Real World Web: Performance & Scalability

Ask Bjørn Hansen
Develooper LLC

http://develooper.com/talks/
Hello.

• I’m Ask Bjørn Hansen
  perl.org, ~10 years of mod_perl
  app development, mysql and scalability consulting
  YellowBot

• I hate tutorials!

• Let’s do 3 hours of 5 minute° lightning talks!

° Actual number of minutes may vary
Construction Ahead!

- Conflicting advice ahead
- Not everything here is applicable to everything
- Ways to “think scalable” rather than be-all-end-all solutions
- Don’t prematurely optimize! (just don’t be too stupid with the “we’ll fix it later” stuff)
Questions ...

• How many ...
• ... are using PHP? Python? Python? Java? Ruby? C?
• 3.23? 4.0? 4.1? 5.0? 5.1? 6.x?
• MyISAM? InnoDB? Other?
• Are primarily “programmers” vs “DBAs”
• Replication? Cluster? Partitioning?
• Enterprise? Community?
• PostgreSQL? Oracle? SQL Server? Other?
Seen this talk before?

- No, you haven’t.
- :-) 
- ~266 people * 3 hours = half a work year!
• Do we have time for questions?
• Yes! (probably)
• Quick questions anytime
• Long questions after
• or on the list!
• (answer to anything is likely “it depends” or “let’s talk about it after / send me an email”)

---

Slides per minute

<table>
<thead>
<tr>
<th>Year</th>
<th>Slides per minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>0.25</td>
</tr>
<tr>
<td>2002</td>
<td>0.25</td>
</tr>
<tr>
<td>2004</td>
<td>1.00</td>
</tr>
<tr>
<td>2005</td>
<td>1.75</td>
</tr>
<tr>
<td>2006</td>
<td>1.00</td>
</tr>
<tr>
<td>2007</td>
<td>0.25</td>
</tr>
<tr>
<td>2008</td>
<td>0.25</td>
</tr>
</tbody>
</table>
• The first, last and only lesson:

• Think Horizontal!

• Everything in your architecture, not just the front end web servers

• Micro optimizations and other implementation details — Bzzzzt! Boring!

(blah blah blah, we’ll get to the cool stuff in a moment!)
Benchmarking techniques

- Scalability isn't the same as processing time
  - Not “how fast” but “how many”
- Test “force”, not speed. Think amps, not voltage
- Test *scalability*, not just “performance”
- Use a realistic load
  - Test with "slow clients"
- Testing “how fast” is ok when optimizing implementation details (code snippets, sql queries, server settings)
Vertical scaling

- “Get a bigger server”
- “Use faster CPUs”
- Can only help so much (with bad scale/$ value)
- A server twice as fast is more than twice as expensive
- Super computers are horizontally scaled!
Horizontal scaling

• “Just add another box” (or another thousand or ...)

• Good to great ...

  • **Implementation**, scale your system **a few** times

  • **Architecture**, scale dozens or **hundreds** of times

• Get the big picture right first, do micro optimizations later
Scalable Application Servers

Don’t paint yourself into a corner from the start
Run Many of Them

- Avoid having *The Server* for anything
- Everything should (be able to) run on any number of boxes
- Don’t replace a server, add a server
- Support boxes with different capacities
Stateless vs Stateful

• “Shared Nothing”

• Don’t keep state within the application server
  (or at least be Really Careful)

• Do you use PHP, mod_perl, mod_...
  • Anything that’s more than one process
  • You get that for free! (usually)
Sessions

“The key to be stateless”
or
“What goes where”
No Local Storage

• Ever! Not even as a quick hack.

• Storing session (or other state information) “on the server” doesn’t work.

• “But my load balancer can do ‘sticky sessions’”

• Uneven scaling – waste of resources (and unreliable, too!)

• The web isn’t “session based”, it’s one short request after another – deal with it
Cookie: session_id=12345

Evil Session

Web/application server with local Session store

12345 => {
  user => {
    username => 'joe',
    email => 'joe@example.com',
    id => 987,
  },
  shopping_cart => { ... },
  last_viewed_items => { ... },
  background_color => 'blue',
},
12346 => { ... },
....

What's wrong with this?
Evil Session

Cookie: session_id=12345

Web/application server with local Session store

Saving state on one server!

12345 => {
  user => {
    username => 'joe',
    email => 'joe@example.com',
    id => 987,
  },
  shopping_cart => {
  },
  last_viewed_items => {
  },
  background_color => 'blue',
},
12346 => {
},
....

Easy to guess cookie id

Duplicate data from a DB table

Big blob of junk!

What’s wrong with this?
Good Session!

Database(s)

Users

987 =>

{ username => 'joe',
  email => 'joe@example.com',
},
...

Shopping Carts
...

Web/application server

Cookie: sid=seh568fzkj5k09z;
user=987-65abc;
bg_color=blue;
cart=...;

memcached cache

seh568fzkj5k09z =>

{ last_viewed_items => {...},
  ... other "junk" ...
},
...

• Stateless web server!
• Important data in database
• Individual expiration on session objects
• Small data items in cookies
Safe cookies

• Worried about manipulated cookies?
• Use checksums and timestamps to validate
  • $cookie=1/value/1123157440/ABCD1234$
  • $cookie=${$cookie_format_version}$value/${timestamp}$checksum$
  • function cookie_checksum {
      md5_hex( $secret + $time + value );
  }
Safe cookies

• Want fewer cookies? Combine them:
  
  - cookie=1/user::987/cart::943/ts::1123.../EFGH9876
  
  - cookie=$cookie_format_version
    /$key::$value[/$key::$value]
    /ts::$timestamp
    /$md5

• Encrypt cookies if you must (rarely worth the trouble and CPU cycles)
I did everything – it’s still slow!

• Optimizations and good micro-practices are necessary, of course
• But don’t confuse what is what!
• Know when you are optimizing
• Know when you need to step back and rethink “the big picture”
Caching

How to not do all that work again and again and again and again...
Cache hit-ratios

- Start with things you hit all the time
- Look at web server and database logs
- Don’t cache if you’ll need more effort writing to the cache than you save
- Do cache if it’ll help you when that one single page gets a million hits in a few hours (one out of two hundred thousand pages on the digg frontpage)
- Measure! Don’t assume – check!
Generate Static Pages

• Ultimate Performance: Make all pages static

• Generate them from templates nightly or when updated

• Doesn’t work well if you have millions of pages or page variations

• Temporarily make a page static if the servers are crumbling from one particular page being busy

• Generate your front page as a static file every N minutes
Cache **full** pages
(or responses if it's an API)

- Cache full output **in the application**
- Include cookies etc. in the “cache key”
- Fine tuned application level control
- The most flexible
  - “use cache when this, not when that”
    (anonymous users get cached page, registered users get a generated page)
  - Use regular expressions to insert customized content into the cached page
Cache full pages 2

- Front end cache (*Squid, Varnish, mod_cache*) stores generated content

- Set Expires/Cache-Control header to control cache times

- **or** Rewrite rule to generate page if the cached file doesn’t exist (this is what Rails does or did...) — only scales to one server

  - `RewriteCond %{REQUEST_FILENAME} !-s
    RewriteCond %{REQUEST_FILENAME}/index.html !-s
    RewriteRule (^/.*$) /dynamic_handler/$1 [PT]`

- Still doesn’t work for dynamic content per user (**”6 items in your cart”**)

- Works for caching “dynamic” images ... on one server
Cache **partial** pages

- Pre-generate static page “snippets”  
  (this is what my.yahoo.com does or used to do...)
- Have the handler just assemble pieces ready to go
- Cache little page snippets (say the sidebar)
- Be careful, easy to spend more time managing the cache snippets than you save!
- “Regexp” dynamic content into an otherwise cached page
Cache data

• Cache data that’s slow to query, fetch or calculate
• Generate page from the cached data
• Use the same data to generate API responses!
• Moves load to cache servers
  • (For better or worse)
• Good for slow data used across many pages
  ("today's bestsellers in $category")
Caching Tools

Where to put the cache data ...
A couple of bad ideas

*Don’t do this!*

- Process memory ($\text{cache}\{\text{foo}\}$)
  - Not shared!
- Shared memory? Local file system?
  - Limited to one machine (likewise for a file system cache)
  - Some implementations are really fast
- MySQL query cache
  - Flushed on each update
  - Nice if it helps; don’t depend on it
MySQL cache table

- Write into one or more cache tables
- id is the “cache key”
- type is the “namespace”
- metadata for things like headers for cached http responses
- purge_key to make it easier to delete data from the cache

```
CREATE TABLE `combust_cache` (
  `id` varchar(64) NOT NULL,
  `type` varchar(20) NOT NULL default '',
  `created` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
  `purge_key` varchar(16) default NULL,
  `data` mediumblob NOT NULL,
  `metadata` mediumblob,
  `serialized` tinyint(1) NOT NULL default '0',
  `expire` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY (`id`,`type`),
  KEY `expire_idx` (`expire`),
  KEY `purge_idx` (`purge_key`) ) ENGINE=InnoDB
```
MySQL Cache Fails

- Scaling and availability issues
  - How do you load balance?
  - How do you deal with a cache box going away?
- Partition the cache to spread the write load
- Use Spread to write to the cache and distribute configuration
- General theme: Don’t write directly to the DB
MySQL Cache Scales

- Persistence
- Most of the usual “scale the database” tricks apply
- Partitioning
- Master-Master replication for availability
- .... more on those things in a moment
- Put metadata in memcached for partitioning and fail-over information
memcached

- LiveJournal’s distributed caching system *(used practically everywhere!)*
- Memory based – memory is cheap!
- Linux 2.6 (epoll) or FreeBSD (kqueue)
  - Low overhead for many many connections
- Run it on boxes with free memory
- ...or a dedicated cluster:
  Facebook has *more than five hundred* dedicated memcached servers (a lot of memory!)
more memcached

- No “master” – fully distributed
- Simple lightweight protocol (binary protocol coming)
- Scaling and high-availability is “built-in”
- Servers are dumb – clients calculate which server to use based on the cache key
- Clients in perl, java, php, python, ruby, ...
- New C client library, libmemcached
  http://tangent.org/552/libmemcached.html
How to use memcached

• It’s a cache, not a database

• Store data safely somewhere else

• Pass-through cache (id = session_id or whatever):

  Read
  ```php
  $data = memcached_fetch( $id );
  return $data if $data
  $data = db_fetch( $id );
  memcached_store( $id, $data );
  return $data;
  ```

  Write
  ```php
  db_store( $id, $data );
  memcached_store( $id, $data );
  ```
Client Side Replication

- memcached is a cache - the data might “get lost”
- What if a cache miss is Really Expensive?
- Store all writes to several memcached servers
- Client libraries are starting to support this natively
Store complex data

• Most (all?) client libraries support complex data structures

• A bit flag in memcached marks the data as “serialized” (another bit for “gzip”)

• All this happens on the client side – memcached just stores a bunch of bytes

• Future: Store data in JSON? Interoperability between languages!
Store complex data 2

- Primary key lookups are probably not worth caching
- Store things that are expensive to figure out!

```php
function get_slow_summary_data($id) {
    $data = memcached_fetch($id);
    return $data if $data
    $data = do_complicated_query($id);
    memcached_store($id, $data);
    return $data;
}
```
Cache invalidation

- Writing to the cache on updates is hard!
- Caching is a trade-off
- You trade “fresh” for “fast”
- Decide how “fresh” is required and deal with it!
- Explicit deletes if you can figure out what to delete
- Add a “generation” / timestamp / whatever to the cache key

```
select id, unix_timestamp(modified_on) as ts from users where username = 'ask';
memcached_fetch("user_friend_updates; $id; $ts")
```
Caching is a trade-off

- Can’t live with it?

- Make the primary data-source faster or data-store scale!
Database scaling

How to avoid buying that gazillion dollar Sun box

~$4,000,000
Vertical

~$3,200
( = 1230 for $4.0M!)
Be Simple

- Use MySQL!
  - It’s fast and it’s easy to manage and tune
  - Easy to setup development environments
  - Other DBs can be faster at certain complex queries but are harder to tune – and MySQL is catching up!

- Avoid making your schema too complicated

- Ignore some of the upcoming advice until you REALLY need it!
  - (even the part about not scaling your DB “up”)

- PostgreSQL is fast too :-}
Replication

More data more places!
Share the love load
Basic Replication

- Good Great for read intensive applications
- Write to one master
- Read from many slaves

Lots more details in “High Performance MySQL”

old, but until MySQL 6 the replication concepts are the same
Relay slave replication

• Running out of bandwidth on the master?
• Replicating to multiple data centers?
• A “replication slave” can be master to other slaves
• Almost any possible replication scenario can be setup (circular, star replication, ...)

[Diagram showing a master database with relay slaves A and B, and slave databases connected through reads and writes.]
Replication Scaling – Reads

- Reading scales well with replication
- Great for (mostly) read-only applications

(thanks to Brad Fitzpatrick!)
Replication Scaling – Writes

(aka when replication sucks)

- Writing doesn’t scale with replication
- All servers need to do the same writes

Diagram:
- Two columns: reads and writes
- Five servers
  - Three reads
  - Two writes
Partition the data

Divide and Conquer!

or

Web 2.0 Buzzword Compliant!
Now free with purchase of milk!!
Partition your data

- 96% read application? Skip this step...
- Solution to the too many writes problem: Don’t have all data on all servers
- Use a separate cluster for different data sets
The Write Web!

- Replication too slow? Don’t have replication slaves!
- Use a (fake) **master-master** setup and partition / shard the data!
- Simple redundancy!
- No latency from commit to data being available
- Don’t bother with fancy 2 or 3 phase commits
  - (Make each “main object” (user, product, ...) always use the same master – as long as it’s available)
Partition with a global master server

- Can’t divide data up in “dogs” and “cats”?
- Flexible partitioning!
- The “global” server keeps track of which cluster has the data for user “623”
- Get all PKs from the global master
- Only auto_increment columns in the “global master”
- Aggressively cache the “global master” data (memcached)
- and/or use MySQL Cluster (ndb)

Diagram:
- global master
- master
- slave (backup)
- webservers
- user 623
- cluster 1, cluster 2, cluster 3
- select * from some_data where user_id = 623
- user 623 is in cluster 3
Master – Master setup

• Setup two replicas of your database copying changes to each-other

• Keep it simple!  (all writes to one master)

• Instant fail-over host – no slave changes needed

• Configuration is easy!
  
  • `set-variable = auto_increment_increment=2`
  • `set-variable = auto_increment_offset=1`

  • (offset = 2 on second master)

• Setup both systems as a slave of the other
Online Schema Changes

The reasons we love master-master!

- Do big schema changes with no downtime!
  - Stop A to B replication
  - Move traffic to B
  - Do changes on A
  - Wait for A to catchup on replication
  - Move traffic to A
  - Re-start A to B replication
Hacks!

Don’t be afraid of the data-duplication monster

http://flickr.com/photos/firevixen/75861588/
Summary tables!

- Find queries that do things with COUNT(\*) and GROUP BY and create tables with the results!
- Data loading process updates both tables
- or hourly/daily/... updates
- Variation: Duplicate data in a different “partition”
  - Data affecting both a “user” and a “group” goes in both the “user” and the “group” partition (Flickr does this)
Summary databases!

• Don’t just create summary tables
• Use summary databases!
• Copy the data into special databases optimized for special queries
  • full text searches
  • index with both cats and dogs
  • anything spanning all clusters
• Different databases for different latency requirements (RSS feeds from replicated slave DB)
Make everything repeatable

- Script failed in the middle of the nightly processing job? (they will sooner or later, no matter what)
- How do you restart it?
- Build your “summary” and “load” scripts so they always can be run again! (and again and again)
- One “authoritative” copy of a data piece – summaries and copies are (re)created from there
Asynchronous data loading

- Updating counts? Loading logs?

- Don’t talk directly to the database, send updates through Spread (or whatever) to a daemon loading data

- Don’t update for each request
  update counts set count=count+1 where id=37

- Aggregate 1000 records or 2 minutes data and do fewer database changes
  update counts set count=count+42 where id=37

- Being disconnected from the DB will let the frontend keep running if the DB is down!
“Manual” replication

- Save data to multiple “partitions”
- Application writes two places or
- last_updated/modified_on and deleted columns or
- Use triggers to add to “replication_queue” table
- Background program to copy data based on the queue table or the last_updated column
- Build summary tables or databases in this process
- Build star/spoke replication system
Preload, -dump and -process

• Let the servers do as much as possible without touching the database directly

• Data structures in memory – ultimate cache!

• Dump never changing data structures to JS files for the client to cache

• Dump smaller read-only often accessed data sets to SQLite or BerkeleyDB and rsync to each webserver (or use NFS, but...)

• Or a MySQL replica on each webserver
Stored Procedures

Dangerous

- Not horizontal

- Bad:
  Work done in the database server (unless it’s read-only and replicated)

- Good:
  Work done on one of the scalable web fronts

- Only do stored procedures if they save the database work (network-io work > SP work)
a brief diversion ...

Running Oracle now?

- Move read operations to MySQL!
- Replicate from Oracle to a MySQL cluster with “manual replication”
- Use triggers to keep track of changed rows in Oracle
- Copy them to the MySQL master server with a replication program
- Good way to “sneak” MySQL in ...
Optimize the database

Faster, faster, faster ....
... very briefly

• The whole conference here is about this
• ... so I’ll just touch on a few ideas
Memory for MySQL = good

• Put as much memory you can afford in the server (Currently 2GB sticks are the best value)

• InnoDB: Let MySQL use ~all memory (don’t use more than is available, of course!)

• MyISAM: Leave more memory for OS page caches

• Can you afford to lose data on a crash? Optimize accordingly

• Disk setup: We’ll talk about RAID later
What’s your app doing?

- Enable query logging in your development DB!
- Are all those queries really necessary? Cache candidates?
  - (you do have a devel db, right?)
- Just add “log=/var/lib/mysql/sql.log” to .cnf
- Slow query logging:
  - log-slow-queries
  - log-queries-not-using-indexes
  - long_query_time=1
- mysqldumps slow parses the slow log
- 5.1+ does not require a server restart and, can log directly into a CSV table...
Table Choice

• Short version:
  Use InnoDB, it’s harder to make them fall over

• Long version:
  Use InnoDB except for
  • Big read-only tables (smaller, less IO)
  • High volume streaming tables (think logging)
    • Locked tables / INSERT DELAYED
    • ARCHIVE table engine
  • Specialized engines for special needs
  • More engines in the future
  • For now: InnoDB
Multiple MySQL instances

- Run different MySQL instances for different workloads
- Even when they share the same server anyway!
- InnoDB vs MyISAM instance
- Move to separate hardware and replication easier
- Optimize MySQL for the particular workload
- Very easy to setup with the instance manager or mysqld_multi
- mysql.com init.d script supports the instance manager (don’t use the redhat/fedora script!)
Config tuning helps
Query tuning works

• Configuration tuning helps a little

• The big performance improvements come from schema and query optimizations – focus on that!

• Design schema based on queries

• Think about what kind of operations will be common on the data; don’t go for “perfect schema beauty”

• What results do you need? (now and in the future)
• Use the “EXPLAIN SELECT ...” command to check the query
• Baron Schwartz talks about this 2pm on Tuesday!
• Be sure to read
Use smaller data

- Use Integers
  - Always use integers for join keys
  - And when possible for sorts, group bys, comparisons
- Don’t use bigint when int will do
- Don’t use varchar(255) when varchar(20) will do
Store Large Binary Objects
(aka how to store images)

• Meta-data table (name, size, ...)

• Store images either in the file system
  • meta data says “server ‘123’, filename ‘abc’”
  • (If you want this; use mogilefs or Amazon S3 for storage!)

• OR store images in other tables
  • Split data up so each table don’t get bigger than ~4GB

• Include “last modified date” in meta data

• Include it in your URLs if possible to optimize caching (/images/$timestamp/$id.jpg)
Reconsider Persistent DB Connections

• DB connection = thread = memory
• With partitioning all httpd processes talk to all DBs
• With lots of caching you might not need the main database that often
• MySQL connections are fast
• Always use persistent connections with Oracle!
  • Commercial connection pooling products
• pgsql, sybase, oracle? Need thousands of persistent connections?
  • In Perl the new DBD::Gofer can help with pooling!
InnoDB configuration

- `innodb_file_per_table`
  Splits your InnoDB data into a file per table instead of one big annoying file
  - Makes `optimize table 'table'` clear unused space

- `innodb_buffer_pool_size=($MEM*0.80)`

- `innodb_flush_log_at_trx_commit` setting

- `innodb_log_file_size`

- `transaction-isolation = READ-COMMITTED`
My favorite MySQL feature

- `insert into t (somedate) values ("blah");`
- `insert into t (someenum) values ("bad value");`
- Make MySQL picky about bad input!
  - `SET sql_mode = 'STRICT_TRANS_TABLES';`
- Make your application do this on connect
Don’t overwork the DB

• Databases don’t easily scale
• Don’t make the database do a ton of work
• Referential integrity is good
  • Tons of stored procedures to validate and process data not so much
• Don’t be too afraid of de-normalized data – sometimes it’s worth the tradeoffs (call them summary tables and the DBAs won’t notice)
Use your resources wisely

don’t implode when things run warm
Work in parallel

• Split the work into smaller (but reasonable) pieces and run them on different boxes

• Send the sub-requests off as soon as possible, do something else and then retrieve the results

Are the horizontal lines parallel or do they slope?
Job queues

• Processing time too long for the user to wait?

• Can only process $N$ requests / jobs in parallel?

• Use queues (and external worker processes)

• IFRAMEs and AJAX can make this really spiffy (tell the user “the wait time is 20 seconds”)

Job queue tools

• Database “queue”
  • Dedicated queue table or just `processed_on` and `grabbed_on` columns
  
• Webserver submits job

• First available “worker” picks it up and returns the result to the queue

• Webserver polls for status
More Job Queue tools

- **beanstalkd** - great protocol, *fast*, no persistence (yet)
  http://xph.us/software/beanstalkd/

- **gearman** - for one off out-of-band jobs
  http://www.danga.com/gearman/

- **starling** - from twitter, memcached protocol, disk based persistence
  http://rubyforge.org/projects/starling/

- **TheSchwartz** from SixApart, used in Movable Type

- **Spread**

- **MQ / Java Messaging Service(?) / ...**
Log http requests

- Log slow http transactions to a database:
  - time, response_time, uri, remote_ip, user_agent, request_args, user, svn_branch_revision, log_reason (a “SET” column), ...
- Log to ARCHIVE tables, rotate hourly / weekly / ...
- Log 2% of all requests!
- Log all 4xx and 5xx requests
- Great for statistical analysis!
  - Which requests are slower
  - Is the site getting faster or slower?
- Time::HiRes in Perl, microseconds from gettimeofday system call
Intermission?
Use light processes for light tasks

- Thin proxies servers or threads for “network buffers”
- Goes between the user and your heavier backend application
- Built-in load-balancing! (for Varnish, perlbal, ...)
- httpd with mod_proxy / mod_backhand
  - perlbal
    - more on that in a bit
- Varnish, squid, pound, ...
Proxy illustration

**Users**

- perlbal or mod_proxy
  - low memory/resource usage

**Backends**

- lots of memory
- db connections etc
Light processes

- Save memory and database connections
- This works spectacularly well. Really!
- Can also serve static files
- Avoid starting your main application as root
- Load balancing
- In particular important if your backend processes are “heavy”
Light processes

- Apache 2 makes it **Really Easy**

```
ProxyPreserveHost On
<VirtualHost *>
  ServerName combust.c2.askask.com
  ServerAlias *.c2.askask.com
  RewriteEngine on
  RewriteRule (.*$) http://localhost:8230$1 [P]
</VirtualHost>
```

- Easy to have different “backend environments” on one IP

- **Backend setup (Apache 1.x)**
  ```
  Listen 127.0.0.1:8230
  Port 80
  ```
perlbal configuration

CREATE POOL my_apaches
  POOL my_apaches ADD 10.0.0.10:8080
  POOL my_apaches ADD 10.0.0.11:8080
  POOL my_apaches ADD 10.0.0.12
  POOL my_apaches ADD 10.0.0.13:8081

CREATE SERVICE balancer
  SET listen = 0.0.0.0:80
  SET role = reverse_proxy
  SET pool = my_apaches
  SET persist_client = on
  SET persist_backend = on
  SET verify_backend = on
ENABLE balancer
A few thoughts on development ...
All Unicode All The Time

• The web is international and multilingual, deal with it.

• All Unicode all the time!
  (except when you don’t need it – urls, email addresses, ...)

• Perl: DBD::mysql was fixed last year! PHP 6 will have improved Unicode support. Ruby 2 will someday, too...

• It will never be easier to convert than now!
Use UTC

Coordinated Universal Time

• It might not seem important now, but some day ...
• It will never be easier to convert than now!
• Store all dates and times as UTC, convert to “local time” on display
Build on APIs

• All APIs All The Time!

• Use “clean APIs” Internally in your application architecture

• Loosely coupled APIs are easier to scale
  • Add versioning to APIs (“&api_version=123”)

• Easier to scale development

• Easier to scale deployment

• Easier to open up to partners and users!
Why APIs?

• Natural place for “business logic”
  • Controller = “Speak HTTP”
  • Model = “Speak SQL”
  • View = “Format HTML / …”

• **API** = “Do Stuff”

• Aggregate just the right amount of data
  • Awesome place for optimizations that matter!
  • The data layer knows too little
More development philosophy

• Do the Simplest Thing That Can Possibly Work
• ... but do it really well!
• Balance the complexity, err on the side of simple
• This is hard!
Pay your technical debt

• Don’t incur technical debt
  • “We can’t change that - last we tried the site went down”
  • “Just add a comment with ‘TODO’”
  • “Oops. Where are the backups? What do you mean ‘no’?”
  • “Who has the email with that bug?”
• Interest on technical debt will kill you
• *Pay it back as soon as you can!*
Coding guidelines

• Keep your formatting consistent!
  • perl: perltidy, perl best practices, Perl::Critic
• Keep your APIs and module conventions consistent
• Refactor APIs mercilessly (in particular while they are not public)
qmail lessons

- Lessons from 10 years of qmail
- Research paper from Dan Bernstein
  http://cr.yp.to/qmail/qmailsec-20071101.pdf
- Eliminate bugs
  - Test coverage
  - Keep data flow explicit
- (continued)
qmail lessons (2)

- Eliminate code – less code = less bugs!
- Refactor common code
- Reuse code (Unix tools / libs, CPAN, PEAR, Ruby Gems, ...)
- Reuse access control
- Eliminate trusted code – what needs access?
- Treat transformation code as completely untrusted
Joint Strike Fighter

• ~Superset of the “Motor Industry Software Reliability Association Guidelines For The Use Of The C Language In Vehicle Based Software”

• Really Very Detailed!

• No recursion!  (Ok, ignore this one :-) )

• Do make guide lines – know when to break them

• Have code reviews - make sure every commit email gets read (and have automatic commit emails in the first place!)
High Availability

and Load Balancing

and Disaster Recovery
High Availability

- **Automatically handle failures!** (bad disks, failing fans, “oops, unplugged the wrong box”, ...)

- For your app servers the load balancing system should take out “bad servers” (most do)
  
  - perlbal or Varnish can do this for http servers

- Easy-ish for things that can just “run on lots of boxes”
Make that service always work!

- Sometimes you need a service to always run, but on specific IP addresses
  - Load balancers (level 3 or level 7: perlbal/varnish/squid)
  - Routers
  - DNS servers
  - NFS servers
- Anything that has failover or an alternate server – the IP needs to move (much faster than changing DNS)
Load balancing

- Key to horizontal scaling (duh)
- 1) All requests goes to the load balancer
   2) Load balancer picks a “real server”
- Hardware (lots of vendors)
  Coyote Point have relatively cheaper ones
- Look for older models for cheap on eBay!
- Linux Virtual Server
- Open/FreeBSD firewall rules (pf firewall pools)
  (no automatic failover, have to do that on the “real servers”)
Load balancing 2

• Use a “level 3” (tcp connections only) tool to send traffic to your proxies

• Through the proxies do “level 7” (http) load balancing

• perlbal has some really good features for this!
perlbal

- Event based for HTTP load balancing, web serving, and a mix of the two (see below).
- Practical fancy features like “multiplexing” keep-alive connections to both users and back-ends
- Everything can be configured or reconfigured on the fly
- If you configure your backends to only allow as many connections as they can handle (you should anyway!) perlbal will automatically balance the load “perfectly”
- Can actually give Perlbal a list of URLs to try. Perlbal will find one that's alive. Instant failover!
Varnish

• Modern high performance http accelerator
• Optimized as a “reverse cache”
• Whenever you would have used squid, give this a look
• Recently got “Vary” support
• Super efficient (except it really wants to “take over” a box)
• Written by Poul-Henning Kamp, famed FreeBSD contributor
• BSD licensed, work is being paid by a norwegian newspaper
• http://www.varnish-cache.org/
Fail-over tools

“move that IP”
Buy a “hardware load balancer”

- Generally *Quite Expensive*
  - (Except on eBay - used network equipment is often great)
- Not appropriate (cost-wise) until you have MANY servers
- If the feature list fits it “Just Works”
- ... but when we are starting out, what do we use?
wackamole

- Simple, just moves the IP(s)
- Can embed Perl so you can run Perl functions when IPs come and go
- Easy configuration format
- Setup “groups of IPs”
- Supports Linux, FreeBSD and Solaris
- Spread toolkit for communication
- Easy to troubleshoot (after you get Spread working...)
- [http://www.backhand.org/wackamole/](http://www.backhand.org/wackamole/)
Heartbeat

- Monitors and moves services (an IP address is “just a service”)
- v1 has simple but goofy configuration format
- v2 supports all sorts of groupings, larger clusters (up to 16 servers)
- Uses /etc/init.d type scripts for running services
- Maybe more complicated than you want your HA tools
- http://www.linux-ha.org/
Carp + pfsync

- Patent-free version of Ciscos “VRRP” (Virtual Router Redundancy Protocol)
- FreeBSD and OpenBSD only
- Carp (moves IPs) and pfsync (synchronizes firewall state)
- (awesome for routers and NAT boxes)
- Doesn’t do any service checks, just moves IPs around
mysql master master replication manager

• mysql-master-master tool can do automatic failover!
• No shared disk
• Define potential “readers” and “writers”
• List of “application access” IPs
• Reconfigures replication
• Moves IPs

• http://code.google.com/p/mysql-master-master/
  http://groups.google.com/group/mmm-devel/
Suggested Configuration

• Open/FreeBSD routers with Carp+pfsync for firewalls

• A set of boxes with perlbal + wackamole on static “always up” HTTP enabled IPs

• Trick on Linux: Allow the perlbal processes to bind to all IPs (no port number tricks or service reconfiguration or restarts!)
  echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind
  or
  sysctl -w net.ipv4.ip_nonlocal_bind=1
  or
  echo net.ipv4.ip_nonlocal_bind = 1 >> /etc/sysctl.conf

• Dumb regular http servers “behind” the perlbal ones

• wackamole for other services like DNS

• mmm for mysql fail-over
Redundancy fallacy!

- Don’t confuse load-balancing with redundancy
- What happens when one of these two fail?
Oops – no redundancy!

- Always have “n+1” capacity
- Consider have a “passive spare” (active/passive with two servers)
- Careful load monitoring!
  - Munin http://munin.projects.linpro.no/
  - MySQL Network
  - (ganglia, cacti, ...)

More than 100% load on 1 server!
High availability
Shared storage

• NFS servers (for diskless servers, ...)
• Failover for database servers
• Traditionally either via fiber or SCSI connected to both servers
• Or NetApp filer boxes
• All expensive and smells like “the one big server”
Cheap high availability storage with DRBD

- Synchronizes a block device between two servers!
- “Network RAID1”
- Typically used in Active/Primary-Standby/Secondary setup
- If the active server goes down the secondary server will switch to primary, run fsck, mount the device and start the service (MySQL / NFS server / ...)
- v0.8 can do writes on both servers at once – “shared disk semantics” (you need a filesystem on top that supports that, OCFS, GFS, ... – probably not worth it, but neat)
Disaster Recovery

- Separate from “fail-over” (no disaster if we failed-over...)
- “The rescue truck fell in the water”
- “All the ‘redundant’ network cables melted”
- “The datacenter got flooded”
- “The grumpy sysadmin sabotaged everything before he left”
Disaster Recovery Planning

- You won’t be back up in 2 hours, but plan so you quickly will have an idea how long it will be
- Have a status update site / weblog
- Plans for getting hardware replacements
- Plans for getting running temporarily on rented “dedicated servers” (ev1 servers, rackspace, ...)
- And ....
Backup your database!

- Binary logs!
  - Keep track of “changes since the last snapshot”
- Use replication to another site
  (doesn’t help on “for $table = @tables { truncate $table }”)
- On small databases use `mysqldump`
  (or whatever similar tool your database comes with)
- Zmanda MySQL Backup
  packages the different tools and options
Backup Big Databases

• **Use mylvmbackup** to snapshot and archive

• Requires data on an LVM device (just do it)

• **InnoDB:**
  Automatic recovery! (ooh, magic)

• **MyISAM:**
  Read Lock your database for a few seconds before making the snapshot
  (on MySQL do a “FLUSH TABLES” first (which might be slow) and then a “FLUSH TABLES WITH READ LOCK” right after)

• Sync the LVM snapshot elsewhere

• And then remove the snapshot!

• **Bonus Optimization:**
  Run the backup from a replication slave!
Backup on replication slave

• Or just run the backup from a replication slave ...
• Keep an extra replica of your master
  • shutdown mysqld and archive the data
• Small-ish databases:
  mysqldump --single-transaction
All Automation All The Time
or
How to manage 200 servers in your spare-time
System Management
Keep software deployments easy

• Make upgrading the software a simple process
• Script database schema changes
• Keep configuration minimal
  • Servername ("www.example.com")
  • Database names ("userdb = host=db1;db=users";...)
  • If there's a reasonable default, put the default in the code (for example)
  • "deployment_mode = devel / test / prod" lets you put reasonable defaults in code
Easy software deployment 2

• How do you distribute your code to all the app servers?

• Use your source code repository (Subversion etc)! (tell your script to svn up to http://svn/branches/prod revision 123 and restart)

• .tar.gz to be unpacked on each server

• .rpm or .deb package

• NFS mount and symlinks

• No matter what: Make your test environment use the same mechanism as production and:
  **Have it scripted!**
have everything scripted!

http://flickr.com/photos/karlequin/84829873/
Configuration management

Rule Number One

- Configuration in SVN (or similar)
- “infrastructure/” repository
- SVN rather than rcs to automatically have a backup in the Subversion server – which you are carefully backing up anyway
- Keep notes! Accessible when the wiki is down; easy to grep
- Don’t worry about perfect layout; just keep it updated
Configuration management

Rule Two

- Repeatable configuration!
- Can you reinstall any server Right Now?
- Use tools to keep system configuration in sync
- Upcoming configuration management (and more) tools!
  - csync2 (librsync and sqlite based sync tool)
  - puppet (central server, rule system, ruby!)
puppet

- Automating sysadmin tasks!
- 1) Client provides “facter” to server
  2) Server makes configuration
  3) Client implements configuration

- service { "sshd":
    enable => true,
    ensure => running
  }

- package { "vim-enhanced": ensure => installed }
package { "emacs": ensure => installed }
node db-server inherits standard {
    include mysql_server
    include solfo_hw
}

dnode db2, db3, db4 inherits db-server {}

dnode trillian inherits db-server {
    include ypbot_devel_dependencies
}

# -----------------------------

class mysql_client {
    package { "MySQL-client-standard": ensure => installed }
    package { "MySQL-shared-compat": ensure => installed }
}

class mysql_server {
    file { "/mysql":
        ensure => directory,
    }
    package { "MySQL-server-standard": ensure => installed }

    include mysql_client
}
puppet mount example

- Ensure an NFS mount exists, except on the NFS servers

```python
class nfs_client_pkg {

    file { "/pkg":
        ensure => directory,
    }

    $mount = $hostname ? {
        "nfs-a" => absent,
        "nfs-b" => absent,
        default => mounted
    }

    mount {
        "/pkg":
        atboot => true,
        device => 'nfs.la.sol:/pkg',
        ensure => $mount,
        fstype => 'nfs4',
        options => 'ro,intr,noatime',
        require => File["/pkg"],
    }

}```
More puppet features

• In addition to services, packages and mounts...
  
  • Manage users
  
  • Manage crontabs
  
  • Copy configuration files (with templates)
  
  • … and much more
  
• Recipes, reference documentation and more at http://reductivelabs.com/
Backups!

- Backup everything you can
  - Check/test the backups routinely
- Super easy deployment: `rsnapshot`
  - Uses rsync and hardlinks to efficiently store many backup generations
  - Server initiated – just needs ssh and rsync on client
  - Simple restore – files
- Other tools
  - Amanda (Zmanda)
  - Bacula
Backup is cheap!

• Extra disk in a box somewhere? That can do!

• Disks are cheap – get more!

• Disk backup server in your office:
  
  Enclosure + PSU: $275
  
  CPU + Board + RAM: $400
  
  3ware raid (optional): $575
  
  6x1TB disks: $1700 (~4TB in raid 6)
  
  = $3000 for 4TB backup space, easily expandable
  
  (or less than $5000 for 9TB space with raid 6 and hot standby)

• Ability to get back your data = **Priceless!**
RAID Levels


http://www.cs.berkeley.edu/~pattrsn/Arch/prototypes2.html
Basic RAID levels

- RAID 0
  Stripe all disks (capacity = N*S)
  Fail: Any disk

- RAID 1
  Mirror all disks (capacity = S)
  Fail: All disks

- RAID 10
  Combine RAID 1 and 0 (capacity = N*S / 2)

- RAID 5
  RAID 0 with parity (capacity = N*S - S)
  Fail: 2 disks

- RAID 6
  Two parity disks (capacity = N*S - S*2)
  Fail: 3 disks!
RAID 1

- Mirror all disks to all disks
- Simple - easiest to recover!
- Use for system disks and small backup devices
RAID 0

- Use for redundant database mirrors or scratch data that you can quickly rebuild
- Absolutely never for anything you care about
- Failure = system failure
- Great performance; no safety
- Capacity = 100%
- Disk IO = every IO available is “useful”
RAID 10

• Stripe of mirrored devices
• IO performance and capacity of half your disks - not bad!
• Relatively good redundancy, lose one disk from each of the “sub-mirrors”
• Quick rebuild: Just rebuild one mirror
• More disks = more failures! If you have more than X disks, keep a hot spare.
RAID 5

- Terrible database performance
- A partial block write = read all disks!
- When degraded a RAID 5 is a RAID 0 in redundancy!
- Rebuilding a RAID 5 is a great way to find more latent errors
- Don’t use RAID 5 – just not worth it
RAID 6

- Like RAID 5 but doesn’t fail as easily
- Can survive two disks failing
- Don’t make your arrays too big
  - 12 disks = 12x failure rate of one disk!
- Always keep a hot-spare if you can
Hardware or software RAID?

- Hardware RAID: Worth it for the Battery Backup Unit!
  - Battery allows the controller to – safely – fake “Sure mister, it’s safely on disk” responses

- No Battery? Use Software RAID
  - Low or no CPU use
  - Easier and faster to recover from failures!
    - Write-intent bitmap
    - More flexible layout options
      - RAID 1 partition for system + RAID 10 for data on each disk
### Service Status Details For All Hosts

<table>
<thead>
<tr>
<th>Host</th>
<th>Service</th>
<th>Status</th>
<th>Last Check</th>
<th>Duration</th>
<th>Attempt</th>
<th>Status Information</th>
</tr>
</thead>
</table>
| app1 | Root Partition| OK     | 2008-03-23 21:15:42 | 5d 19h 45m 13s | 1/4     | DISK OK - free space / 243 MB (22% inode=94%):  
/dev/mapper/vg/vgvar: 80 /dev/md0: 31  
/dev/mapper/vg/vgroot: 78 /dev/mapper/vg/vgusr: 54  
/dev/mapper/vg/mirror/local: 1 /dev/mapper/vg/mirror/xen: 4  
/dev/mapper/vg/vgtmp: 2 |
| df   | OK            |        | 2008-03-23 21:15:59 | 518d 14h 26m 46s | 1/4     | DNS OK: 0.016 seconds response time. dnsntest.la.sol returns 127.0.0.2 |
| dns_auth_internal | OK        |        | 2008-03-23 21:15:45 | 573d 16h 35m 22s | 1/4     | DNS OK: 0.006 seconds response time. dnsntest.la.sol returns 127.0.0.2 |
| dns_cache | OK       |        | 2008-03-23 21:17:18 | 573d 16h 35m 22s | 1/4     | DNS OK: 0.006 seconds response time. dnsntest.la.sol returns 127.0.0.2 |
| ldap | OK            |        | 2008-03-23 21:17:55 | 0d 16h 28m 52s | 1/4     | LDAP OK - 0.002 seconds response time |
| ntpd | OK            |        | 2008-03-23 21:15:50 | 420d 14h 59m 24s | 1/4     | NTP OK: Offset -5.125999451e-08 secs |
| smtp | OK            |        | 2008-03-23 21:17:15 | 53d 5h 51m 9s | 1/4     | SMTP OK - 0.023 sec. response time |
| ssh  | OK            |        | 2008-03-23 21:17:18 | 7d 18h 41m 38s | 1/4     | SSH OK - OpenSSH_4.3 (protocol 2.0) |
| ups1 | CRITICAL      |        | 2008-03-23 21:15:43 | 6d 9h 0m 1s | 1/4     | UPS OK - Status=Online Utility=111.05V Batt=100.00% Load=55.00% Temp=27.0C |
| app2 | df            | OK     | 2008-03-23 21:15:59 | 8d 20h 55m 55s | 1/4     | |
| ntpd | OK            |        | 2008-03-23 21:16:27 | 8d 20h 53m 45s | 1/4     | NTP OK: Offset -0.4193743467 secs |
| ssh  | OK            |        | 2008-03-23 21:17:23 | 8d 20h 54m 51s | 1/4     | SSH OK - OpenSSH_4.7 (protocol 2.0) |
| app3 | df            | CRITICAL | 2008-03-23 21:17:23 | 0d 8h 52m 30s | 1/4     | |
| memcached | CRITICAL |        | 2008-03-23 21:17:55 | 0d 8h 51m 35s | 1/4     | |
| ntpd | OK            |        | 2008-03-23 21:15:35 | 8d 8h 50m 55s | 1/4     | NTP OK: Offset -0.44059890839 secs |
| ssh  | OK            |        | 2008-03-23 21:17:45 | 8d 8h 51m 0s | 1/4     | SSH OK - OpenSSH_4.7 (protocol 2.0) |
| ypglot app | OK      |        | 2008-03-23 21:16:22 | 4d 4h 13m 26s | 1/4     | HTTP OK  HTTP/1.1 200 OK - 25897 bytes in 0.116 seconds |
| son1 | ssh           | OK     | 2008-03-23 21:16:40 | 0d 8h 50m 10s | 1/4     | SSH OK - OpenSSH_4.4 (protocol 1.99) |
nagios

- Monitoring “is the website up” is easy
- Monitoring dozens or hundreds of sub-systems is hard
- Monitor everything!
- Disk usage, system daemons, applications daemons, databases, data states, ...
nagios configuration tricks

- nagios configuration is famously painful
- Somewhat undeserved!

examples of simple configuration
- templates
- groups
nagios best practices

• All alerts must be “important” – if some alerts are ignored, all other alerts easily are, too.
• Don’t get 1000 alerts if a DB server is down
• Don’t get paged if 1 of 50 webservers crashed
• Why do you as a non-sysadmin care?
  • Use nagios to help the sysadmins fix the application
  • Get information to improve reliability
Resource management

- If possible, only run one service per server (makes monitoring/managing your capacity much easier)

- Balance how you use the hardware
  - Use memory to save CPU or IO
  - Balance your resource use (CPU vs RAM vs IO)
  - Extra memory on the app server? Run memcached!
  - Extra CPU + memory? Run an application server in a Xen box!

- Don’t swap memory to disk. Ever.
Netboot your application servers!

- **Definitely netboot the installation** *(you’ll never buy another server with a tedious CD/DVD drive)*
  - RHEL / Fedora: Kickstart + puppet = from box to all running in ~10 minutes
- Netboot application servers
- FreeBSD has awesome support for this
- Debian is supposed to
- Fedora Core 7 & 8 ?? looks like it will (RHEL5uX too?)
No shooting in foot!

- Ooops? Did that leak memory again? Development server went kaboom?
- Edit `/etc/security/limits.conf`
  - `@users   soft rss     250000`
  - `@users   hard rss    250000`
  - `@users   hard as    500000`
- Use to set higher open files limits for mysqld etc, too!
noatime mounts

• Mount ~all your filesystems “noatime”

• By default the filesystem will do a \texttt{write} every time it accesses/reads a file!

• That’s clearly \texttt{insane}

• Stop the madness, mount noatime

\begin{verbatim}
/dev/vg0/lvhome /home ext3 defaults 1 2
/dev/vg0/lvhome /home ext3 noatime 1 2
\end{verbatim}
graph everything!

- **mrtg**
The Multi Router Traffic Grapher

- **rrdtool**
  round-robin-database tool
  - Fixed size database handling time series data
  - Lots of tools built on rrdtool

- **ganglia**
  cluster/grid monitoring system
Historical perspective

basic bandwidth graph

Launch

Steady growth

Try CDN

Enable compression for all browsers
munin

• “Hugin and Munin are the ravens of the Norse god king Odin. They flew all over Midgard for him, seeing and remembering, and later telling him.”

• Munin is also AWESOME!

• Shows trends for system statistics

• Easy to extend
mysql query stats

MySQL queries 3307 - by day

- **select**: Cur: 339.41, Min: 110.58, Avg: 379.53, Max: 920.81
- **delete**: Cur: 0.00, Min: 0.00, Avg: 11.20m, Max: 45.84m
- **update**: Cur: 1.42, Min: 898.54m, Avg: 1.64, Max: 87.46
- **insert**: Cur: 8.98, Min: 1.77, Avg: 4.62, Max: 17.87
- **cache_hits**: Cur: 72.85, Min: 8.91, Avg: 125.75, Max: 550.90
- **replace**: Cur: 53.95m, Min: 3.47m, Avg: 32.38m, Max: 99.04m
- **total**: Cur: 0.00, Min: 0.00, Avg: 510.17, Max: 1.42k

Last update: Mon Feb 11 09:55:06 2008
Query cache useful?

- Is the MySQL query cache useful for your application?
- Make a graph!
- In this particular installation it answers half of the selects
squid cache hitratio?

• Red: Cache Miss
• Green: Cache Hit
• Increased cache size to get better hit ratio
• Huh? When?

Don’t confuse graphs with “hard data”

Keep the real numbers, too!
munin: capacity planning, cpu

- xen system
  6 cpus
- plenty to spare
Blocking on disk I/O?

- Pink: iowait
- This box needs more memory or faster disks!
More IO Wait fun

- 8 CPU box - harder to see the details
- High IO Wait
More IO Wait fun

- Upgraded memory, iowait dropped!
IO Statistics

- per disk IO statistics
- more memory, less disk IO
more memory stats

- fix app config
- fix perlbal leak

plenty memory free
room for memcached?

took a week to use new memory for caching

plenty memory to run memcached here!
munin: spot a problem?

- 1 CPU 100% busy on “system”?
- Started a few days ago
munin: spot a problem?

- Has it happened before?
- Yup - occasionally!
munin: spot a problem?

- IPMI driver went kaboom!
Make your own Munin plugin

- Any executable with the right output

```
# ./load config
graph_title Load average
graph_args --base 1000 -l 0
graph_vlabel load
...
load.label load
load.info Average load for ...
```

```
# ./load fetch
load.value 1.67
```
Munin as a nagios agent

- Use a Nagios plugin to talk to munin!
- Munin is already setup to monitor important metrics
- Nagios plugin talks to munin as if the collector agent

```
define service {
    use               local-service
    hostgroup_name    xen-servers,db-servers,app-servers
    service_description df
    check_command     check_munin!df!88!94
}
```
A little on hardware

• Hardware is a commodity!

• Configuring it isn’t (yet – Google AppEngine!)

• Managed services - cthought.com, RackSpace, SoftLayer ...

• Managing hardware != Managing systems

• Rent A Server
  (crummy support, easy on hardware replacements, easy on cashflow)

• Amazon EC2 (just announced persistent storage!)

• Use standard configurations and automatic deployment

• Now you can buy or rent servers from anywhere!
Use a CDN

- If you serve more than a few TB static files a month...
- Consider a Content Delivery Network
- Fast for users, easier on your network
- Pass-through proxy cache - easy deployment
- Akamai, LimeLight, PantherExpress, CacheFly, ...
  (only Akamai supports compressed files (??))
Client Performance

“Best Practices for Speeding Up Your Web Site”
Recommended Reading

- “High Performance Web Sites” book by Steve Souders
Use YSlow

- Firefox extension made by Yahoo!
- Quickly checks your site for the Yahoo Performance Guidelines
- I’ll quickly go over a few server / infrastructure related rules ...
Minimize HTTP Requests

• Generate and download the main html in 0.3 seconds

• Making connections and downloading 38 small dependencies (CSS, JS, PNG, …) – more than 0.3s!

• Combine small JS and CSS files into fewer larger files
  • Make it part of your release process!
  • In development use many small files, in production group them

• CSS sprites to minimize image requests
Add an “Expires” header

• Avoid unnecessary “yup, that hasn’t changed” requests

• Tell the browser to cache objects

• HTTP headers
  
  • Expires: Mon, Jan 28 2019 23:45:00 GMT
  Cache-Control: max-age=315360000

• Must change the URL when the file changes!
Ultimate Cache Control

- Have all your static resources be truly static
- Change the URL when the resource changes
- **Version number** – from Subversion, git, ...
  /js/foo.v1.js
  /js/foo.v2.js
  ...
- **Modified timestamp** – good for development
  /js/foo.v1206878853.js
- **(partial) MD5 of file contents** – safe for cache poisoning
  /js/foo.v861ad7064c17.js
- Build a “file to version” mapping in your build process and load in the application
Serve “versioned” files

• Crazy easy with Apache rewrite rules

• “/js/foo.js” is served normally

• “/js/foo.vx.js” is served with extra cache headers

```apache
RewriteEngine on
# remove version number, set environment variable
RewriteRule ^/(.*\.)v[0-9a-f.]+\.(css|js|gif|png|jpg|ico)$ \/$1$2 [E=VERSIONED_FILE:1]

# Set headers when “VERSIONED_FILE” environment is set
Header add "Expires" "Fri, Nov 10 2017 23:45:00 GMT" \ env=VERSIONED_FILE
Header add "Cache-Control" "max-age=315360001" \ env=VERSIONED_FILE
```
Minimize CSS, JS and PNG

• Minimize JS and CSS files (remove whitespace, shorten JS, …)


• Add to your “version map” if you have a “-min” version of the file to be used in production

• Losslessly recompress PNG files with OptiPNG
function EventsFunctions() {
    this.get_data = function(loc_id) {
        if (this.TIMEOUT) {
            window.clearTimeout(this.TIMEOUT);
            this.TIMEOUT = null;
        }
        var parameters = 'auth_token=' + escape(global_auth_token) + ';total=5;location='+loc_id;
        var request = YAHOO.util.Connect.asyncRequest('POST', '/api/events/location_events',
            {
                success:function(o) {
                    var response = eval('(' + o.responseText + ')');
                    if (response.system_error) {
                        // alert(response.system_error);
                    }
                    else if (response.length) {
                        var eventshtml='';
                        for (var i=0; i<response.length; i++) {
                            eventshtml+='<br /><a href="http://example.com/event/'+response[i].id+'">'+response[i].name+'</a> - '+response[i].start_date;
                            if (response[i].start_time) eventshtml+=' '+response[i].start_time;
                            if (response[i].description) eventshtml+='<br />'+response[i].description;
                            eventshtml+='<br /><br />';
                        }
                        var le = document.createElement("DIV");
                        le.id='location_events';
                        le.innerHTML=eventshtml;
                        document.body.appendChild(le);
                        tab_lookups['events_tab'] = new YAHOO.widget.Tab({
                            label: 'Events',
                            contentEl: document.getElementById('location_events')
                        });
                        profileTabs.addTab(tab_lookups['events_tab']);
                    }
                    try{ pageTracker._trackPageview('/api/events/location_events') } catch(err) {} 
                },
                failure:function(o) {
                    // error contacting server
                }
            });
    }
}
function EventsFunctions(){this.get_data=function(loc_id){if(this.TIMEOUT)
window.clearTimeout(this.TIMEOUT);
this.TIMEOUT=null;}
var parameters="auth_token=\"+escape(global_auth_token)\";total=5;location=\"+loc_id;\"
var request=YAHOO.util.Connect.asyncRequest("POST","/api/events/location_events",null,
{success:function(o){var response=eval("\"\"+o.responseText+\"\"\")
if(response.system_error){}
else{
if(response.length){var eventshtml="";for(var
i=0;i<response.length;i++){
eventshtml+="<br /><a href=\"http://example.com/event/\"+response[i].id+\"/\">"+response[i].name+"</a> - "+response[i].start_date;
if(response[i].start_time){eventshtml+="
"+response[i].start_time;}
if(response[i].description){eventshtml+="<br /
"+response[i].description;}
}
eventshtml+="<br />";
}var
le=document.createElement("DIV");le.id=\"location_events\";le.innerHTML=eventshtml;
document.body.appendChild(le);tab_lookups.events_tab=new
YAHOO.widget.Tab({label:\"Events\",contentEl:document.getElementById(\"location_events\")});
profileTabs.addTab(tab_lookups.events_tab);}
try{pageTracker._trackPageview("/api/events/location_events");
}catch(e){}

Minimized JS
~1600 to ~1100 bytes
~30% saved!
Gzip components

• Don’t make the users download several times more data than necessary.

• **Browser:**
  Accept-Encoding: gzip, deflate

• **Server:**
  Content-Encoding: gzip

• **Dynamic content (Apache 2.x)**
  LoadModule mod_deflate ...

  AddOutputFilterByType DEFLATE text/html
  text/plain text/javascript text/xml
Gzip static objects

- Pre-compress .js and .css files in the build process
  `foo.js > foo.js.gzip`

- AddEncoding gzip .gzip

  # If the user accepts gzip data
  RewriteCond %{HTTP:Accept-Encoding} gzip

  # … and we have a .gzip version of the file
  RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME}.gzip -f

  # then serve that instead of the original file
  RewriteRule ^(.*)$ $1.gzip [L]
remember

THINK HORIZONTAL!

(and go build something neat!)
Books!

- “Building Scalable Web Sites” by Cal Henderson of Flickr fame
  - Only $26 on Amazon! (But it’s worth the $40 from your local bookstore too)

- “Scalable Internet Architectures” by Theo Schlossnagle
  - Teaching concepts with lots of examples

- “High Performance Web Sites” by Steve Souders
  - Front end performance
Thanks!

- Direct and indirect help from ...
  - Cal Henderson, Flickr Yahoo!
  - Brad Fitzpatrick, LiveJournal SixApart Google
  - Graham Barr
  - Tim Bunce
  - Perrin Harkins
  - David Wheeler
  - Tom Metro
  - Kevin Scaldeferri, Overture Yahoo!
  - Vani Raja Hansen
  - Jay Pipes
  - Joshua Schachter
  - Ticketmaster
  - Shopzilla
  - .. and many more
– The End –

Questions?

Thank you!

More questions? Comments? Need consulting?

ask@develooper.com

http://develooper.com/talks/

http://groups.google.com/group/group/scalable