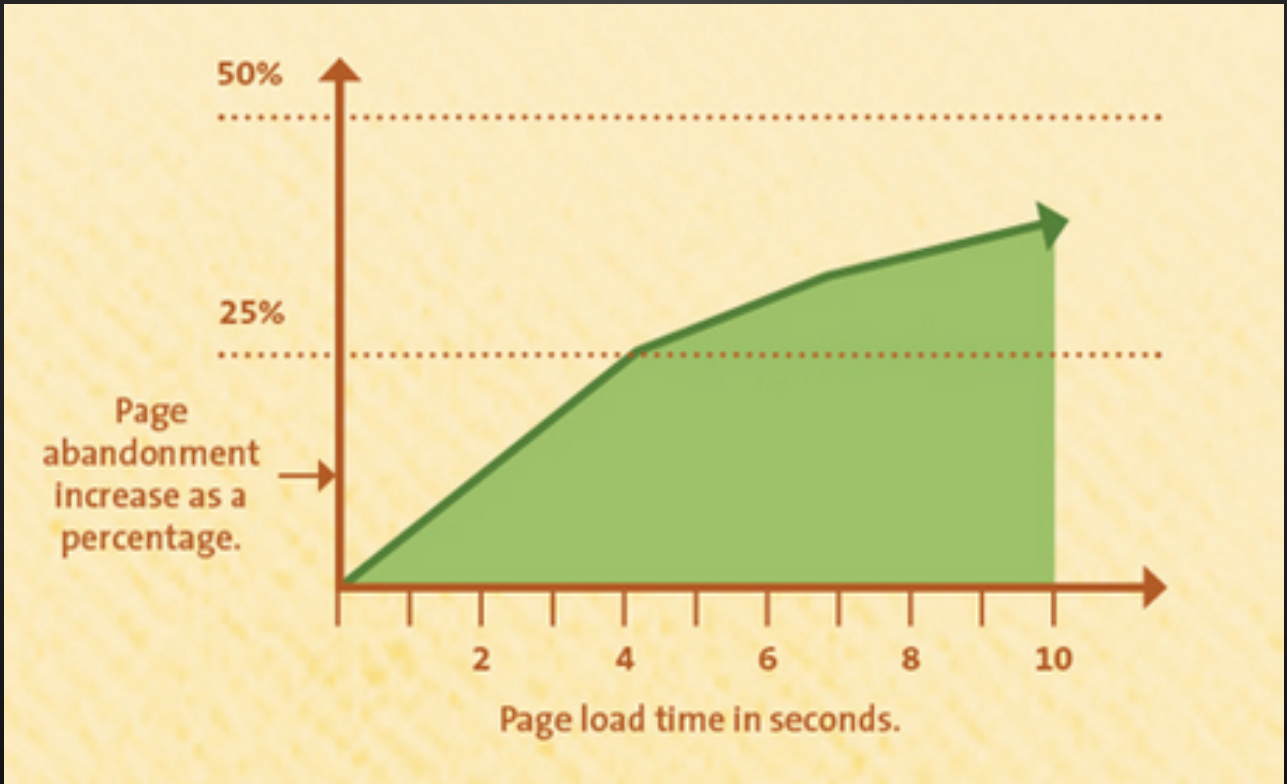
Caching in Rails

cause cache is cash

@all4miller http://www.shuntyard.co.za

A 1 SECOND DELAY IN PAGE RESPONSE CAN RESULT IN A 7% REDUCTION IN CONVERSIONS.

If an e-commerce site is making $100,000 per day, a **1 second page delay could potentially cost you $2.5 million in lost sales every year**.



http://blog.kissmetrics.com/loading-time/?wide=1

So what is this caching business...?

- Essentially a way skip expensive processing
- Expensive code runs once, result cached, next request uses cached result
- Low hanging fruit in terms of performance gains
- Site speed affects your bottom line
- No reason not to "cache in"... it's so easy!

Basic example

```
>> Rails.cache.fetch('answer')
==> nil
>> Rails.cache.fetch('answer') {1 + 1}
==> 2
Rails.cache.fetch('answer')
==> 2
```

What does Rails offer?

Page caching (removed in Rails 4)
Action caching (removed in Rails 4)
Fragment caching
HTTP caching
Low level caching
Automatic cache digest including dependencies
Abstracted cache store

Page Caching

- Page caching bypasses the entire application by serving up a file in /public from disk
- Have to expire cache manually using `expire_page`
- Can expire cache using observers
- Not really that useful, rather use HTTP caching
- Removed in Rails 4 (available as separate gem)

Page Caching example

```
class ProjectsController < ApplicationController
  caches_page :index

  def index
    @projects = Project.all
  end
  ...
end
```

Action Caching

- Action caching is an around filter for specific controller actions
- Requests hit the Rails stack
- This allows authentication and other restrictions to be run while still serving the result of the output from a cached copy
- Cache key is generated from the url
- Can use `:cache_path` to create custom key
- Removed in Rails 4 (available as separate gem)

Action Caching example (basic)

```
class ProjectsController < ApplicationController
  caches_action :show

  def index
    @projects = Project.all
  end

  def show
    @project = Project.find(params[:id])
  end
  ...
end
```

Action Caching issues... updates

- Now our project model gets updated, how do we deal with forcing the update into cache?
- We can use `:cache_path` to add a project attribute that will auto expire cache for us

Using cache_path effectively

```
class ProjectsController < ApplicationController
  caches_action :show, :cache_path => proc { |c|
    project = Project.find c.params[:id]
    { :tag => project.updated_at.to_i }
  }

  def show
    @project = Project.find(params[:id])
  end
end
```

Fragment Caching

- Fragment caching is taking rendered HTML fragments and storing them in the cache
- Basic form takes no arguments besides cache block
- Content rendered from block written to cache
- Can pass cache key to enable auto expiry
- Rails can auto generate key from ActiveRecord object
- Pain is dealing with code updates, usually add version to cache key. Rails 4 maintains nested dependency manifests for us. Can use cache_digets gem in Rails 3
- Must add `touch` to `belongs_to` association so updates on child update parent models `updated_at` attribute

Fragment Caching example

```
<% cache task do %>
  <li>
    <%= task.name %>
    <%= link_to "rename", edit_task_path(task) %>
  </li>
<% end %>
```

Russian Doll caching

- The technique of nesting fragment caches to maximize cache hits is known as russian doll caching
- Major benefit is sibling caches can be reused
- E.g. in our app we have many tasks per project. When a single task gets updated the project cache is reworked but we can read sibling tasks straight from cache
- NB# don't forget to add `touch` to `belongs_to` associations

Russian Doll example

/views/projects/show.html.erb

```
<% cache @project do %>
  <h2><%= @project.name %></h2>
  <%= render @project.tasks %>
<% end %>
```

/views/tasks/_task.html.erb

```
<% cache task do %>
  <p><%= task.name %></p>
<% end %>
```

HTTP Caching

- HTTP caching is the most complex and powerful caching strategy you can use
- HTTP caching works at the protocol level
- You can configure HTTP caching so the browser doesn't even need to contact your server at all (e.g. js/css assets)
- It uses a combination of headers and response codes to indicate whether the user agent should make a request or use a locally stored copy instead
- The invalidation or expiring is based on ETags and Last-Modified timestamps

HTTP Caching in Rails

- Rails gives us `stale?` and `fresh_when` methods
- Use `stale?` when you don't use default rails idioms
- Use models `updated_at` attribute to determine freshness
- Defer sql execution using `scoped`
- Set cache to public for public content
- There is also `expires_in` time base helper

HTTP Caching example

```
 def index
    @projects = Project.scoped
    fresh_when last_modified: @projects.maximum(:
updated_at)
 end
```

Low level caching

- Rails gives us Rails.cache
- Rails.cache.fetch will read and write to cache if not present
- Takes 3 arguments; the cache key, options hash and a block
- Can store anything in cache
- Useful for caching expensive methods such as external API calls or shared global data

Low level caching example

```ruby
def some_expensive_method
   Rails.cache.fetch("some_expensive_method", :expires_in => 5.minutes) do
     sleep 4
     "some_expensive_method"
   end
end
```

Rails Cache Stores

- Used to store cached content in something useful like Memcached
- Configure default cache store using `config.cache_store`
- Rails supports following cache stores:
  - ActiveSupport::Cache::MemoryStore
  - ActiveSupport::Cache::FileStore
  - ActiveSupport::Cache::MemCacheStore
  - ActiveSupport::Cache::EhcacheStore
  - ActiveSupport::Cache::NullStore (test/development)

Useful Resources

http://guides.rubyonrails.org/caching_with_rails.html
http://edgeguides.rubyonrails.org/caching_with_rails.html
http://www.broadcastingadam.com/2012/07/advanced_caching_revised/
https://devcenter.heroku.com/articles/caching-strategies