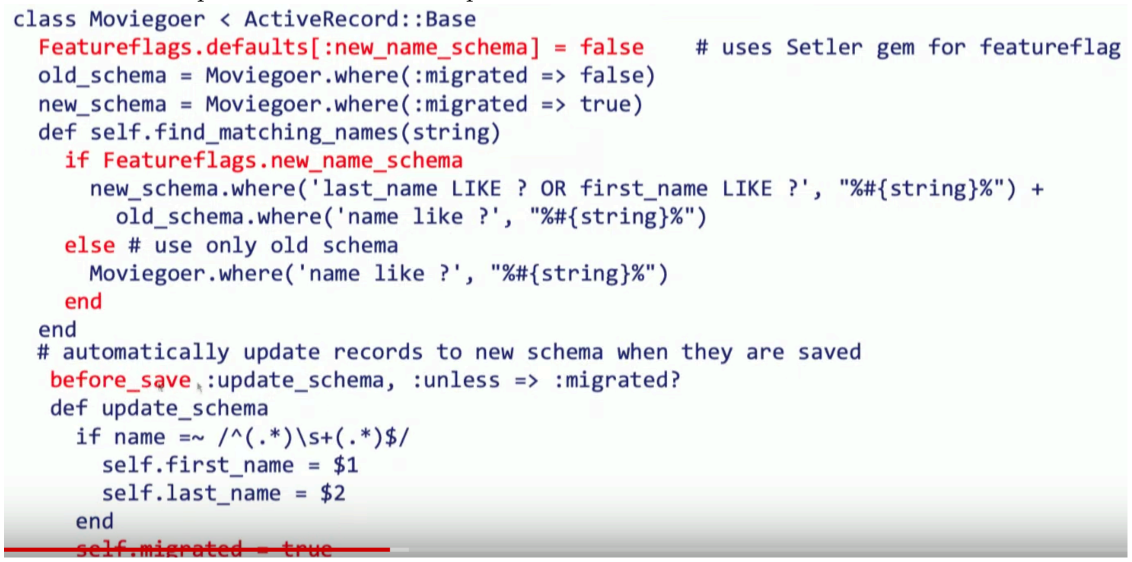
Upgrades and Feature Flags

FFs

- Multiple code paths which you can select at runtime which code path is used
- Can use ffs to update code without rolling out a feature
- Makes sure change affects all customers at once
- Store in DB not in config file because config files not easy to check on the fly
- Can use `rollout` gem to cover main cases

Upgrading With Inconsistent DB Schemas

- Without FFs: Can bring service down and upgrade it
- Procedure to use FFs:
  1. Apply nondestructive migration
  2. Deploy code protected by ff
  3. Flip feature flag on; if disasters flip it back
  4. Once all records moved, deploy new code without ff
  5. Apply destructive migration (remove old columns)
     - Have an 'undo' plan for destructive steps
  - eg:



```
class Moviegoer < ActiveRecord::Base
  Featureflags.defaults[:new_name_schema] = false    # uses Setler gem for featureflag
  old_schema = Moviegoer.where(:migrated => false)
  new_schema = Moviegoer.where(:migrated => true)
  def self.find_matching_names(string)
    if Featureflags.new_name_schema
      new_schema.where('last_name LIKE ? OR first_name LIKE ?', "%#{string}%") +
        old_schema.where('name like ?', "%#{string}%")
    else # use only old schema
      Moviegoer.where('name like ?', "%#{string}%")
    end
  end
  # automatically update records to new schema when they are saved
   before_save :update_schema, :unless => :migrated?
   def update_schema
     if name =~ /^(.*)\s+(.*)$/
       self.first_name = $1
       self.last_name = $2
     end
     self.migrated = true
```

    - Lazy migration strategy
    - Can also have another OLP to automatically passively migrate records
  - Good because can update with the app running
  - Can always turn off the ff if things go wrong
  - Once completely migrated, can remove conditional checks on ff and ff itself as well as db columns migrated and no longer used or added for migration

Other Uses of FFs

- Preflight checking ≙ gradual rollout of feature to increasing numbers of users
- A/B testing
- Complex feature whose code spans multiple deploys
- Eg use case: password algorithm update

Monitoring

- In development (profiling) ≙ identify possible performance/stability problems before they get to prod

- In production
  - Internal ≙ instrumentation embedded in app and/or framework
  - External ≙ active probing by other site(s)
    - Detects if site is down
    - Detects if site is slow for reasons outside measurement boundary of internal monitoring
- Can make cucumber tests for monitoring

Stress/Load Testing

- Assess how far you can push a system
  - Before performance (apdex 95th percentile response) becomes unacceptable
  - Before it gasps and dies
  - To expose longevity bugs like memory links
- Can help assess a bottleneck
- Can be simple or sophisticated

Resource Leaks

- Leaked memory can be fixed temporarily by rebooting
  - Can do automatically with 'rejuvenation' / 'rolling reboot'
- Can run out of db rows
  - PaaS often allows minimally disruptive upgrade-in-place
- Database pollution from unclaimed objects (eg: shopping carts)
  - Can run periodic 'stale object' sweepers that look at `updated_at` to purge

Caching

- Avoid touching db if answer to query hasn't changed
- Avoid re-rendering a page or partial if the underlying objects on which they depend haven't changed
- Expire stale cached versions when they become invalid
- Goal ≙ understand what the 'unit of caching' is and how to expire things

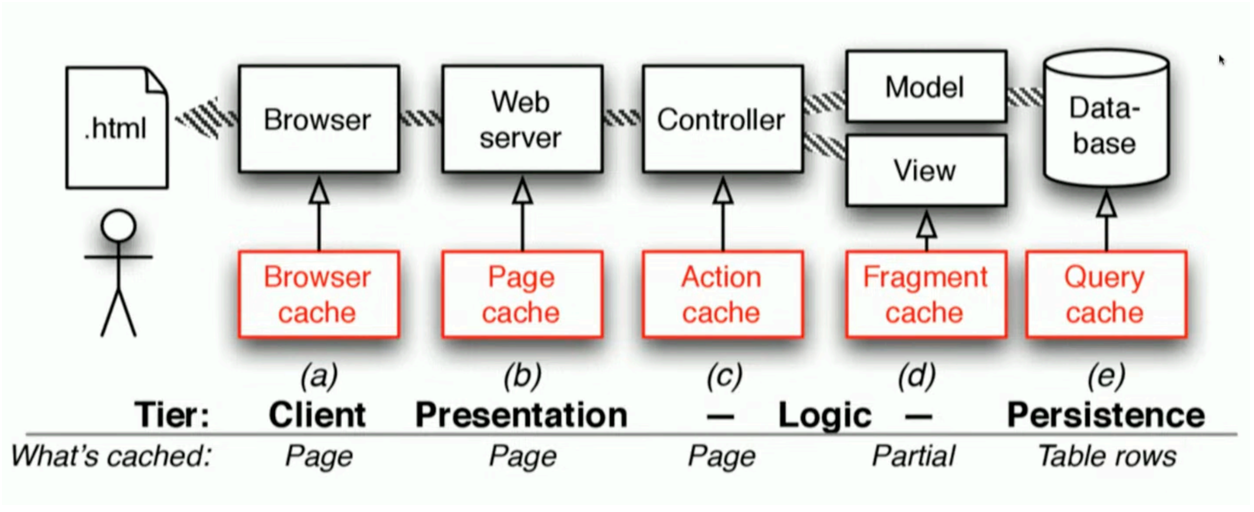
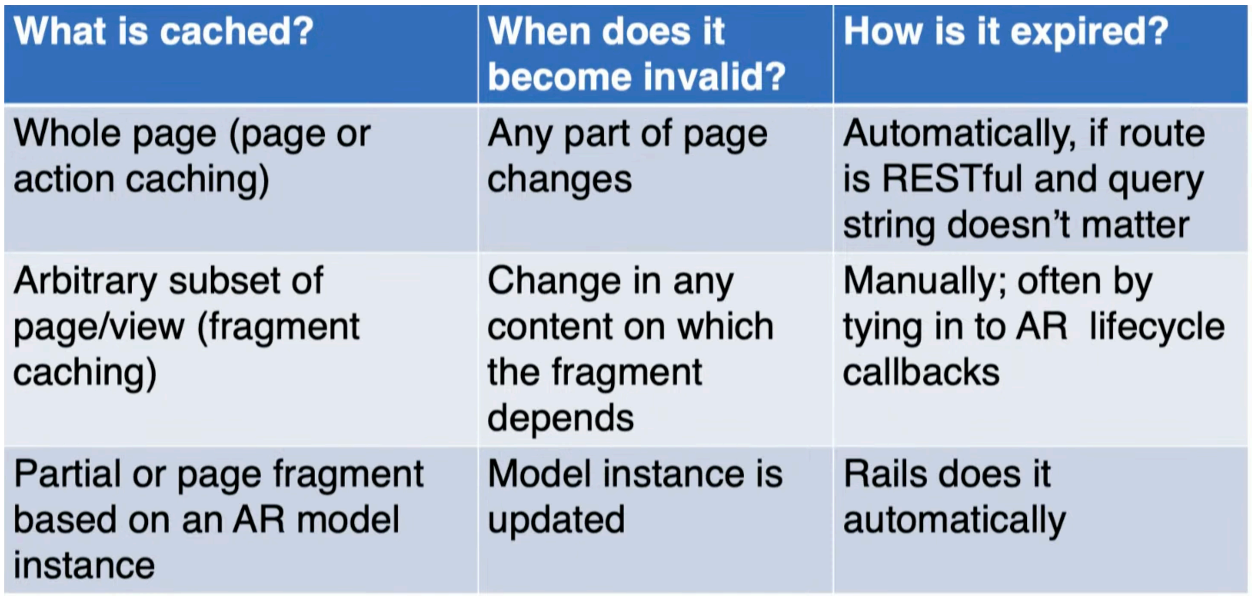


Figure 1: Screenshot_2023-11-14_at_3.36.02_PM.png

  –

- Stored in key-value store

  - `ActiveSupport::Cache::Store` facade: `read`, `write`, `delete`, `exist?`, `fetch`
  - Out-of-the-box support: memcached, Redis, file system
  - `ActiveSupport::Cache::NullStore` uses null-object pattern to 'disable' caching in development and test envs



What is cached?	When does it become invalid?	How is it expired?
Whole page (page or action caching)	Any part of page changes	Automatically, if route is RESTful and query string doesn't matter
Arbitrary subset of page/view (fragment caching)	Change in any content on which the fragment depends	Manually; often by tying in to AR lifecycle callbacks
Partial or page fragment based on an AR model instance	Model instance is updated	Rails does it automatically

Figure 2: 525

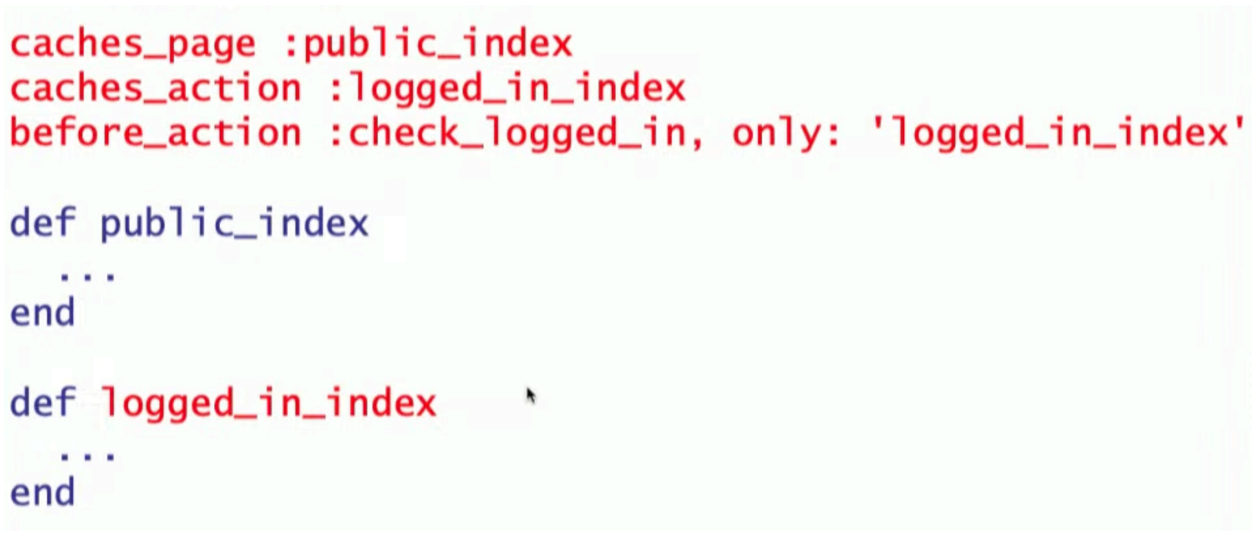
-

Conditions

- Page caching bypasses controller action with `caches_page :index`
  - Only should be done on static pages since it doesn't touch the controller at all
- Action caching runs controller filters first
- Note: caching is based on page URL without query params so don't mix filter and non-filter code paths in same action
-
- Fragment caching is almost always helpful

Fragment Caching for Views

- Name the part of the view you want to cache (partial, collection, etc.)
- Wrap with `< % cache 'name' do %> ... < % end %>` to 'memoize'
- Makes Rails look in cache store to cache that specific cache block
  - Could also cache on an active record object to update dynamically
- Can use a cache sweeper to observe a model and look for anything that will invalidate the cache

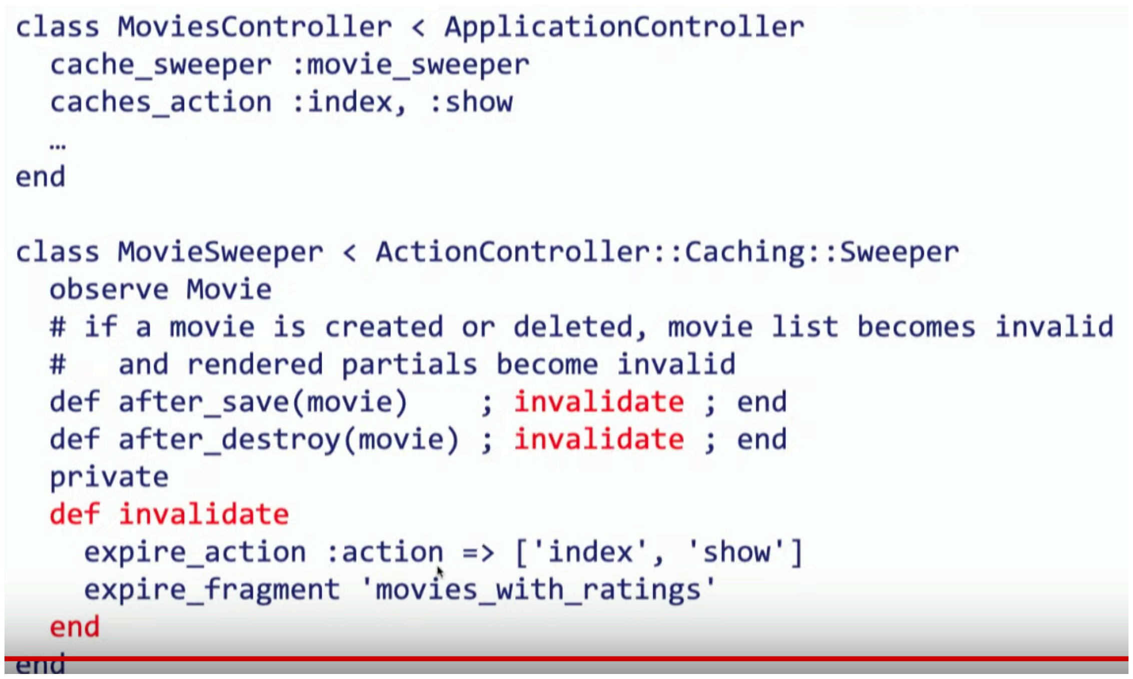


```
caches_page :public_index
caches_action :logged_in_index
before_action :check_logged_in, only: 'logged_in_index'

def public_index
  ...
end

def logged_in_index
  ...
end
```

Figure 3: 475

  - eg: 

```
class MoviesController < ApplicationController
  cache_sweeper :movie_sweeper
  caches_action :index, :show
  …
end

class MovieSweeper < ActionController::Caching::Sweeper
  observe Movie
  # if a movie is created or deleted, movie list becomes invalid
  #   and rendered partials become invalid
  def after_save(movie)    ; invalidate ; end
  def after_destroy(movie) ; invalidate ; end
  private
  def invalidate
    expire_action :action => ['index', 'show']
    expire_fragment 'movies_with_ratings'
  end
end
```

- Always better to expire something valid than not expire something that is invalid

Database Indices

- Index ≙ hash-like data structure that speeds up access when searching DB by column other than the PK
- Don't have an index on every column b/c all indices must be updated on each table when modifying
- To index:
  - Foreign key columns
  - Columns that appear in `where()` clauses of `ActiveRecord` queries
  - Columns on which you sort
- `rails_index` is a gem that helps identify missing indices (and unnecessary ones)
- Probably a better idea to over-index than under-index