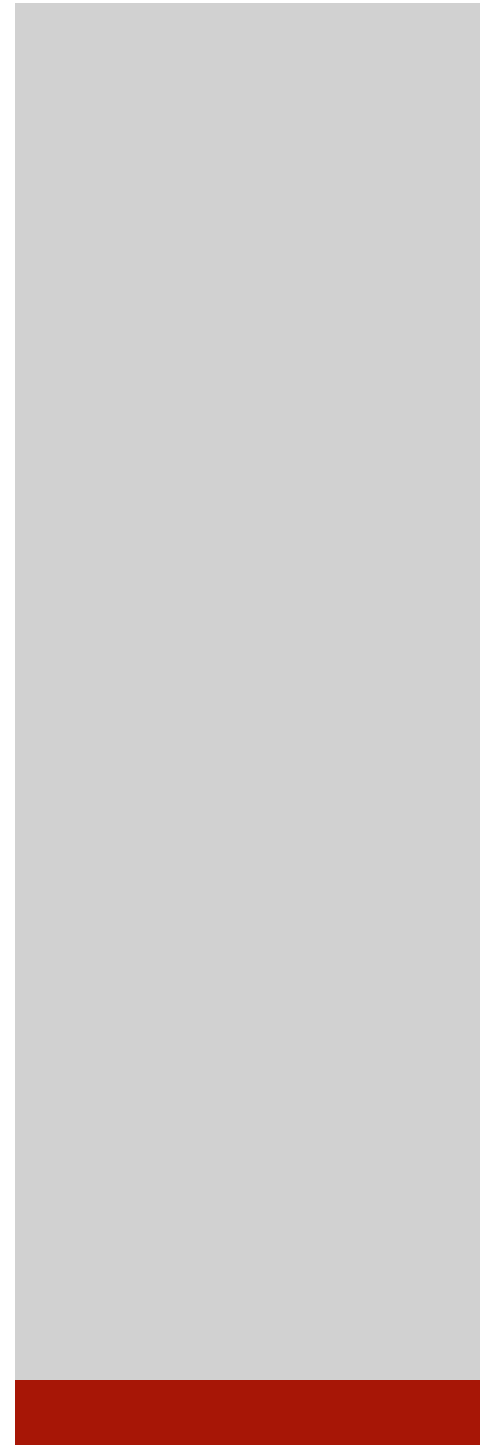




Rails Patterns

Jonathan Weiss, 10.06.2008
Peritor Wissensmanagement GmbH



Who am I?

Jonathan Weiss

- Consultant for Peritor Wissensmanagement GmbH
- Specialized in Rails, Scaling, and Code Review
- Webistrano – Capistrano Web-UI deployment tool
- FreeBSD Rubygems and Ruby on Rails maintainer

<http://blog.innerewut.de>

Patterns

Patterns

A Definition:

Each pattern **describes a problem** which occurs **over and over** again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution a million times over, **without ever doing it the same way twice**.

Alexander et al.

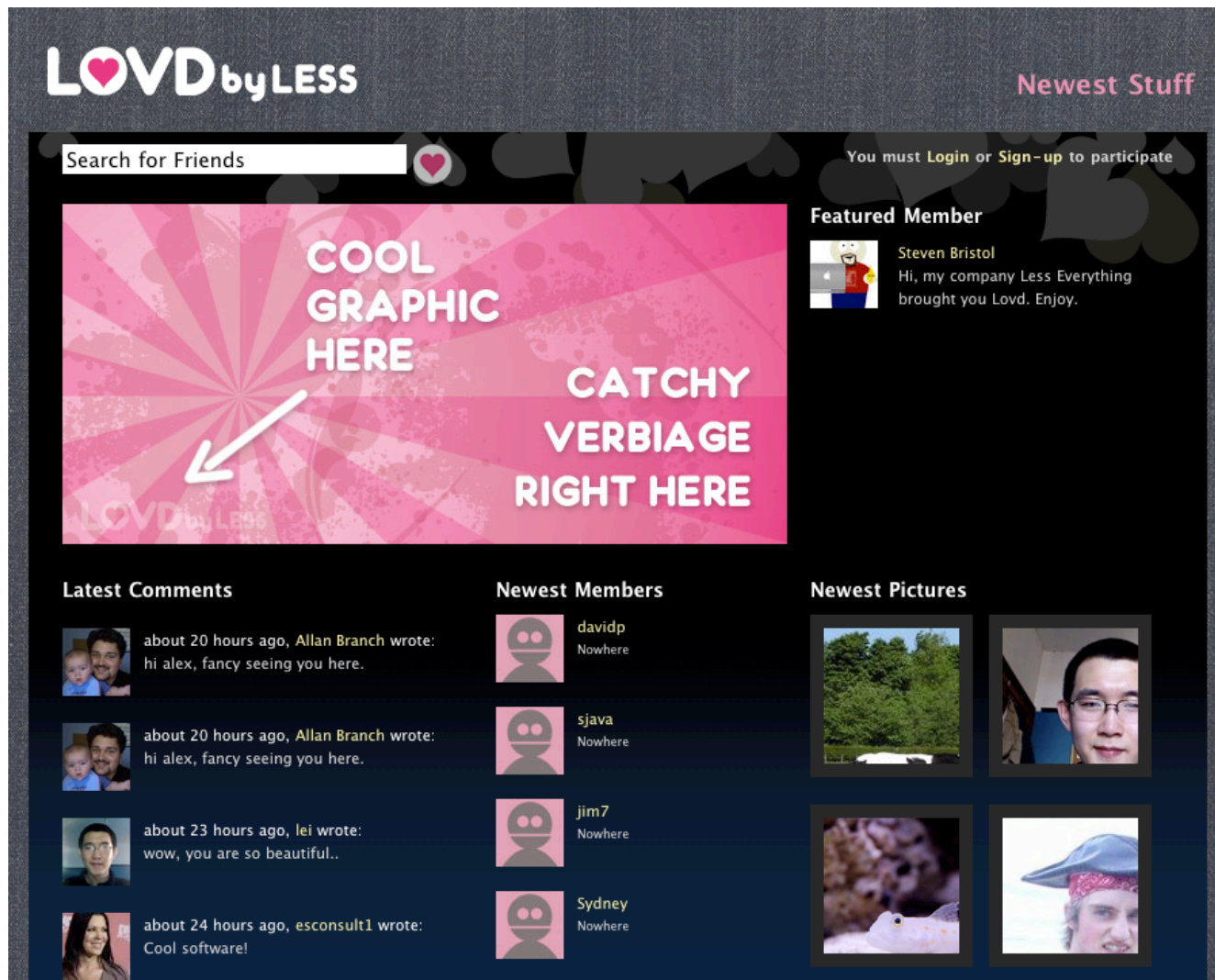
Patterns

Characteristics:

- General reusable solution to a commonly occurring problem
- Depend heavily on the context
- Not a final solution, but a template for a possible solution

Agenda

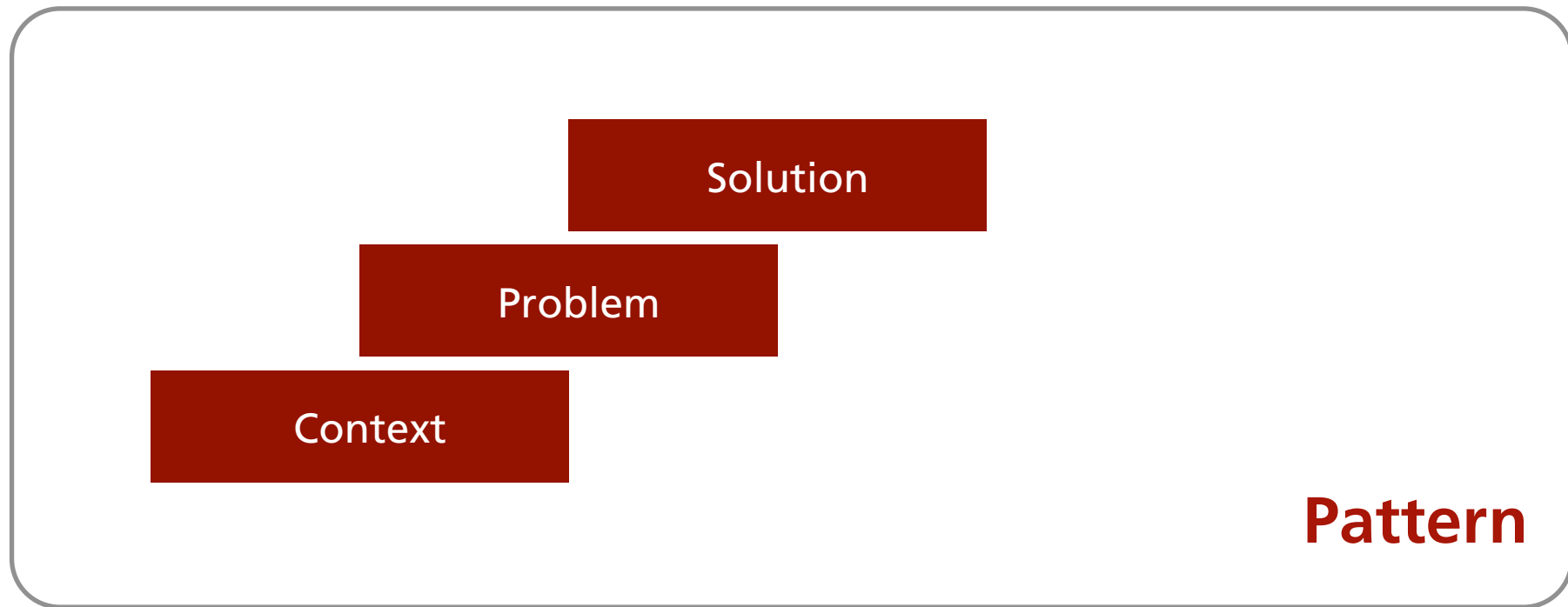
Your Social Network v.238



Typical issues:

- Media conversion
- User data
- Lots of Plugins
- External dependencies
- Slow requests

Approach



Media Conversion

Scenario

**Your application handles and converts media files
(images, audio or video)**

Scenario

Your application handles and converts media files (images, audio or video)

Typically, the conversion is done “inline” after saving the file:

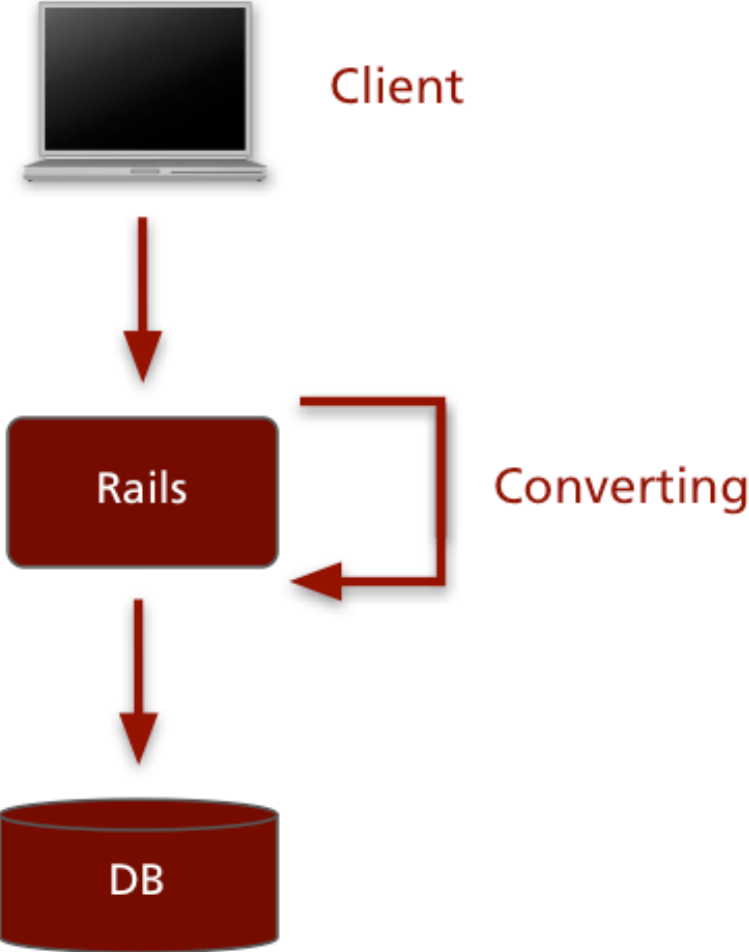
```
has_attachment :thumbnails => { :thumb => [50, 50], :geometry => 'x50' }
```

Or

```
after_create :convert

def convert
  system("ffmpeg -i #{public_filename} -ab192k #{public_filename}.mp3")
end
```

Scenario



Problem

Converting media files takes time...

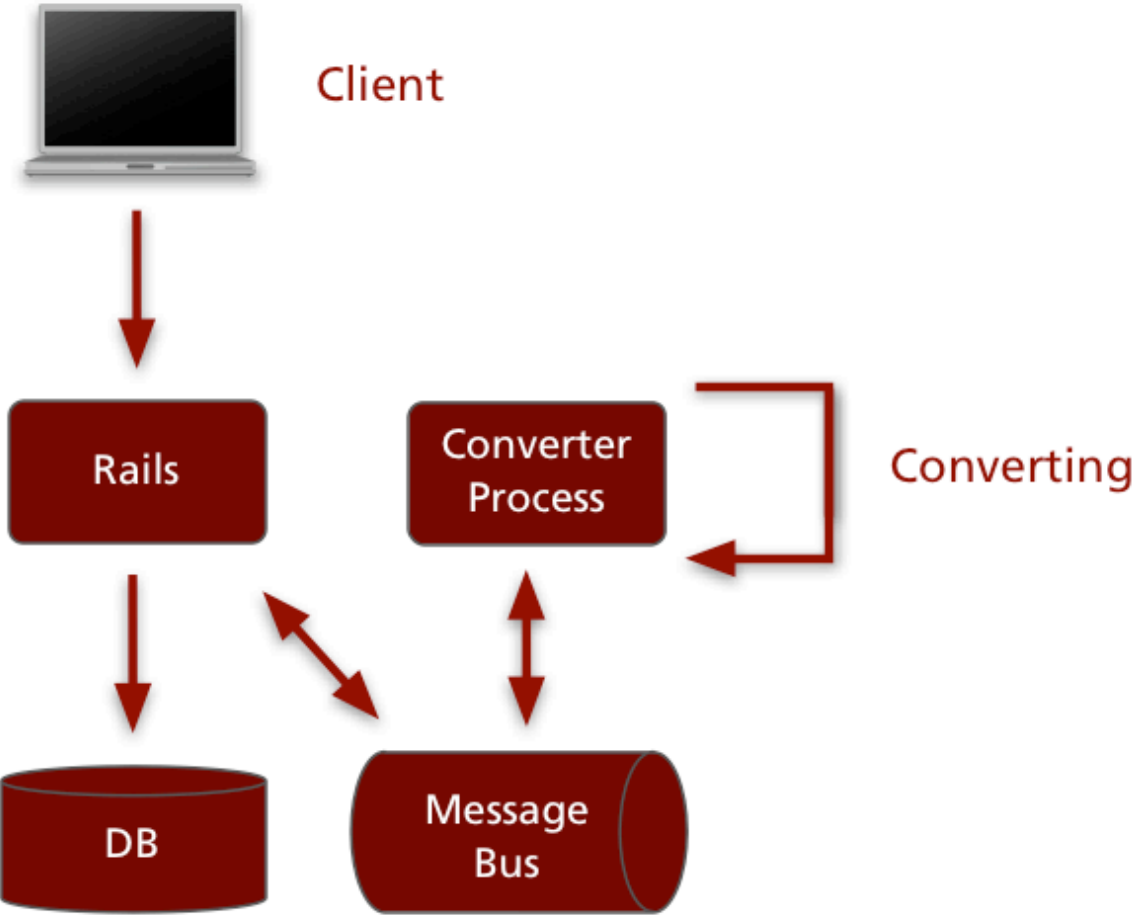
- One Rails process is blocked – no further requests will be processed
- Long-running HTTP/TCP connections can be dropped by upstream proxy servers
- The user gets no real feedback
 - Blank “post” screen on HTTP POST
 - Image spinner on Ajax POST
- All extra resources needed for the conversion (CPU, RAM, libraries) are consumed on the application servers

Solution

Process media files asynchronously

- After the upload, just save the file and create a “processing request”
- A second process will process/encode the file after receiving the request

Solution



Pattern

Separate long-running and resource-hungry tasks

- This separation will make your application more robust and user friendly.
- Use a message bus to communicate with the background process.

Options for message bus:

- Database
- Amazon SQS
- Drb
- Memcache
- ActiveMessaging
- ...

Options for background process:

- (Ruby) Daemon
- Cron job with script/runner
- Forked process
- (Backgrounddrb)
-

Database/Ruby daemon example

```
class Audio < ActiveRecord::Base
  has_many :jobs, :as => :processable
  has_attachment

  after_create :create_conversion_job

  def create_conversion_job
    self.jobs.create!
  end

  def convert
    system("ffmpeg -i #{public_filename} -ab192k #{public_filename}.mp3")
  end

  def process!
    convert
  end
end

# run method of Ruby daemon
def run
  loop do
    job = Job.next
    unless job.blank?
      job.process!
    else
      sleep 30
    end
  end
end
```

```
class Job < ActiveRecord::Base
  belongs_to :processable, :polymorphic => true

  def self.next
    next_job = find(:first, :conditions => ["status IS NULL"], :lock => true)
    if next_job
      next_job.mark_in_process!
    end
    next_job
  end

  def mark_in_process!
    self.status = "in_process"
    self.processed_at = Time.now()
    self.save!
  end

  def process!
    begin
      self.processable.process!
    rescue => e
      self.status = 'failure'
      self.status_description = "#{e.message} - #{e.backtrace.join("\n")}"
    else
      self.status = 'processed'
    end
    self.completed_at = Time.now
    self.save!
  end
end
```

Handling User Generated Data

Scenario

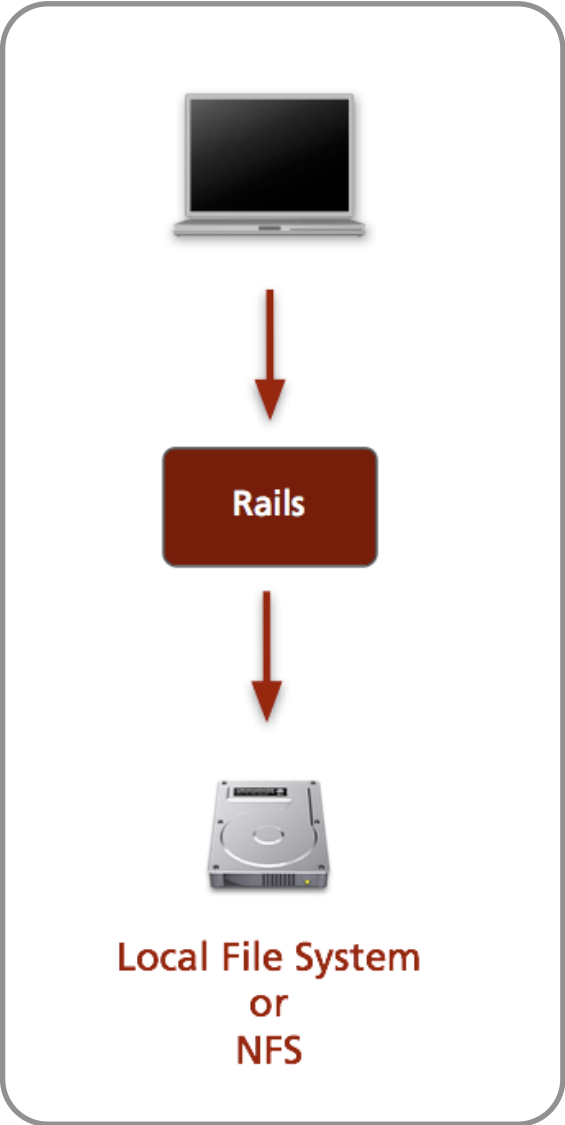
Your users upload lots of data (documents, videos, images, ...)

Scenario

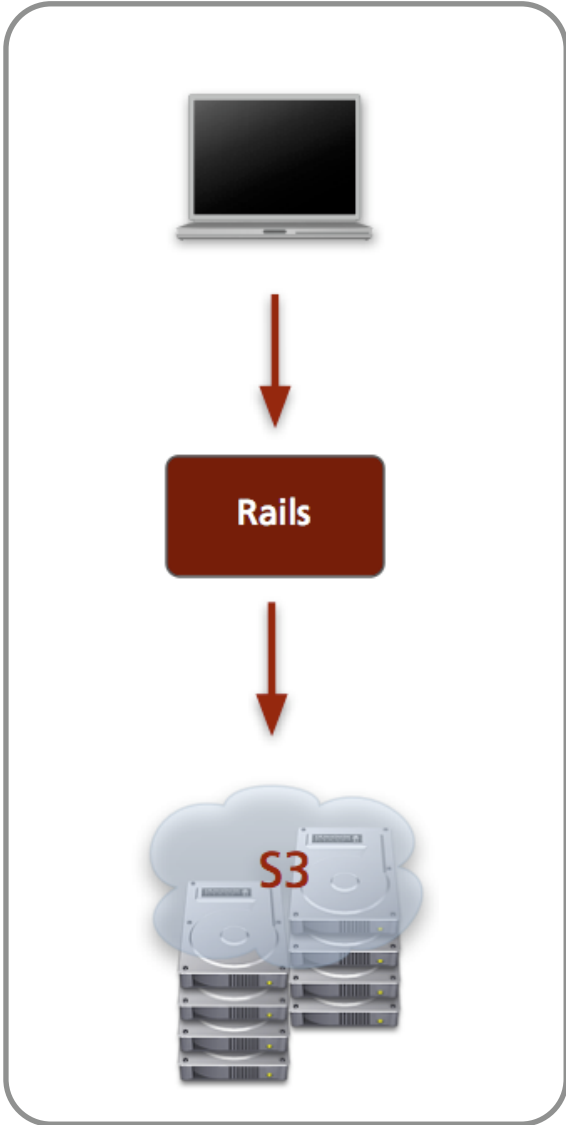
Your users upload lots of data (documents, videos, images, ...)

- Usually you start with storing the data locally on your server
- When several servers are in place, NFS or Amazon S3 is used as a shared space

Scenario



or



Problem

Pure NFS or Amazon S3 data backends have their problems:

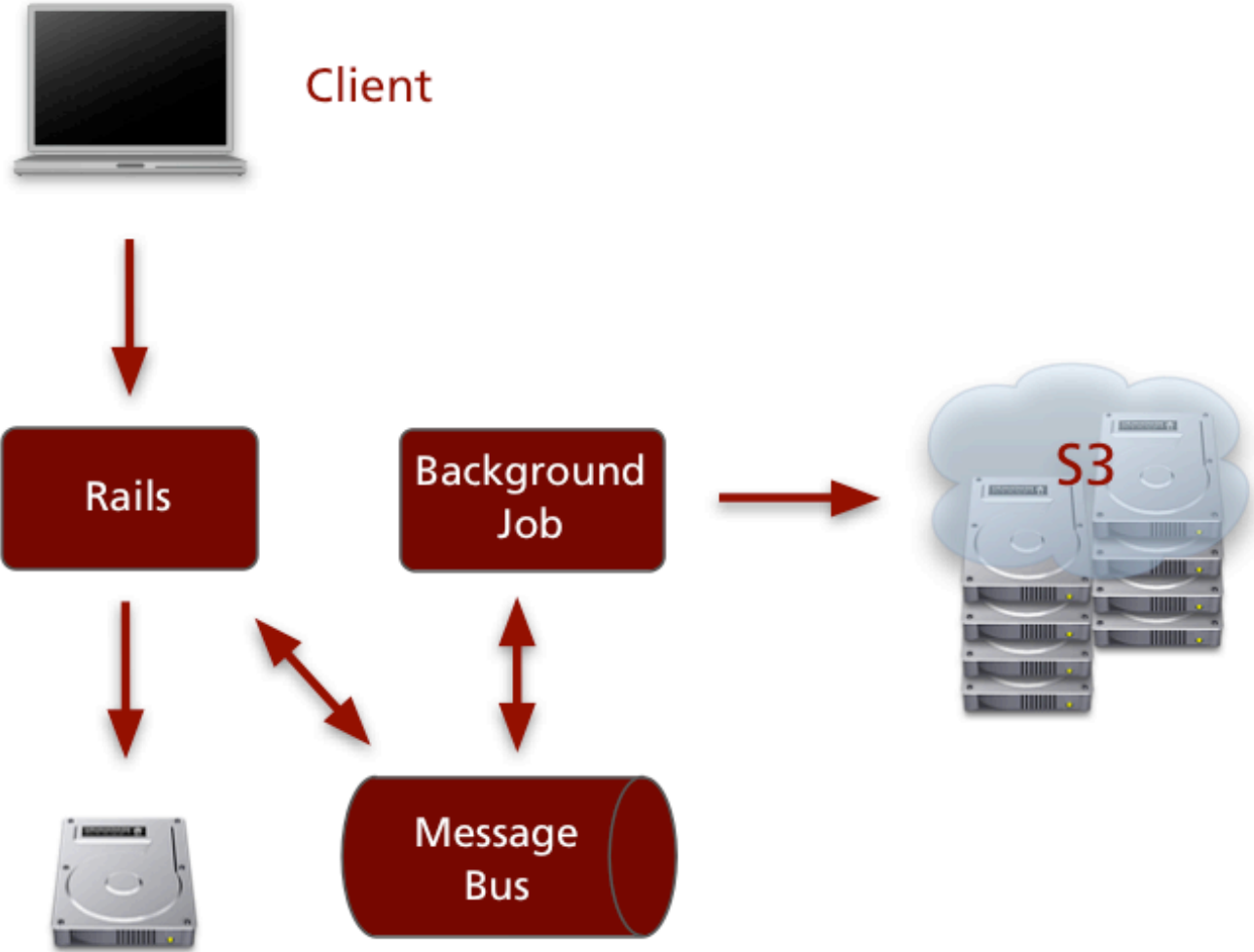
NFS

- Does not easily scale for many servers
- Slow for a “write heavy” application
- Single point of failure

S3

- Uploading to S3 is slow
- Background processing jobs need to fetch from S3 and upload again

Solution



Pattern

Combine file system and S3 backend for user data

- Use one as the cache of the other
- Push to S3 in the background
- Access files on NFS if present, if not use S3

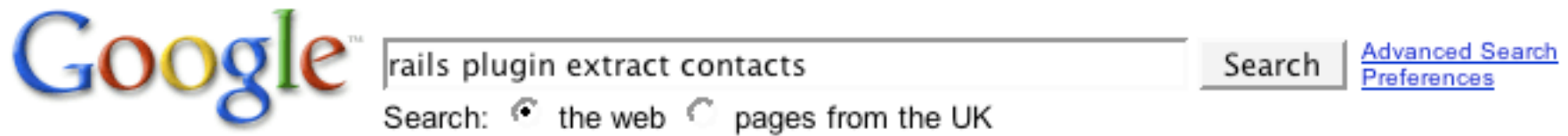
Plugin Mania

Scenario

You want to extract contracts from users email accounts

Scenario

You want to extract contracts from users email accounts



```
$ ruby script/plugin install .....
```

Problem

Often you get unmaintained, inflexible code

Many plugins are of poor quality and unmaintained after a couple of months

Most plugins are written to solve the authors specific problem and are hard to adjust to your own requirements.

Event popular plugins have problems:

- Backgroundrb
- acts_as_ferret
- restful_authentication

Solution

Do not blindly use plugins

- Always audit the code
- Look for stable and mature plugins
- Watch for plugin updates and announcements
- Consider writing code yourself

Pattern

Choose your dependencies wisely

- Every new library is potentially a new burden
- Always review and test new dependencies
- Writing custom code is more often than you think the right thing to do
- At least make yourself familiar with added code

External Dependencies

Scenario

Your applications has several external dependencies

You depend on:

- Rails 1.2.3
- Memcache-client 1.5.0
- FasterCVS 1.2
- Ruby DBI

Those dependencies are installed by different means:

- Rubygems
- OS package manager
- In vendor/plugins

Problem

Your deployment becomes brittle

As soon as you deploy the first time or install a new server, everything breaks:

- The server administrator installed Rails 2.0
- Some gems are missing
- The needed library is installed, but it has the wrong version

Solution

Vendor everything

Put all needed libraries and dependencies in /vendor

Rails in vendor/rails

```
$ rake rails:freeze:gems
```

Gems in vendor/gems

```
config.gems "fastercvs", :version => '1.2'  
  
$ rake gems:install  
$ rake gems:unpack GEM=fastercvs
```

Pattern

Freeze your dependencies

In order to archive a stable and reproducible platform, package all dependencies

Use a combination of

- /vendor
- OS packages
- Custom packages (.deb/.rpm)

And keep the packages in a save place (e.g. version control)

Handling slow requests

Scenario

Your site allows people to upload and download files

Scenario

Your site allows people to upload and download files

Rails is single-threaded and a typical setup concludes:

- Limited number of Rails instances
 - ~8 per CPU
 - Even quite active sites (~500.000 PI/day) use 10-20 CPUs
- All traffic is handled by Rails

```
<Proxy balancer://rails_cluster>
  BalancerMember http://127.0.0.1:5000
  BalancerMember http://127.0.0.1:5001
  BalancerMember http://192.168.0.1:5000
  BalancerMember http://192.168.0.1:5001
  BalancerMember http://192.168.0.5:5000
</Proxy>

ProxyPass / balancer://rails_cluster/
ProxyPassReverse / balancer://rails_cluster/
```

Problem

Denial of Service Attacks

A denial of service attack is very easy if Rails is handling down/uploads. Just start X (= Rails instances count) simultaneous down/uploads over a throttled line.

This is valid for all slow requests, e.g.

- Image processing
- Report generation
- Mass mailing

Solution

Contaminate slow requests

Serve static files through the web server

- Apache, Lighttpd, nginx
(use x-sendfile for private files)
- Amazon S3

Contaminate slow requests

- Define several clusters for different tasks
- Redirect depending on URL

```
<Proxy balancer://main_cluster>
  BalancerMember http://127.0.0.1:5000
  BalancerMember http://127.0.0.1:5001
  BalancerMember http://192.168.0.1:5000
  BalancerMember http://192.168.0.1:5001
</Proxy>

<Proxy balancer://image_cluster>
  BalancerMember http://192.168.0.5:5000
</Proxy>

<Proxy balancer://upload_cluster>
  BalancerMember http://127.0.0.1:5000
  BalancerMember http://127.0.0.1:5001
</Proxy>
```

Pattern

Use the right tool for the job

Consider using other systems to handle uploads/downloads

- Merb
- Pure CGI / C / Perl

Separate resources in order to enforce quotas/limits

Conclusion

Conclusion

Real life solutions are more complex than they seem

Don't be too fast with throwing foreign code at a problem

Look for patterns in other peoples work

Questions?



Peritor Wissensmanagement GmbH

Lenbachstraße 2
12157 Berlin

Telefon: +49 (0)30 69 40 11 94
Telefax: +49 (0)30 69 40 11 95

Internet: www.peritor.com
E-Mail: kontakt@peritor.com