

# RailsConf Europe 2008 Berlin Notes

by Marston Alfred ([marston@sugarstats.com](mailto:marston@sugarstats.com))  
<http://www.marstononline.com/>

## Re-Factoring Rails Notes:

By Trotter Cashion ([cashion@gmail.com](mailto:cashion@gmail.com))

- rCov – Test Coverage
  - `gem install -y rcov`
  - `rcov test/unit/*.rb test/functional/*.rb`
  - Standard Rails Test Generation doesn't cover all code, don't trust it.
- Nearly impossible to re-factor if you don't have good tests.
- Try to get as much test coverage of your app as possible.
- Clean code makes it really easy to cache information and variables
- We don't re-factor because of deadlines – You have to say “No, it isn't ready yet.”
- Explain it to managers in a business sense/scenario.
- People don't re-factor because they don't write tests
- What to not re-factor?
  - When adding new features? (in the initial moment when adding the feature. Re-Factor AFTER the tests pass and it **works** then re-factor.
  - When Killing Bugs.
- What tools do you need?
  - Use rCov (test coverage gem) – Tells you what percentage of your code that your tests are covering and which percentage it isn't. i.e: 19% of your helpers aren't being tested.
  - Use auto test tools (ZenTest/AutoTest) – Runs tests for you all the time.
  - Use grep? (Textmate can do project wide search).
  - SpiderTest – Spiders your views, clicks on links. If something fails you will know. Great for view testing.
  - Selenium – Like SpiderTools for view testing.
- Use `before_filters` everywhere, make things simpler and better.
- Though if the `before_filter` only applies to one method, move it into the method.
- CRUD Re-Factoring:
  - Test, Make Test Break, Test the Method.
  - Common naming: `test_new`, `test_create`, `test_destroy`, `test_update`

- Don't use "pubsh\_with\_attributes" Deprecated. Used to be used with has\_and\_belongs\_to\_many
- Remember YAGNI (You ain't gonna need it). You think you'll need it, but you really don't.
- Basic 4 rules for re-factoring:
  - Move attributes and associations – When an attribute is being used more by another class. Many times, the need to load the current class just to retrieve this one attribute is a drain on performance. Removing/Moving associations removes middleman and costly joins.
  - Move Methods (in models) – When a method uses more attributes from a different class than it does from its defining class.
  - Move Methods (in controllers) – When the method uses objects/attributes typically governed by another controller.

## Scaling a Rails Application Notes:

- Questions like: How scale from 1 DC to multiple Data centers across the globe and still maintain a unified back-end and only have my costs multiple by 1x per DC.
- Slingshot about making a Rails app an offline desktop app.
- TextDrive deploys 10 new servers every 2-4 days.
- "Scalability" independent of "performance". Things can be slow and still scale.
- Rules of Ten:
  - Tiers are different functionally
  - Tiers should be 10x different in throughput
  - Infrastructure costs  $\leq$  10% of "revenue"
- Web Apps are stateless – Of course.
  - Inherently "scalable"
  - LB -> App -> Back-end
- Types of Scalability:
  - Load – If I have 1 thing that is 100% utilized, if I add 9 more then I should have 10 things that are 10% utilized
  - Geographic – Can I separate things across the world and have it scale to each in a balanced way.
  - Administrative – Most expensive, as we scale hardware/software can I scale the people needed to maintain them in a cost effective manner.
- How many servers fit into 100kw? Around 250-400 servers.
- 1Gbps = 125MB/s, 100Mbps = 12.5MB/s
- 100Mbps usually around \$5000-\$8000/month. (\$50-\$80 per MB per Second)

- BigIP's can do 20k-100k RPS easy. Apache, Lighty or LSWS can do 1k-15k proxy static request/s
- Google spends 10% of their revenue on server/hardware/infrastructure. That includes CC processing charges. \$307 million in 2006.
- I.E: if you're making \$1 million a year, you should be spending \$100,000/yr on infrastructure.
- Joyent builds in racks of 3 with a 4<sup>th</sup> admin app. 1 is app rack, storage rack and multi-core rack for processing heavy things.
- Keep things standardized, simple and use open technologies.
- You need to understand what a given "component" can do at a minimum and maximum (i.e a single server, a single VPS, a single proxy)
- It is generally cheaper at the \$20-30k/month spending range to do hosting "in-house" i.e buy your own hardware etc, collocate. Assuming you have a good network admin and know what you're doing.
- Joyent used to do all their hosting at "The Planet". Had problems, bad data centers. "It looks like a robot vomited."
- If you run your own hardware, keep one kind of:
  - Console server
  - Switch
  - Server
  - CPU
  - RAM
  - Storage
  - Disc
  - OS
  - Interconnect
  - Powerplug
  - Powerstrip
- Color code cabling
- They had SAN, got rid of it for ZFS.
- People don't know how to scale DB's and don't know how to store files.
- Level 3 and Equinix good tier1 providers.
- Typical costs:
  - \$500-\$750 for a rack
  - \$500-1k for power per rack
  - \$1k for bandwidth
  - \$4k for systems
  - A great sysadmin/racker is \$100k+ (hah!)
  - \$6500 for 20 systems in a rack on a lease.
- What do you run on servers? VIRTUALIZE!! (Xen, VMware, Solaris Zones)
- VMWare is around \$2k per server. Better for "Administrative Scalability".

- Do you “get” linear performance? Typically no, usually take a 10-20% performance hit. Things are getting better.
- Lessons learned:
  - Write separate apps that do one thing and talk to each other via web services.
- Typical Server Setup:
  - 1x – 16GB, 4 Core AMD Box
  - 4 VPS’
  - 10 Mongrels per CPU/Core (10 per CPU)
- Scale Process:
  - Run more and more processes
  - They should add up, in the front and in the back. Should add up linearly.
- In Front: Load Balancers (Hardware). In Back: DB Middleware (Sequoia)
- DNS Federation: Easy way to split users into pods. User.dns.com instead of dns.com/user
- BingoDisk does this. Use PDNS (Power DNS). Run on FreeBSD. MySQL Backend. Master/Slave MySQL instead of Master/Slave PDNS.
- Joyent uses “predictive” on Mongrels.
- Layer7 Load Balancing is great. Separate mongrels for different parts of app. i.e: signup/login pages served faster.
- Wordpress.com does DNS Federation.
- When doing stress testing, use ab or httpperf. Us against ONE mongrel. Base it off that. Then try against 10. Base your LB against that.
- Software Load Balancers:
  - Varnish – 10k RPS
  - BigIP: 100k RPS
  - Nginx/LSWS: 1k RPS
  - HA Proxy: ?
- Varnish also does caching. On some apps, server 80-90% of requests from cache, good like squid.
- Varnish has VCL language too. Makes it easy to config via web services. Purge cache from web service for example. Easy to purge cache via Rails app.
- Nginx: Good for LB + Static Content.
- Nginx -> Mongrel -> MySQL = 1k RPS Easy
- Vanish -> Nginx -> Mongrel -> MySQL = 10k RPS w/ Varnish Cache to server lots of requests.

- Use “Browser Pipelining” via several static servers with different domain names.
- BingoDisk not so good for small asset storing/serving. 2GB uplink. Good for large files.
- Offload static content to CDN at the 20k/rps area.
- Memcache helps you “deal” with Scalability in the backend.
- Use Squoia as a DB middleman for DB replication. Makes MySQL DB dumb. “RAIDED Databases.” You can program DB sharding with it.
- Nginx/LSWS good up to 1000 RPS, then Hardware Load Balancers are needed.
- Static server naming: (assets1-4.domain.com, assets5-8.domain.com)

## **ActiveRecord and SDO Notes:**

- SDO and AR are similar
- Allows developers to focus on things that are useful.
- SCA’s are like SDO but include encryption, auth etc
- SDO = Service Data Objects
- SCA = Service Control Architecture
- Boring... Leaving.

## **Talks with EngineYard:**

- Switched from CoyotePoint LoadBalancers to using LVS Open-source project. Much more flexible.
- CoyotePoint just couldn’t keep up with their specific setup. Thousands of static IP’s pointing to many clusters. It just wasn’t what they were setup for.
- MySQL Setup is Amazing. 4 Sata HD’s per server dedicated to MySQL. 2x in RAID1 for Master, 2x RAID1 for Slave. Then syncs to bigger replication slices.
- 3GB MySQL Slices, also have 6GB Slices and dedicated slices/servers for bigger sites. Also backups go to SAN.
- Uses GFS with the CoRaid SAN’s.
- CoRaid has 16 and 24 drive SAN’s. 10Gbps Ethernet connections. AOE: Ethernet without the TCP overhead. Uses standard SATA drives, want more stored? Just add more disks. They have MANY CoRaid’s in their racks.
- 6x 1Gbps Ethernet ports per server.
- 2x (2Gbps) dedicated to SAN Storage per server.
- Dedicated, redundant servers for Mail, DNS and SVN also
- eXtreme Networks Super High performance Switches in between servers and SAN.
- Use Cacti / Nagios for Monitoring. Also Monit.

- Custom Gentoo Setup for Application Slices. 640MB each, ONLY for mongrels. Bare-Bones setup just enough to serve the app. 4 x Mongrels per slice. Event-driven if needed. 100% utilization of 1 core. Usually 8 or 16 core CPU's per server.
- 2 production slices (Load Balanced) can easily do 80RPS (4.5 Million DYNAMIC requests per day). Static files quickly served by Nginx.
- Redundant, Dual Switched Servers.
- Dual Network Connections to Public Internet

## Really Scaling Rails Notes:

(Alex Payne)

- Setup:
  - Big IP -> Apache mod\_proxy\_balancer -> Hundreds of Mongrels -> A big ass MySQL DB
- They setup mod\_proxy to send only 1 request at a time to mongrels
- If all their mongrels are taken users get an error page. (usually not good). At some point you'll have scaled out to know how many request you can get and have enough mongrels to serve that volumn.
- They use Benchmark.measure to calculate the runtime for any action.
- The show\_runtime shows the amount of time spent in DB, time spent in rendering etc.
- Shows an HTML comment at the end of each Twitter page. Easy to narrow bottlenecks.
- Never over-architect. Easy to want to tackle scaling problems early when they don't matter.
- You need a community first in order to scale.
- You need to abstract long running processes to Daemons.
- Use Queuing to send instructions from Mongrel to Daemon.
- They wrote a distributed queuing system called "Starling"
  - Speaks to memcache language
  - Transactional playback
  - Fast, simple, 100% pure ruby
  - They might release it (Twitter). Hope to open-source it within the month (Oct 2007)
- DB 101 –
  - Index everything you query on (if there is a WHERE)
  - Avoid complex joins
- Cache Cache Cache
- Use Monit to kill processes if they get too big.
- Scale where it matters, obviously. Find where the users need to spend their time.

- They use Munin, Nagios. They loves graphs, graphs are great.
- Use the community. Create and API
- They use EDGE Rails (using Piston).
- Memcache roundtrip time is about 20ms.
- Actively expire memcache, not general sweepers.
- They use pure Ruby mem-cache. No fragment caching.
- No DB Connection Pooling
- 1 Big master DB. 3 slaves: 1 slave for SolR Search, 1 for stats aggregations. 1 for hot-backup.
- About to partition their DB into shards based on content.

## Rubinus Notes:

- Is a Virtual Machine.
- Brand new code base. Written in C/Ruby. No patches.
- Nov 3: 1.0 preview. End of 2007 1.0
- Fairly backwards-compatible.
- Much better memory usage. Works to relieve memory leak pain. Garbage collector is new and very simple.
- Addresses the forking problem. i.e: Having lots of Mongrel processes. Handles children better.
- Works with the OS so that forked children share data.
- Rubinus doesn't run Mongrel yet.
- You can pre-compile your projects. Somewhat obfuscate code.
- Doesn't use any existing Ruby 1.8 code.
- When Ruby 1.9 comes out, they'll be able to commit in a majority of the features.
- Improves on Error Reporting. – Very flexible. Completely override backtrace.
- Leaves the door open for many performance improvements after 1.0. inline-caching, pre-compiling code, JIT (eventually) etc.
- When C-extensions misbehave it brings the whole Ruby VM down.
- Improves on Profiles to do performance testing of Ruby/Rails apps. In Ruby 1.8 it gives a big performance hit.
- In Rubinius they built in a sampling profiler and has little performance hit. Almost no impact on runtime.
- Rubinus vs Yarv (Ruby 1.9): trying to avoid this. BUT... Not much progression with YARV. Been promised for 3 years now.
- JRuby share the same RSpecs. IronRuby doesn't to their knowledge. Yarv isn't either to their knowledge.
- Communicating with Ruby Core is VERY difficult.
- Rubinius is faster now in MRI(Ruby 1.8) in many micro-benchmarks.

- They have no windows developers yet, actively seeking. ([ephoenix@engineyard.com](mailto:ephoenix@engineyard.com))
- You can run multiple VM's in the same process.
- You can embed it fairly easy. No shared data in the C part of it.
- Currently has a green thread model.
- You have access to low-level syntax tree as well.
- 100% compatible with existing C extensions. They have a layer called "sub-10" to do this.

## Ruby on Rails Best Practices Notes:

(Marcel Molina Jr, Michael Koziarski) [www.therailsway.com](http://www.therailsway.com)

- Controller with more than 6 or 7 actions with more than 6 or 7 lines each, you're doing something wrong.
- Huge actions in Controllers not so good. Common Rails app problem.
- Think of controllers as moderators to the people speaking (in this case the Models). Do whatever you can in the Model.
- Strive for the very "Skinny Controller" and very "Fat Models"
- Try to create custom finders in the model instead of inline Model Finds in the controllers. Make them reusable.
- Every line of code in a method should be **AT THE SAME LEVEL OF ABSTRACTION** – Keep low level details out of controller methods, keep it at a high level of abstraction.
- Use "Model.association.find/create etc" instead of individual Finds for different models that are related and have associations. AR Magic, use it.
- Custom, re-usable finds are your friend.
- In models, you'll want to "extract" functions into higher level, clearly named methods. Easier to understand than parsing the actual login. Call method instead of writing the specific logic. It is also re-usable. Gives clarity of thought and is more simplified. You keep a smaller buffer of things to remember after parsing specific pieces of logic.
- :with\_scope is usually not needed, 9 times out of 10. They will make it harder to use in Rails 2.0
- Though :with\_scope looks cool and easy to use at first, it makes things very hard to understand what is going on when things get bigger. Makes it hard for new developers to pick up off the code, you're being "Too Clever."
-

## Building Rich Internet Applications with Flex and Ruby on Rails Notes:

- build advanced UI's using flex.
- RIA's = Rich Internet Applications
- 2 methods: DTHML & Ajax and Plugin based (Flash, Silverlight etc)
- Flex based off Flash Player 9 and Actionscript 3. New VM in Flash 9
- Flex is a programming language. Based off NXML and Actionscript. Uses Swif.
- Flex 3 coming out.
- AVM2 (new VM in Flash 9) built for new rich apps. Aimed at performance and speed. Supports full runtime error reporting, built-in debugging, binary socket support. Also backwards compatible with AVM1 for backwards compatibility.
- Old apps based on AVM1 (Written in Actionscript 2) running in AVM2 (Written in Actionscript 3) also get speed benefits.
- AVN2 is open-sourced to Mozilla. Will be built into Firefox 4 (JS2, modified version of Actionscript). Tamarin Project.
- AS3 fully OO programming scripting language.
- Flex SDK cross-platform dev framework. Flex 3 SDK will be opened-sourced.
- YouTube uses Flex. Mixbook is a Flex app (uses Rails as a backend).
- Buzzword an online word processor built in Flex. Very dynamic and fluid UI.
- Yahoo local mapping engine is done in Flex 2
- Connecting Rails to Flex, 3 components in the SDK
  - HTTPService
    - Little requirements
    - Widest variety of connection options
    - Offers little in options of data return, doesn't offer much control.
    - Allows you to connect to RESTful interfaces.
    - Use XML as an interface. Respond\_to – Feed that from Rails to Flex.
    - FlexBuilder for Eclipse is a great plugin as a Flex IDE.
    - Binding in Flex is handled in the framework.
    - Basically use REST calls from Flex and put/manipulate the info WITHIN the Flex UI.
    - Converts XML into Actionscript Objects within Flex
    - Downside:
  - RPC Object
    - Use SOAP based XML wrappers.
    - Publish in WSDL
  - RemoteObject Component (Best option)
    - Client side component
    - Standard requests over http and https
    - Packaged like SOAP but descriptors are binary

- Smaller Data Payload
- Faster Net Transfer
- Typed Object
- Requires Flash Remoting Gateway on Server
  - Server Side Endpoint for Communications
  - Implemented in many languages
  - Uses SSI – Java, CF, Ruby, Rails, PHP and .NET
- WebORB for Rails is a good Rails plugin for using RemoteObject to integrate Flex and Rails. By “Midnight Coders”. Written by Java guys, made like a java app.
- WebORB uses AMF for data transfer.
- WebORB lets you communicate between Rails/Flex like one app.
- RubyAMF is also good. Open Source, 1 developer. Rubyamf.org
  - RubyAMF, same features as WebORB but 20-30% **FASTER**
  - Can be used with or without Rails
  - MIT License
  - Uses respond\_to in controllers to specify correct response format.
  - Installed as a plugin.
- Flex.org – For Ruby Developers. Lots of tutorials / Docs.
- [simeon@simb.net](mailto:simeon@simb.net) - Simeon Bateman
- [blog.simb.net](http://blog.simb.net)
- [simb.net/blog](http://simb.net/blog) (older none-migrated content).

## Creating Hybrid Web + Desktop Apps with Rails and SlingShot:

- Allows full sync with desktop version
- Removes latency of Internet, duh.
- Distributes processing to the edge of the network (users computer, ala Flash)
- Advanced Sync with peer to peer serviing. Even create “micro-lands” on an application level.
- No address bar, no backbutton.
- Open Source.
- SlingShot:
  - Mac + PC Desktop app
  - Rails Plugin
  - Ruby + Mongrel Runtime
  - Set of conventions
  - GPLv2 open sourced

- Sync DB + Filesystem
- Use same code online + offline.
- Drag + Drop functionality.
- Competitors:
  - Silverlight
  - Adobe Flex
  - Google Gears
  - Firefox 3
  - Custom desktop app + REST API
- Architecture:
  - Client side Rails Server. Communicates with your servers via XML/HTTP
  - You package your app in the SlingShot container. Need to be ok with showing your code.
- Radient CMS has a slingshot version. (interesting)
- Syncs at the DB level. Uses SQLite3 locally
- Provides hooks to write your own sync code
- You need to use Rails Ruby Wrappers for SQL.
- Need to have created\_at and updated\_at are added to the models you want to sync.
- D/L Dragging out of the browser only works in OS X at the moment
- SlingShot Doesn't:
  - Sync conflict resolution –
  - Encrypt data
  - Domain-specific on/offline issues
  - Package your app automatically
  - Update yourself + Rails app code

## **Development Case Study: MindMeister**

- Online Mind Mapping Programming.
- Lots of Ajax/JS. Nice interface.
- Easy to collaborate, share and use with others.
- iPhone interface
- Started in 2006 Had idea. Prototype and Started implementation in mid 2006.
- Went live May 2007 – 1 Month, 30,000 users.
- Based on HTML/ CSS / JavaScript / Ajax only. Enabled real-time collab (with effects).
- Choosing name took 6 months.
- Private BETA start in Feb 2007. Lots of benefits.
- Freemium model as well.
- Feb 2007 had their 1000<sup>th</sup> user. Lots of traffic from Delicious.

- March 2007 – Lots of competitors started coming up. 6-8. Helped to be first.
- 1-2% premium user rate. (1,500) as of Sept 2007. (1 month later).
- Lessons learned:
  - Kept it simple. Better design and usability.
  - Spent lots on euros and it was worth it.
  - Use a GREAT designer.
  - Target non-technical users.
- With share functionality they went through many iterations
- Marketing:
  - Write to bloggers
  - UI – Spent lots on this.
  - Post to app portals.
  - Create an API.
  - Write regular newsletters and blog posts.
  - Add to delicious button
  - SEO
  - Be generous with Premium accounts
  - Don't do:
    - Comment spam
    - Didn't try to pay for ads / placements
  - Carefully select premium features
  - Keep “sharing” features free
  - Announce maintenance windows, be transparent. Explain outages and be honest.
  - Be responsive with support requests.
- Use CC base subscription. Use wirecard for processing.
- Use prototype and scriptaculous for all JS, lots of customization.
- In the beginning just 2 people. Now they have 3 developers plus the 2 founders.
- Yearly subscriptions. Don't currently deal with CC rejections
- Use Ajax to poll back to the server every 5 seconds.

## **Outsourcing to Open-source:**

- Toby, from Germany. Shopify founder and Rails-Core member.
- Created liquid. A fluid and safe template language.
- Also made ActiveMerchant – Ruby Gateway API for CC / Payment Processing
- 2 OpenBSD poxes. Debian cluster from Nginx -> HA Proxy -> Mongrel
- Use Solr for search.
- 25,000 shopify stores. All indexed by spiders and public facing
- Use memcache, works great.

