real-time on Reddit is actually sourced from memcached. Every single listing is pre-computed and stored in memcached. This is also the reason that you can't go back beyond about 1000 links on any given listing. The upside is that it is now very fast.

1.6.6 6.6. Interview With Neil

Steve Huffman

Neil Williams, Lead Engineer, Reddit

SH: What does your job entail these days?

Basically, I first focus on fixing anything if it's broken. Otherwise I try to figure out what's the path for the site – what's becoming a bottleneck – and then try to fix that so that the site becomes fast again.

SH: How much traffic does Reddit get these days?

NW: Last month we got 2.7 billion page views. That's about a tenfold increase since 2009.

SH: What we are going to do in the interview is that I'll start of by describing how things worked at Reddit before I left, and then Neil will tell us what parts of the system are gone, and what they are doing preparing for the future because a lot of things have really changed. When a site grows, it grows by huge amounts and you have to keep adjusting.

SH: So, if I recall, when I left, the first piece of infrastructure that we had was the load balancer called HAProxy. Are you guys still using HAProxy?

NW: Yeah. It's fast, never breaks and works for us all the time.

SH: I remember the day I found HAProxy – it was probably in 2007 – and for a long time we had this super scalable architecture and just one HAProxy. We had one machine running one HAProxy, and that handled millions of requests per day. It was pretty incredible. Of course, outside of HAProxy is the Internet, and all of the Redditers – the users of the site.

They also used Akamai as their Content Delivery Network (CDN) for web application acceleration. These are basically third-party caches. Reddit pays Akamai to ping the Reddit site periodically, cache the content, and then deliver it to users who can then view the cached content.

NW: We use it for logged out users only, since everyone has their own personal page if they're logged in. Logged out users all see the same content.

SH: So the logged out users see Akamai, while the logged in users – who have custom content - their user name, their votes and all that stuff – come in via the internet and hit HAProxy directly.

1.6.7 6.7. App Server Architecture

SH: Behind HAProxy we actually get into the meat of Reddit's infrastructure. We had a bunch of app servers. These are physical machines that are running Python programs like the ones you have been working on in this class. When I left we had about 20 of these. How many do you guys have now?

NW: 180.

SH: Now the app servers are running Python. They're using a web framework called Pylons. Are you guys still using Pylons?

NW: Yes.

SH: This handled just about every request. Do you guys have special web servers for static content now?

NW: We use S3 (**Amazon Simple Storage Service**).

SH: Everything we do is on Amazon AWS. Basically you rent machines from Amazon, and one of the systems in AWS is S3. Maybe you could explain what S3 is and how it works?

NW: Sure. S3 is a simple storage service, and its basically a distributed file storage thing in "the cloud". Amazon lets you put objects into buckets, and it's literally just a key value store of these files, and other people can hit a URL and grab the object. We store all of our static content like CSS and JavaScript on S3. When you're hitting the site, you're actually going via Akamai to S3 instead of our infrastructure.

SH: So for static content, a user never even hits HAProxy, or the app servers, or anything at all.

NW: Yes, for the most part.

SH: As you grow there is no reason to waste the resources of your app servers handling connections for CSS, JavaScript and images.

NW: For a while we ere using nginX for the static content, and it got to the point where the one nginX server doing all of the static content couldn't handle it any more when we changed the content. Everybody's caches were invalidated.

So we would do a deploy of new code, and there would be this static content change, and al of a sudden that nginX would get overloaded, and everybody would be getting completely unstyled Reddit.

SH: So nginX is just a web server?

NW: Yes.

SH: Back in the early days of the Internet, almost all of the content online was static, and it was served by these things called web servers. These basically take HTTP requests, find the file that was in the URL, and then serve it. Over the last 15 years, online content has changed from being basically 100% static to almost 100% dynamic. Can you think of any websites that are 100% static?

NW: There has actually been a resurgence of static in the form of websites that get compiled from files, and so there are a lot of blogs out there now that are purely static served, but which are generated from files dynamically.

SH: We are going to talk about how much content Reddit pre-computes, and what you are talking about there is kind of an extreme form of that?

NW: Yeah. You only compile it when you need to. When you change it.

SH: Static content makes the whole thing easy to serve because it is the same content for everybody. When I left Reddit, even though it was pretty big, we could get away with having one web server holding static content in S3 or nginX, because Akamai handled most of the load.

-

1.6.8 6.8. Database Architecture

SH: Beyond the app servers we had our database. Now we just talked about how funky our database setup was, but it was still Postgres. We had a number of verticals. We stored links on one, votes on another, comments on a third and then we had a miscellaneous one.

NW: Subreddits were stored with comments. Users were stored with links.

SH: When we first built the system it was all on one machine, and we had different tables for each data type. A natural way to grow was to split the data types apart. I said in earlier units that scaling is a lot easier if you don't do joins. This is an example of that.

Do you know how many comments are in the comments database now?

NW: Something on the order of 250 million.

SH: All these databases were replicated. I think we used Londiste. Do you guys still use Londiste?

NW: Yes we do.

SH: Londiste is a third-party tool made by Skype.

NW: When you write to the master it hits a trigger, which the inserts the same query into a queue and that's replicated to all the slaves. The same inserts will run on all the slaves.

SH: So all of the database machines were replicated using Londiste, both for durability, and also if we lost the machine we wouldn't lose all our data, and also for load, which spread our database reads across all these machines. This is also where I found out about replication lag, and we had some really subtle bugs as a result.

1.6.9 6.9. Cache Architecture

Steve Huffman

SH: The way we got around the problem of replication lag was to add memcached. I think when we started out, memcached was on each app server, but as we started adding more and more app servers, the configuration got a little weird. So when I left we had a separate cluster of memcaches.

We used memcache at Reddit for everything. It's like the Swiss army knife of systems. We definitely used it to avoid replication lag – we'd write to the database and to memcached at the same time. so that when you do an immediate redirect - to a permalink page or something like that – the data is good to go.

The other big system that we had was the pre-compute system. Doing real-time queries against the database was too slow to generate our listing pages. So we had a whole separate database cluster called the pre-compute cluster, which was just mirrored versions of the main database servers.

When you submitted a link to Reddit, or voted on a link, we would submit a job the a queue. The job would be something like 'update the front page' or 'update this person's Liked page'. Then what the queue would do is to run the query to recompute that listing using the pre-compute cluster.

We also had a separate set of memcaches that were running a technology called memcache-db. Memcache-db is just like memcache except that instead of being totally in memory, it had a little disk data-store. This made it kind of fast, like memcache, because it was mostly in memory, but it would also persist. Data that got recomputed would be stored in memcache-db as kind of a middle-layer cache. This is all to avoid running queries on the main databases.

That was basically the state of the system when I left. Lots of replicated databases, the really complicated pre-compute cluster and memcache.

1.6.10 6.10. Problems With Memcachedb

Steve Huffman

Neil Williams, Lead Engineer, Reddit

SH: So, which of these pieces don't exist anymore?

NW: Well, after you left, but before I started, memcache-db hit a scaling wall. It would not go any further. The writes were just too fast for it.

SH: Memcache-db isn't really designed for the kinds of heavy loads we were throwing at it. It is basically the memcache code bolted on top of the Berkeley db. So you guys got rid of this?

NW: That's right, and we replaced it with Cassandra which is a distributed NoSQL database. The way it works is that you have rows, and a row is sharded by its key to somewhere in the ring of servers, so you get automatic sharding across the entire ring. When a row has columns inside it and that is where the actual data is stored. The way that Cassandra operated was pretty similar to memcache-db.

SH: Remember how we talked about databases and how you can replicate them, which is where you send the same data to multiple machines which then helps with distributing the load and with durability, or you can shard them, which is where you send a chunk of the data to one machine and another chunk of data to another.

NW: Cassandra is configurable, and in our case we are using a replication factor of 3. This means that if a piece of data exists on one node, then it also exits on the two adjacent nodes. This means that a read can be serviced from any one of the three nodes (if we allow it to be), and it also means that if we lose any one node we won't lose all our data for that segment of data.

SH: If you lose a node, is the content redistributed?

NW: No. They are assigned a key space and you have to move tokens if you want to rebalance the ring.

SH: Is that something that you, as a developer, would have to do?

NW: Yes. That's an operational thing.

SH: One of the things that happens as things grow is that people's roles change. When you are a small website, you are the designer, developer and sysadmin, all in one. Reddit now has a team of operations guys?

NW: Yes. A team of two.

1.6.11 6.11. Locking And Memcache

SH: You replaced memcache-db with Cassandra for the pre-compute stuff?

NW: Yes, and that solved the write-contention issue. Say you have a listing which is a set of IDs of all of the links in a subReddit, and you have to modify this listing. To do this you have to lock around the item at the subReddit level. If you have a lot of people hanging out in one subReddit, that particular subReddit will suffer from a lot of locking.

SH: In Office Hours 4, someone asked about what happens in Datastore if two users tried to register the same username at the same time (Datastore doesn't enforce any uniqueness constraints on a field in the database). My answer was that you can either use transactions in the Datastore (which I don't fully understand), or you can use memcache since Memcache has this global lock.

So you guys are moving away from that?

NW: Yes. When you use memcache as a locking service, the problem is that, if you lose a single memcache node, the you lose the site! For example, say half the apps can't see a memcache node, and the others can. The ones that don't see it decide that they aren't going to talk to that guy any more, so they try to lock on a different set of servers.

SH: When we store data on multiple nodes in Cassandra, it is different from how we store data it is different from how we store data across different nodes of memcache. When you distribute cross memcache, you basically just hash your key to particular memcache box, and there is no notion of replication.

NW: It's similar in Cassandra. Each node has a key space, and it also goes to +1 and to -1. In fact you can do replication in memcache like that as well.

SH: When I left, we were using a naïve memcache library that basically took a key and said "which box is this hashed to", and then stored it on that box. If you lost that box, it would effect the hashing of every other key. Losing a memcache box was really painful! So, we'd replicate memcaches for mostly space – if we had more space, things wouldn't expire out of the cache too fast – but that had a big downside that if we lost a single memcache box, all of the keys would get rearranged.

NW: Right. You were using Modulo Hashing. We are using Consistent Hashing now, and the way that works is it builds up a ring of nodes, but instead of mapping batches of keys to specific nodes, it assigns them to a place on a circle, and finds the nearest server to that point on the circle. If that node fails, it will just go to the next nearest. We have 10 nodes. Using this technique, if we lose one node only 1/10 of the keys get redistributed.

SH: Unlike when we were using Modulo hashing, which is a really naïve way to distribute keys. If we suddenly went from modulo 10 to modulo 9 because we had lost a node (or more often because we had misconfigured a node, and the app servers couldn't se it any more), instead of losing 1/10 of our keys, we'd lose 9/10 of our keys. And that's a big problem! Suddenly the cache isn't warm any more and you're databases will take a pounding.

1.6.12 6.12. Zookeeper

SH: So you are changing the way that you're doing locking?

NW: For the same reason, a single node failing means that we lose the entire site because of locking. So we're moving locking into something called Zookeeper, which implements a tree system with a guaranteed order of operations on those, and with that you can build locking. The main advantage of Zookeeper is that it has much higher availabilities, so if we lose a Zookeeper node, we should be able to come back within 200 milliseconds. As opposed to whenever someone notices that the cache is down and replaces it.

SH: And Zookeeper is a separate system?

NW: Yes. It's an Apache project, like Cassandra.

SH: Is there one Zookeeper box, or a bunch of Zookeeper boxes..?

NW: They form an ensemble. The way it works is that there is a master and the others are read-replicants, and if the master fails it automatically elects a new leader from the ensemble.

SH: Do all the requests go to one machine?

NW: No. You write to any of them, so your client could reconnect to any one of them, and they automatically figure out who to talk to.

SH: Is Zookeeper just for locking, or does it do other stuff?

NW: We are also going to be moving dynamic configuration type stuff into Zookeeper, and Zookeeper provides watches on the nodes so that the apps can actually get notification when something changes in Zookeeper. Say we're going to change the ad on the front page. We just set something in Zookeeper, and all the apps update themselves.

SH: That reminds me of one thing that was always a challenge. We had all these app servers. The app servers had all the configuration. A lot of the configuration was basically part of the code. One example would be how many memcache boxes you have, because the memcache library would exist on the app servers, and that's where the hashing would happen. If you're deploying a new configuration, once you have more than a couple of app servers, the deployment takes time. So, suddenly you might have half your app servers with a different configuration, and that's when things can get weird. Have you guys improved that at all?

NW: No. That will be Zookeeper. Zookeeper can hold information about what databases there are and so on. The reason that you wouldn't want to just do that in memcache, for example, is that you would need to fetch that information for every request because there is no other way to know if it has changed.

1.6.13 6.13. Improving Memcache

SH: One other thing that you mentioned that you are improving is this notion of memcache ejection.

NW: Yes. The memcache client library has the ability to notice that a memcache node has gone down, and decide that it's not going to try and talk to it any more. It's kind of a heuristic. It notices that a number of failures happen, so it just decides to back off and it will then treat the ring as if it had one fewer node. We cannot use that until the locking has gone, but once we've done that, memcache will automatically heal itself.

In fact, we could probably do with more memcache right now, but we don't want to add more because it increases our risk of failure.

SH: Yes, that's something that I remember too. Sometimes we wouldn't add a memcache because we didn't want to redistribute the keys. Just the simple act of redistributing the keys was this really scary thought.

NW: Yes, and right now, even with consistent hashing, if we add one memcache, in the middle of the night, the database slaves will be quite unhappy for maybe an hour or two, and that is just one server.

SH: Something I used to do (this is a kind of hacky thing – I can feel people losing respect for me even as I describe it) is, whenever we used to bring up a new database slave, I would actually go into the database and I would have an app server that would connect only to that machine, and I would hit all the most popular pages by hand to load all those queries and make sure that the cache was all up to date and warm and that the database was good. This was because, when you bring up a new database slave, it hasn't run any queries yet, so nothing has been cached.

One of the things that Postgres does really well is that it manages the disk-memory dichotomy where all of the data is on disk, but only some of that data will fit into memory, so you need to tell Postgres this is the data I want in memory now which is why we used to run these queries.

NW: Yes. And heating it up is also great because you don't have to worry about the piling-on effect or cache stampede.

1.6.14 6.14. Pre-Compute Architecture

SH: Now, my understanding is that the pre-compute servers have also gone.

NW: In the process of moving to Cassandra a lot more of the cached files were mutated in place. Say you remove something from your saved links, so it has to be taken out of your saved page. What happens is that it fetches the pre-computed version of your saved page from Cassandra, removes the item you no longer want, and then puts it back into Cassandra. It has to lock around that, but the big advantage there is that it doesn't have to re-run the entire query, because it's only making one little change.

SH: So instead of sending the complex query to a machine that is already doing lots of complex queries, you just update the cache directly?

NW: Yes. There are a few types of queries that this is not possible with. Those are things like your top links of this hour. The top links for the hour and stuff like that is recomputed en masse for the entire site using Mapreduce. What that does is that it will dump every link that was submitted in the last hour, groups them and figures where they should go and then just completely overwrites these listings every 10-15 minutes.

SH: My first question is, where is that data that you are mapping over stored?

NW: It's coming out of Postgres.

SH: So you have a job that takes that data out of Postgres and then runs the big job on it.

NW: Right.

SH: Is that from the main database?

NW: We have a dedicated slave for links.

SH: So there's another little replica that is only read for dumps, and then you run Mapreduce jobs on this every so often, and the results get stored in Cassandra?

NW: Yes.

SH: What about comments and votes?

NW: There're no actual queries that need that.

1.6.15 6.15. Mapreduce

Steve Huffman

Mapreduce is basically this programming technique for doing batch jobs across huge amounts of data. There are two functions if you are programming functionally: map and reduce. Map basically says, given this list of things, apply this function to it. Reduce basically says, given these two things apply this function to them and combine them into one thing.

Google made Mapreduce famous when they built their indices for the web. Essentially, they take all the web-pages on the Internet and you can then apply this function to them, which is basically "find the words in this document". And then you can reduce on that to basically reduce the content down to the kind of target output that you want.

1.6.16 6.16. Hadoop

Steve Huffman

Neil Williams, Lead Engineer, Reddit

SH: After I left, someone wrote a Mapreduce program by hand, and you guys are now replacing that?

NW: We're switching over to Hadoop using a language called Pig, or Pig Latin. Hadoop is a Mapreduce system. It has the advantage of handling all the details of distributing these mappers and reducers across a cluster of nodes, and we're actually using Amazon's Elastic Mapreduce, which has hosted Hadoop to make our lives simpler.

SH: We talked previously about AWS – the Amazon Web Services, and Amazon has built in, not just machines, but these whole Hadoop clusters that you can commission and pay for. So you guys are moving in that direction. We are also using this at Hipmunk now too for our log analysis.

It's really cool because you can basically just put in queries and say, "here's my data" (in fact we store our data in S3 in big text files) and then run the query using (say) 20 machines. Effectively, you can say how long you want to wait and Amazon will bring up all the machines, load up your data, run the job, and then shut all the machines down.

You guys are going to be running Mapreduce constantly though.

NW: That's right. How to do it with EMR is an interesting question. It's clear we can do it, we just need to set it up right.

SH: Is that going to compute the hot pages for every Reddit?

NW: It's mostly for user pages, but there are a few other things like your top links of this hour.

1.6.17 6.17. Dealing With Search Indexing

SH: One interesting thing that we learned at Reddit over the years was the issues that arise from old data.

In the beginning it had just a small amount of content, but as we grew the content turned into two flavours. Not just hot versus saved, but a kind of new versus old. If you go to your profile page, that content isn't accessed very often. Before we did the pre-computing thing, the hot pages were highly optimised. The queries were fast. Everything worked nicely. It was cached perfectly. Then somebody would hit a user page that hadn't been accessed in a month, and all of a sudden it's stale, cold data. It's not in the cache and the whole system chugs just to bring that out. That's when we started pre-computing and now you guys are taking it to the next level.

NW: The comments database is the really bad one. There are just so many comments, and so much data. These databases get overloaded. If you pull up an old comment page from two years ago that nobody has seen in more than three months, and which has to load 500 comments involving a disk hit for every single one, this can really hurt the system. We have a dedicated comment slave that is just for the likes of Google who pull these up every so often.

SH: There's a whole separate stack for Google?

NW: Yes, and it's read-only.

SH: There are two or three app servers just for Google, because Google will come through and index everything. Reddit gets a lot of traffic from Google. You can take any two word combination from the front page of Reddit, do a Google search for it, and it will be in the top 10. That's because Google is just murdering Reddit. So they had to build a whole separate infrastructure just for Google, because they hit the site in a way that is really hard to cache for and which is completely from how users access Reddit.

1.6.18 6.18. Using The Queue

SH: I think we've covered most of the big things. Is there anything else that you guys are working on now?

NW: We still have the queue system, and a lot of the actions – like doing a vote – when you hit the API in-point, all it does is to insert an item into the queue. It doesn't really do any work on it, and this happens in the background. If one of the databases is going slow, and it's taking too long to write, it doesn't effect the user. We use that for a lot of stuff now, and we're trying to move further towards that.

SH: The queuing architecture is nice. You are using AMQP?

NW: Yes.

SH: We were using RabbitMQ when I left. I remember seeing AMQP and thinking "this is a really nice architecture". I'm glad that that's working out.

1.6.19 6.19. Lock Contention

SH: Anything else?

NW: Lock contention is a big issue for us right now. As I was saying with this Cassandra stuff, whenever you vote on a link in a popular subReddit, it has to lock on that whole subReddit's listing. We had an example of that biting us recently.

We have these bulk queue processors. The app servers write what has happened to the queue, and the queue processor takes an item from the queue, perhaps saying that someone's cast a vote, records the vote in Postgres and Cassandra, and then updates all the listings effected by that vote. This is the real cause of a lot of the locking in Cassandra right now.

We have a lot of these queue processors for processing votes. We get a lot of votes simultaneously, so we need a lot of queue processors. But it turned out that we had too many, and they were all fighting each other for those locks. Just halving the number of those queue processors actually sped up queue processing in general.

SH: So you actually need enough queue processors to handle the depth of the queue, but if you have too many, they spend too much time fighting each other.

NW: Exactly. One of the ways we are working on that is we're getting rid of the locking in Cassandra, and trying to get rid of locks as far as possible in general.

SH: Locking is a common theme. Python is multi-threading – running two pieces of your program at the same time. It's multi-threading is far from state-of-the-art. When Reddit first switched to Amazon they had a problem where Python was spending so much time locking it's data structures so the two threads could access the same data structure at the same time, that it actually slowed the computer's ability to read traffic over the network. They solved this by making Python single-threaded.

NW: We very rarely use threaded processes. The ad servers use threaded processes, but other than that we just have lots and lots of separate processes letting the OS handle the task switching. Linux is pretty good at that.

SH: This has been really great. Thank you so much for coming.

One thing I'd like to point out is that everything that we've talked about here, at least all the main stuff is all open-source software. It's amazing how far you can get without paying for anything. The vast majority of Reddit's code is also open-source and online, so if you want to look at this stuff, just go to:

https://github.com/reddit

If you do look at the Reddit code, you will see a lot of the code we developed in this class, for hashing and passwords and so on, in there somewhere.

1.6.20 6.20. Growing Reddit

Steve Huffman

You now have a pretty good idea about the architecture decisions that we made on Reddit, how we grew the site and how we changed the way we stored data over time. Now I'd like to talk about how we actually grew the site from a social point-of-view. How we got users, and how we maintained the feel that Reddit has.

It's hard for any one person to take the credit for growing Reddit, because it's a social thing. If we started over today, even knowing everything we know now, it would be very hard to get a site like Reddit off the ground. Social networking sites need a little bit of luck to succeed. We were lucky enough to have it. We also did a lot of important things that I think helped.

In the beginning, Alexis and I submitted all the content. If you go to the Reddit submission page now, it basically boils down to just two fields. In the early days, it *was* just two fields. You had a field called “url” which held the URL of the site you were submitting and “title” which was the title of the site, and which would be the link you submitted to Reddit. If Alexis or I went to the site, we had an additional field called “user”. We could type in a name here, and when we hit submit it would automatically register the user if the name didn’t already exist and submit under that name.

For the first couple of months there was a lot of content, but almost all of it was submitted by us. The content was genuine, but the users were fake. This did two things. Firstly, it set the tone. We submitted content that we would be interested in seeing. This meant that the content on Reddit (at least for us and our peer group) was all good, interesting stuff.

The other thing it did was to make the site feel ‘alive’. Users like to feel a part of something. If they turned up and the front page was blank it just looks like a ‘ghost town’. A few months in, we had the great experience of not having to submit content. The site was working on its own.

We also didn’t have comments at the start. We just didn’t get around to writing it. If you look at Reddit today, I think you’ll agree that the comments are what makes Reddit work. But Reddit grew quite a bit even before we had comments. Sometimes it’s better just to get something online and start experimenting with it rather than to build the whole system and then have it fail.

We didn’t have categories. What we wanted was for users to submit links. We didn’t want them to have to think about categories or tags. Actually, we did have categorisation on the site – for about one day! I added it one night, and Alexis took all the links and categorised them. One of our first investors and advisors said “why did you add categorisation? That ruined everything! Get rid of it”. So we did. That was really good advice.

Another thing was that we didn’t collect emails. Every other site had a sign-up process that required email confirmation. If you want members, and you want submissions, why would you erect the barriers? We also had no censorship. We didn’t care what content was submitted to Reddit unless it was overtly racist. We just let it be. Especially if it was critical of us. This created a community spirit and sense of trust I believe that this is what allowed Reddit to grow to such an extraordinary size. We just wanted to make a site that was new and interesting. We didn’t want to sell our users. We didn’t want to ruin the site.

We just wanted to have good content.

1.6.21 6.21. Spam Prevention

Steve Huffman

One other thing that we spent almost no time on when we started up, and a lot of time on now, is spam prevention.

You’d think that on a site like Reddit, where it’s like the Wild West, with no rules, and where anybody can submit anything it would quickly be overrun by spam. Actually, you’d be surprised. Now that Reddit is really big, yes it receives lots of spam, but if you understand what motivates spammers it can re relatively easy to prevent.

A spammer basically wants links. An easy way of preventing their links having any value is to add an attribute to your anchor tags that looks like this:

- <a rel=”nofollow” href=…

If you have a link with this extra attribute it basically tells Google that the link shouldn’t be followed for search quality purposes. The way that Google works is to look at a given page, and how many pages elsewhere link into that page gives that page some authority. This attribute says that I don’t want to give this link any authority.

Every link on Reddit has rel=nofollow until it has a certain number of points. This is because we didn’t want to put rel=nofollow on all links because we like the way the internet works, and we figured that if the link was good on Reddit then that link *should* have some authority.

Another thing about spam is that it is rarely submitted by an individual by hand. They almost always use automated tools. Looking at the behaviours of the bots that were submitting content to Reddit, they always had these behaviour ‘gaps’ that humans never do. For example, a bunch of users will all have the same password.

Another thing that spammers did was that they submitted a link and then commented on their own link. Real users almost never did this. So we just marked links with one comment from the author of the link as spam. Just these odd behaviours were enough for us to identify most of the spam. Our behaviour heuristics worked really, really well.

The most important thing that Reddit did, and still does, it don't let them know when they've been caught. When we ban a submission on Reddit, that user would still see their submission on our hot page, or on their user page. If their vote didn't count, it still looked as though it did. Sometimes, security through obscurity works really, really well.

1.7 7. Next Steps

Steve Huffman

Over this course, you have learned the major pieces of web applications and how they fit together. You know about application servers, databases, cookies, hashes and how HTTP works. With that knowledge you can build quite a few things, but there are still a few concepts that we haven't really talked about that, if you are working on your own, it would be good to get good at. A lot of it revolves around front-end technologies.

- CSS - to control how your HML appears
- JavaScript – a programming language that runs in the browser.
    - ♦ can manipulate your HTML or make further requests to the server
- AJAX – Asynchronous JavaScript
    - ♦ making HTTP requests in the background to update the pages dynamically.

These would all be good technologies to learn.

The last thing that we're going to do in this unit is to talk to Udacity about how they use Google App Engine in production, so you can get a sense of the difficulties that they've encountered, and some of the solutions that they've found.

1.8 8. Interview With Chris

Steve Huffman

Chris Chew, Engineer, Udacity

SH: We are talking with Chris Chew, who is one of the Udacity engineers and has been working on App Engine, and we're going to talk a little bit about how Udacity uses App Engine in production. We've been writing these simple 'toy' applications, while Udacity actually has thousands of users, so they are solving a very different problem.

So, Chris, how has your experience with App Engine been so far?

CC: Actually, it's been interesting. I think a lot of people, especially the more engineering experience you have, tend to be very sceptical about something that makes claims as big as App Engine. Everything always involves trade-offs, and App Engine is no exception.

I really did come to Udacity thinking that I would be able to talk everybody into moving to something different where we would have a little bit more control. In fact, I think the opposite has actually happened. I've been converted the other way, and I think that App Engine might be the perfect spot for Udacity. Udacity does run on App Engine, and we do have a lot of traffic, in fact there are even larger websites that have even more traffic running successfully on App Engine.

SH: What are some of the things that App Engine does that make it so nice to work in?

CC: The big thing is that they provide a lot of infrastructure. It always boils down to that. Caching, highly replicated datastores. They force you to address what consistency means in your application. Does a name have to be updated everywhere immediately, or can things be a little bit slower to update? These are the sorts of things you don't have to think about when you build a simple app, and then, all of a sudden you get really big and have to scramble to redefine how these sorts of things are going to work in your system. The queuing is actually really powerful. It was really smooth and seems very reliable. This all lets you build tasks, and focus on those tasks rather than the infrastructure needed to deliver the tasks.

By and large, the logging infrastructure works really well. Especially considering the scale of the logging statements you can put through.

SH: Can you talk about how the version handling works in App Engine and how that fits into the bigger picture?

CC: Versions are actually really interesting in App Engine. They're different from what I'd expected. They actually allow you to upload a totally different code base, but using the same datastore and the same services. You can use that in a couple of different ways. One thing is that it enables what we call A-B testing where two versions of a site are running at the same time, and there's actually in A-B testing some administration there that allows you to blend the percentage of users in a somewhat sticky manner. This allows you to roll out new features to some users and test them, and if they're successful, migrate all the users to the new features.

Another thing it that they provide you with ways to have 'aspects' of a system. You could have a version of an app that is mainly for administrative tasks, another for testing or for monitoring, and then the version that you users would normally see. And they'd all be running simultaneously against the same datastore and services. This allows you to 'partition' your logic letting you re-use models and things between them

1.8.1 8.1. Using App Engine At Scale

SH: A lot of the things that are needed to scale an application, like adding new app servers or memcaches, or scaling the database are taken care of by App Engine for you. Have you guys been taking advantage of these features? Has that worked out for you in production?

CC: Absolutely! The auto-scaling is probably the best feature of App Engine. It works really smoothly. It's very transparent. Our traffic definitely has peaks and troughs. There's been a lot of press, and that brings a lot of traffic. Course launches bring a lot of traffic. Homework deadlines generate a lot of traffic! We can handle the extra traffic without thinking about it.

I've heard some stories from the early classes, and from the early AI prototype course, and they were pretty much spinning up servers non-stop. We don't have to do that. We don't have to think about it. We just check each day to see how many instances we have running.

There was a time when we accidently DDoS'd ourselves with an AJAX bug that was sending way too many requests back to the server. When we spun up to a couple of hundred instances everything was fine, our users were still able to continue. Every site that's learning how to scale has made that mistake, but most of those other sites took themselves down for the day while they figured out what was going on, but we were able to figure out what was going on without going down.

SH: Can you comment at all on how App Engine was designed, and how it fits with the way you've done things in the past?

CC: It has become very clear to me in the time that I have been working with it, that it was designed by people who had been through the process of scaling many times and with a range of different kinds of apps, because the solutions and the things that they provide have those characteristics. If you were building something over again, this is how you would do it. For example, queuing can get really complex. In their queuing system, you create a task and you give it some properties. The kind of properties that they give you, and which you can tune, are exactly the properties that you'd want to tune. The trigger is just a simple web hook callback, so your worker logic is just like any other request handler.

1.8.2 8.2. Problems Scaling App Engine

SH: Not everything in App Engine is perfect. Maybe we could talk about some of the things that are more difficult in App Engine?

CC: In App Engine, the 'Happy Path' is paved with concrete, but it has chains along the sides. so you really can't leave that 'Happy Path'. This is something that you do have to keep in mind.

There are also failures from time to time. A query can work just fine and then the next time you make exactly the same call, it doesn't work. These are just things that happen, and you have to deal with them. This means that you'll often do a sort of 'query loop' where you run the same query three times, knowing that within those three attempts you'll probably get a result. You just re-perform the same query before you let your request handler send an error response back to your user. This is actually quite good practice, because with distributed computing you're going to get failures. You need to program defensively to manage these 'system' failures. Being ready to issue a retry and put that into your normal call stack or framework is the way forward if you want your app to run smoothly. And those are the sorts of things that Udacity is dealing with, or learning about, right now.

You always have to be cognizant of how long a long-running back-end task is going to take. I think the limit is 10 minutes, which seems like a long time, but on a highly virtualised system, you can actually have long-running tasks that take longer than the 10 minute limit. You have to be prepared to shut these tasks off early and be ready to pick them back up with a subsequent request.

Another thing is that, when you get a larger Python system, you want to have C modules pre-compiled. You don't get that with App Engine. You have to forget about those sorts of things.

The biggest issue, and one which I know that Google is addressing, is better SSL certificate support, especially for the custom domain. Google has a beta program running and I hope it happens soon. You get SSL on domain.appspot.com, but if you point a custom domain at it, you don't get SSL support for that – which is a big drawback.

SH: One thing that I was sceptical about was the database, and how much access you have to your own data. Are there any limitations there that you need to think about?

CC: Definitely. It's something that's probably worthy of a whole unit in a course about going deeper with App Engine. There are a few trade-offs that you have to come to terms with. Things like all queries have a limit of 1000 records. And that is before paging, which kind-of sucks! You could potentially pull out a result set that was 100 empty records and then have to page to the next one to get the 3 records that match, depending on how the indexing or your model is structured.

Backup and restore is really awkward. There are some backup and restore utilities built in, but not if you just wanted to take the data each night and sync it to a testing or QA version of your app. There are some tools out there that basically allow you to export the data as Python code which you could then run and import into another app.

On the one hand, it's nice to have transparent costs. We know pretty much exactly how much every one of our queries costs, which is great! The downside is that when you're doing something like a backup, in the back of your mind you're thinking "man, this is costing a lot of money!". So it's things that you can get around, but not as smoothly as just taking a hot backup.

The other thing is that there's not really great data viewer. There are times when you're troubleshooting and you really don't know what's going on. You want to issue some SQL queries directly against the data, to see what the data looks like. There is a data viewer on the dashboard, but it's not that great, it only works with certain kinds of data types, and you can't really update anything. So if you just need to flip a field to change the status of something which would allow you to keep processing while you work on fixing the bug in the background, you can't. Which is frustrating.

On the other hand, that might force you to build some better administrative features. If you can fix things easily with some duct tape, you end up not actually building a good administrative interface. With App Engine, sometimes the only way that you can do it is to build the administrative interface.

SH: So some of the limitations seem to be not without reason, but they are still limitations.

CC: Yeah!

SH: Thank you for doing this with us. We've learned a lot about how App Engine works and how to scale in the App Engine environment.