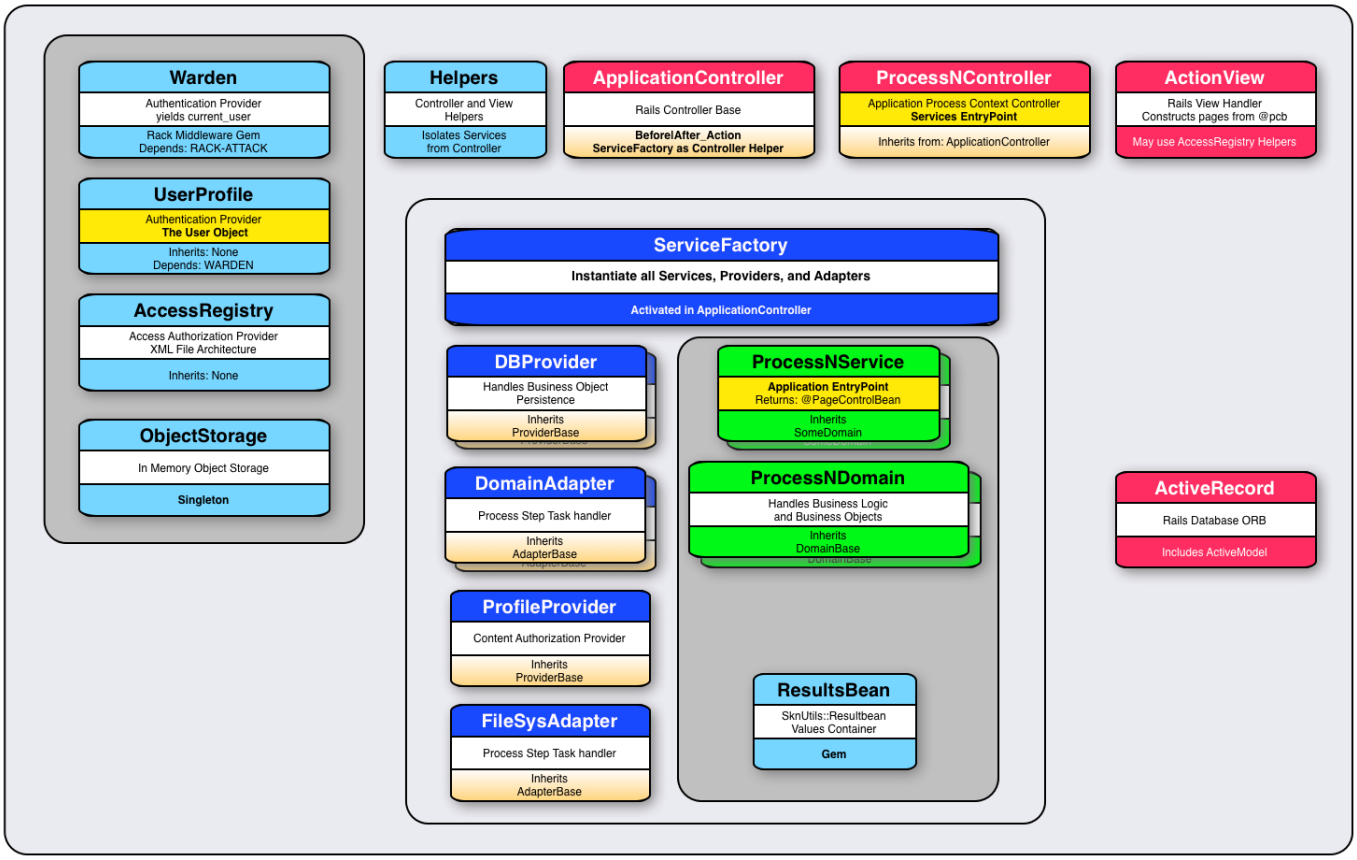
# Architectural Details




**Design Guidance: Commentary**

## This application is a Rails 4.2.5+ & Ruby-2.2.3 app. However, WE DO NOT FOLLOW THE CLASSIC RAILS WAY.

Rails is an MVC Web Framework. The MVC model is a classic and used with success, as is, for many applications. However, there are many maintenance, scaling, and growth issues when the codeset exceeds a certain size or features set mix. We understand these growth issues, and have chosen to apply the more traditional **Domain Driven Design(DDD)** approach, with Rails isolated as a Web Interface; as much as possible.

Our way involves treating all of Rails as a Web Interface or adapter, and isolating our application business logic from it. Calling on Rails only when absolutely needed to exchange resul with the user. One outcome is, we don't waste time testing Rails and can focus on our delivery with significant code re-use opportunities.

The persistent assumptions, by Rails's Developers, that MVC and convention over configuration is all that's needed for a web app is disturbing. There is sooo much talent in the Rails community, it is possible to build apps better.

## This application source, and implementation, serves as a model for how web applications can be engineered without sacrificing longevity or major reworks.

Rails is a Great Web Framework. Your application can derive great benefit from Rails when engineered (Designed) to co-exist with Rails.

**Design Guidance: Our (Services) Way**

## We do not add business process logic to Models, controllers, or views! (the classic Rails Way)

We choose to take one step back from Rails code containers, models - views - controllers, and place our code in different containers more appropriate to our application purpose. This need not be considered a huge departure from the Rails Way, just an alternate approach to building application with Rails.

We consider controller methods as entry points into our business logic and those entry points are coded with one-line invocations of our specialty service methods `:service_name.entry_method_name(params)`. Where `:service_name` is the name of a business process service class associated with that page. This service's method will be/is specially designed to accept input via the requests params and return results using a consistent package of values, which can easily be converted into JSON, or consumed by the view that has been prepared to accept ONE instance variable `@page_controls` values bean.

For apps that have some sort of menu-bar or top of page information section. We up the instance variable count by one, by allowing a `@page_info` values bean, which would contain all the values needed to populate the top of page information section of that page thru the page partials prepared for it. In cases where the return code to the controller method is negative, we maintain the `:success, :message` keys in the values bean (i.e. ResultsBean) to carry this failure state. This allows the controller to redirect to the logical next page based on error, or continue to regular destination on success; message is used for the Flash message when needed.

```
class ProfilesController < ApplicationController

  # HTML only response
  def content_profile_demo
    @page_controls = content_service.handle_demo_page(params)
    flash.notice.now = @page_controls.message if @page_controls.message.present?
  end

  # JSON only response
  def api_content_profile
    @page_controls = content_service.handle_content_api(params)
    render json: @page_controls.to_hash, status: @page_controls.success ? :accepted : :not_found, layout: false, content_type: :json and return
  end

  ...
end
```

For models we have a similar approach. Only add what is relevant and required to effectively use or protect the data record.

```
ProfileType < ActiveRecord::Base
  has_one :content_profile
  validates_presence_of :name, :description

  def self.option_selects
    self.all.map do |pts|
      [pts.name, pts.id, {data: {description: pts.description}}]
    end
  end
end
```

In the case of models this mean only short methods to help with form option lists, and the model's own validation methods. Nothing more!

If we don't add business logic to the controllers or the models, where does it go? Answer, in the Services objects which inherit directly from Domain objects.

A side benefit of not adding business logic in controllers and models is: ***If there is no code there, then you don't have to test them!***. Additionally, if you must test a controller; mocking a single method call, and mocking the @page_controls makes controller testing simpler.

---

**Design Guidance: Domain and Services Model**

## Using object-oriented design methodologies, we develop a domain model for our application, and encapsulate those objectives in a Domain class.

To delivery the value contained in a **Domain class** we wrap them in a **Services class**. Which acts as an adapter to bring that value to the user. This typically means developing an html page with views and controllers to present the informational results to the user. However, since we only delivery a package of data values to the controller interface method, it is possible to serve those data values as JSON to a browser based application, like BackBone, or to a wide range of endpoints; even a PDF generator.

The core of our Application is defined in an object-oriented fashion and protected or isolated from the delivery frameworks(s) used to delivery its value to the user. When changes and enhancements to the business logic is needed, we can make those changes without great concern or contraints being imposed by the delivery mechanism. This also faciliates making changes or upgrades to the delivery frameworks.

Services inherit directly from Domains to further shield our application from the external frameworks; this is easy to see when considering the controller. Models on the other hand need to be wrapped by Domain methods to prevent their inline introduction or leakage into the Domains. I've chosen to go one step further and dedicate a `<context>Provider` class that wraps all the ActiveRecord IO calls, expecting it to be called by either Domain methods or Services methods. I have considered using a Repository strategy to add more isolation and to facilitate testing. For now the specialty domain level IO class provides enough isolation. Ideally we should be able to write RSpec tests for the service/domain classes without invoking the Rails environment, or at least with fewer dependencies.

If service objects are focused on transforming domain results to external interfaces, and domains themselves are focused on the core business logic of a single process. Then yes, there will be many or several domains classes, one to support each business process. However, a domain class need not do all the heavy lifting, its single-responsibility is producing a contextually valid result from the triggering event or input.

Domains produce those results best by invoking secondary task oriented objects to complete elements of the overall process, positioning domains to be the orchestrator or main-line for the business process they are assigned. Task oriented objects must adhere to the single-responsibility guideline to maintain their value to the overall system's design. Again enhancing our ability to test off-line from Rails, and making the code more portable with greater reuse opportunities. A task object would completely handle a task step or elements of the logical process flow domain objects are responsible for. *use the force!*

```
class ServiceFactory < ::Factory::FactoriesBase


  ##
  # Application Services used in Controller methods
  ##

  def access_service
    @sf_access_service ||= ::AccessService.new({factory: self})
    yield @sf_access_service if block_given?
    @sf_access_service
  end

  ...
end
```

```
class ContentService < ::ContentProfileDomain


  # Controller Entry Point
  def handle_demo_page(params={})
    SknUtils::PageControls.new({
                                  success: true,
                                  message: "",
                                  page_users: get_page_users(PROFILE_CONTEXT)
                               })
  rescue Exception => e
    Rails.logger.error "#{self.class.name}.#{__method__}() Klass: #{e.class.name}, Cause: #{e.message} #{e.backtrace[0..4]}"
    SknUtils::PageControls.new({
                                  success: false,
                                  message: e.message,
                                  page_users: []
                               })
  end
```

```
 class ContentProfileDomain < ::Factory::DomainsBase


  PROFILE_CONTEXT=""  # override in service

  ##
  # Returns a bundle for each available user
  # - includes access/content profile
  def get_page_users(context=PROFILE_CONTEXT)
    usrs = []
    Secure::UserProfile.page_users(context).each do |u|
      usrs << {username: u.username,
               display_name: u.display_name,
               user_options: u.user_options || [],
               get_content_object_url: factory.page_action_paths([:api_get_content_object_profiles_path]),
               package: [ access_profile_package(u), content_profile_package(u) ]
      }
    end
    usrs
```

Admittedly, this demonstration has not reached that level of *Nirvana* yet, but it is the design objective! Refactoring now has a clea target (i.e plan).

**Design Guidance: This Application's Value Domain and Intent**

Intended to be a feasible example demonstrating Authentication, Authorization, and Content Access Control.

| | |
|---|---|
| **Primary Use Case** | Allow users to download a collection of file based content. The content is protected by access and content controls, ensuring only users authorized for th type of content have access. |
| **Administer Profiles** | Represent what content access we desire for each user in a standardized way, via the ContentProfile. Provide ancillary and administrative processes to register and maintain users for the application. |
| **Execute AccessProfile** | Apply the pre-defined AccessProfile control to all navigation and clickable actions of the application. |
| **Execute ContentProfile** | Apply the pre-defined ContentProfile controls against down-loadable assets requests from a user. |

Now for the architectural component details. i.e how these design principals were applied to meet the business model objective?

## [Services Strategy] UserProfile Service

### UserProfile Service

**Instantiate** By Warden at user login, via Class methods.

Requires a User object to instantiate itself, and that object must have a unique Id.

During initialization the user's roles are rendered from those assigned via group roles.

**Inheritance** Secure::UserProfile is a base class with direct access to the memory-based Object Storage Interface Module `Factory::ObjectStorageService`, capable of keeping references to memory objects across the users request/response cycle; much like the session does and for the same reasons. i.e som objects are expensive to create and may benefit from session-like retention until there usage is complete.

**Provides** Secure::UserProfile provides the User Context for all application processes.

Isolates external authentication sources and methods from our internal details. External systems need only authenticate a user and pass along a persisten and unique id. User permission or roles can be included in this external bundle, or we can do an internal mapping of that external user. This system has its own internal User table which contains the persistent Id, individual and group roles.

Static class methods are provided to allow Warden to:

| | |
|---|---|
| **`:find_and_authenti...`** | Locate a user with these credentials in whatever user store is available |
| **`:fetch_remembered_...`** | Locate a user using their remember token |
| **`:fetch_cached_user...`** | Locate an existing user exclusively from the inMemory storage facility. |
| **`:logout(token)`** | Log out the user and remove them from the inMemory storage, clear their remember_token and session. |
| **`:last_login_time_e...`** | Update the last time user access the system and if expired, revalidate their credentials. |
| **`:authenticate?(une...`** | Bcryts Authenticate returns self, we need to override that return value to return self instead |
| **`:enable_authentica...`** | After successful login, this method saves the user object into our inMemory storage for later session level retrieval. Also upda :last_access. |
| **`:disable_authentic...`** | Remove the user object reference from inMemory storage, updates :last_access. |

Static class methods are provided to facilitate content profile, and testing service to:

| | |
|---|---|
| **`:page_users(contex...`** | Retrieve the list of all users without logging them in. For use with offline operations. |
| **`:page_user(uname, ...`** | Retrieves a single user object without logging it in, for offline use. |

Instance methods are provided to allow AccessRegistry access control application wide:

- `:has_access?(resource_uri,options)`
- `:has_create?(resource_uri,options)`
- `:has_read?(resource_uri,options)`
- `:has_update?(resource_uri,options)`
- `:has_delete?(resource_uri,options)`

Instance methods support ApplicationHelper methods for use in view and controller actions:

- `:current_user_has_access?(uri, options)`
- `:current_user_has_read?(uri, options)`
- `:current_user_has_create?(uri, options)`
- `:current_user_has_update?(uri, options)`
- `:current_user_has_delete?(uri, options)`

Designed to be the User's proxy to the whole system. Working in concert with Authentication and Authorization components it enables delivery of secure content with the availability of different authentication schemes.

The User's Context

## [Services Strategy] ServiceFactory

### ServiceFactory

**Instantiate** ServiceFactory is instantiated by the top level Application Controller using a `:before_action` callback to a private method called `:establish_domain_services`. There is also a twin to this callback referred to as `:manage_domain_services`, called by the `:after_action` callback.

`:establish_domain_services` and `:manage_domain_services` work to recover assets from the request session, and to save keys to those assets ir the session just before the controller exits back to the browser. The ServiceFactory is lightwieght and would derive no real benefit from being serialized int a session. So it is simply created anew with each request/response cycle.

The ServiceFactory itself is instantiated as `@service_factory` in the top level ApplicationController, behind a memotizing method `service_factory`. instantiate a ContentService use: `service_factory.content_service.some_method_name(params)`.

The intended use of the `service_factory` is to provide lazy instantiation and memotization of all service/domain objects in the application. And to additionally provide them critical facilites, like `:current_user`, and act as an isolation gateway to the application controller.

Inside of services, like ContentService or ProfileProviders, reference the service_factory as `:factory` or `:service`, as in: `factory.content_service.some_method`.

**Inheritance** ServiceFactory, inherits from `::Factory::ServicesBase` which yields a memory-based Object Storage Interface Module `Factory::ObjectStorageService`, capable of keeping references to memory object across the users request/response cycle; much like the session does and for the same reasons. i.e some objects are expensive to create and may benefit from session-like retention until there usage is complete.

**Provides** ServiceFactory provides Services with the `:current_user`, `:controller` reference, and a `:method_missing` feature capable of directing :not_found method calls to the controller to be serviced if possible.

Services should truly limit their demands of the controller object to zero if possible, and have the service_factory devise a solution to their requirements. Th most often experienced demand for the controller is when a service needs to resolve a named route or render a template partial to embed in a json response.

Let's not recreate the application controller, for those types of requirements use the controller, just wrap those patterns in a method defined in the service_factory or a controller helper/presenter. This way it can be easily mocked out later for testing. In fact, I do use a `:page_actions` view helper located in the ApplicationHelper module, to handle rendering and route resolutions from a hashed set of control data from a service. Basically, outsourcing to the ActionView those things it does best; and totally removing the need for a controller object in the majority of services and domains.

Engineering objectives shall include, avoidance of the controller in all services, domains, and specialty business services classes. It is possible and practical, your RSpec tests will demonstrate it later.

**ObjectStorageServic...**

```
# Saves object to inMemory ObjectStore
# Returns storage key, needed to retrieve
:save_new_object(obj)

# Updates existing container with new object reference
# returns object
:set_existing_object(key, obj)

# Retrieves object from InMemory Storage
# returns object
:get_existing_object(key)

# Releases object from InMemory Storage
# returns object, if present
:remove_existing_object(key)
```

**Session API**

```
:get_session_params(key)
:set_session_params(key, value)
```

**Named Route Helpers**

Converts named routes to string
Basic: `[:named_route_path, {options}, '?query_string']`
Advanced: `{engine: :demo, path: :demo_profiles_path, options: {id:111304},query:'?query_string'}`

Example: `factory.page_action_paths(paths)`

All Services, like ContentProfileBuilder, ProfileDataServicess or ContentService, should be instantiated via the service_factory facilities. And primary utility objects can also be supporte via the service factory.

Container, and Provider of Business Object Services!

---

## [Services Strategy] Services

**ContentService**

**Instantiate** ServiceFactory provides instantiation services to any requester in the request/response cycle.

The intended use of services object methods is to handle all operations for a controller entry point(url), and return a single information bundle `@page_controls`, based on the `SknUtils::PageControls` group of result bean containers.

**Inheritance** ContentService, inherits io routines and common business logic from the `ContentProfileDomain` class. The ContentProfileDomain class inherits from th `::Factory::DomainsBase`, which yields common initialization services of like the current_user() and access back to the service_factory for interaction with the controller if needed. In particular, the `service_factory` includes object storage services and session storage services through its inheritance chain.

**Provides** ContentService provides customized business logic and standard information packaging for the application entry points assigned to it. Where ever used y can be assured of a valid response encoded in a `@page_controls` bean container, and all exceptions are trapped by its top level method. PageControl bean containers are required to include `success: true|false, and message: error_message|success_message|empty` values at a minimum.

JSON Api methods are hosted by this service.

Instantiated by: `service_factory()`

---

## [Services Strategy] Domains

### Domains

**Instantiate** Domains are not instantiated directly, they are inherited by higher level and more specialized service classes.

The intended use of domains class methods is to offer a business logic control of a single business domain. It is the starting point for all business process services.

**Inheritance** Domains inherit directly from the `::Factory::DomainsBase` class, and yields business logic methods to higher level service objects. Services take the results of one or more of these *business logic complete* methods and package them for external exchange in the controller response.

**Provides** Domains have logical access to the service factory and all other peer services the factory manages. Typically you would find two to three levels of method in a Domain class.

- **Level One** Top level business interface. Methods at this level take responsibility for doing the whole task; process a order or payment, would be an example of this.
- **Level Two** The notion of the whole task, is likely to have component steps involved. At this level methods are expected to perform one elements of the whole task. For this single-responsibility reason, we recommend creating dedicated objects to handle these component parts.
- **level Three** Is rarely needed, but if so would handle very narrow objectives; like I/O or RESTFul routines.

This is the best place to exploit object-oriented design principles.

The Business Process Interface is here.

Inherited by: `ContentService, ...`

## [Services Strategy] domainProviders

### DBProfileProvider

**Instantiate** ServiceFactory provides instantiation services to any requester in the request/response cycle.

The intended use of this services object methods is to handle all IO related operations for domain context data. It should not interact directly with ApplicationController entry points.

**Inheritance** DBProfileProvider, inherits common instantiation service from the `Factory::ProvidersBase` class. The Factory::ProvidersBase yields common initialization services, like the current_user() and access back to the service_factory for interaction with the controller if needed, and object storage service

**Provides** DBProfileProvider provides contextual data access for content profile data objects with standard data packaging for internal processes.

Instantiated by: `service_factory()`

## [Services Strategy] taskAdapter

### FileSystemAdapter

**Instantiate** ServiceFactory provides instantiation services to any requester in the request/response cycle.

This service retrieves documents indicated by a DB-backed ContentProfile or a XML-backed AccessProfile, normaling the results.

The intended use of a task adapter is to handle external IO and retrieve documents from their storage location based on the contents of a content profile entry.

**Inheritance** Providers::FileSystemAdapter inherits directly from the `Factory::ContentAdaptersBase` class, and yields access services to domain classes objects.

**Provides** FileSystemAdapter provides logical access to content storage facilities.

Currently there are two implementation of this component: FileSystemAdapter and InlineValueAdapter classes.

Instantiated by: `service_factory()`

## [Services Strategy] ResultBean Value Containers

The **skn_utils.gem** contains dynamic base classes inherited to create local value containers or beans; plain old ruby objects (PORO).

Initialized with a Hash of key value pairs: `res = SknUtils::PageControls.new({one: 1, two: [{one: 'one', two: 'two'}]})` The keys become method names that return the associated values. This transformation of the hash continues, during its initialization, to follow nested hashes and arrays of hashes.

Nesting can be controller by initialization params, or choose from ready made base classes, preset to follow or not follow nesting. SknUtils::ResultBean class only follows directly value hashed, while the SknUtils::PageControls class follows directly valued hashes and arrays of hashes. SknUtils::Genericbean only follow the initial hash, leaving all nested values untouched.

Overall the result of using this gem is an easily used container for transporting or packaging values for use in views, json responses, etc. Accessing the values in a SknUtil bean can be done using `dot` notation, or `hash` notation: example: `res.one #=> 1`, `res.two.first.two #=> 'two'`

If you need to unwrap the bean to convert it to json or just to have the hash back;
`res.to_hash #=> {one: 1, two: [{one: 'one', two: 'two'}]}`, gets the job done.

There are many more logical features built into the package, read more at SknUtils (https://github.com/skoona/skn_utils). The is one feature that you should know and use: the present `<keyName>?` feature. Example: `res.two? #=> true, res.twenty? #=> false`

| | |
|---|---|
| **SknUtils:PageControls** | Used directly to transport values to controller method. |
| **Utility::ContentProfile...** | Locally defined container used to transport data-driven content profiles. |

skn_utils.gem

### [Services Strategy] Rails View Helpers

Rails Views: BootStrap, JQuery DataTables, and SimpleForms are our choice for classic erb page composition and rendering.

## View Helpers/Builders

| | |
|---|---|
| **PageActionsBuilder** | A Class designed to build page action dropdown menus from an array of hashes contain with keys describing each menu item.<br><br>It is initiated by an `ApplicationHelper#do_page_actions` and assumes it input array is in either `@page_controls.page_actions or .package.page_actions`.<br><br>In your services routine response you can add a `:page_actions` array of params, and this helper will create the classic pulldown or action button for you |
| **#page_action_paths** | Located in `ApplicationHelper#page_action_paths`, this methods resolves named routes into strings. You would use this method to resolve named paths for inclusion in json responses, or anywhere else you might need it. |

Describes helpers we created for views.

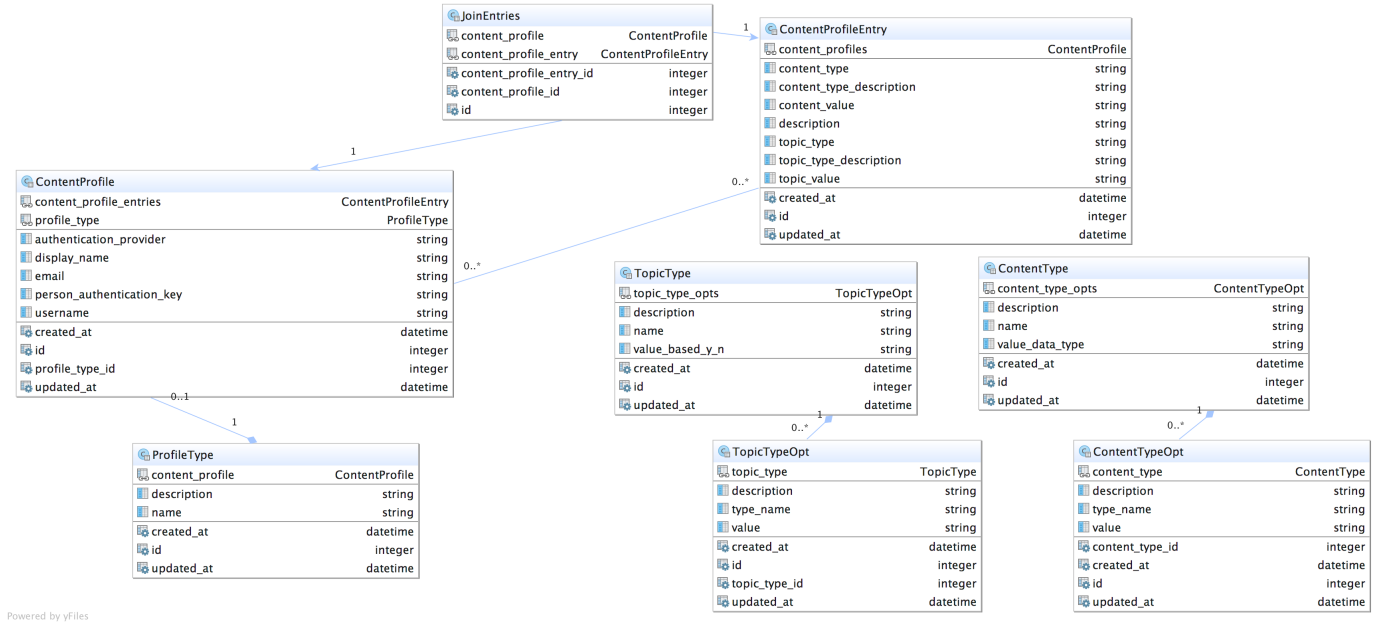
### RSpec Testing

RSpec Testing: This Rocks

More Stuff ...

### Authorization Data Models

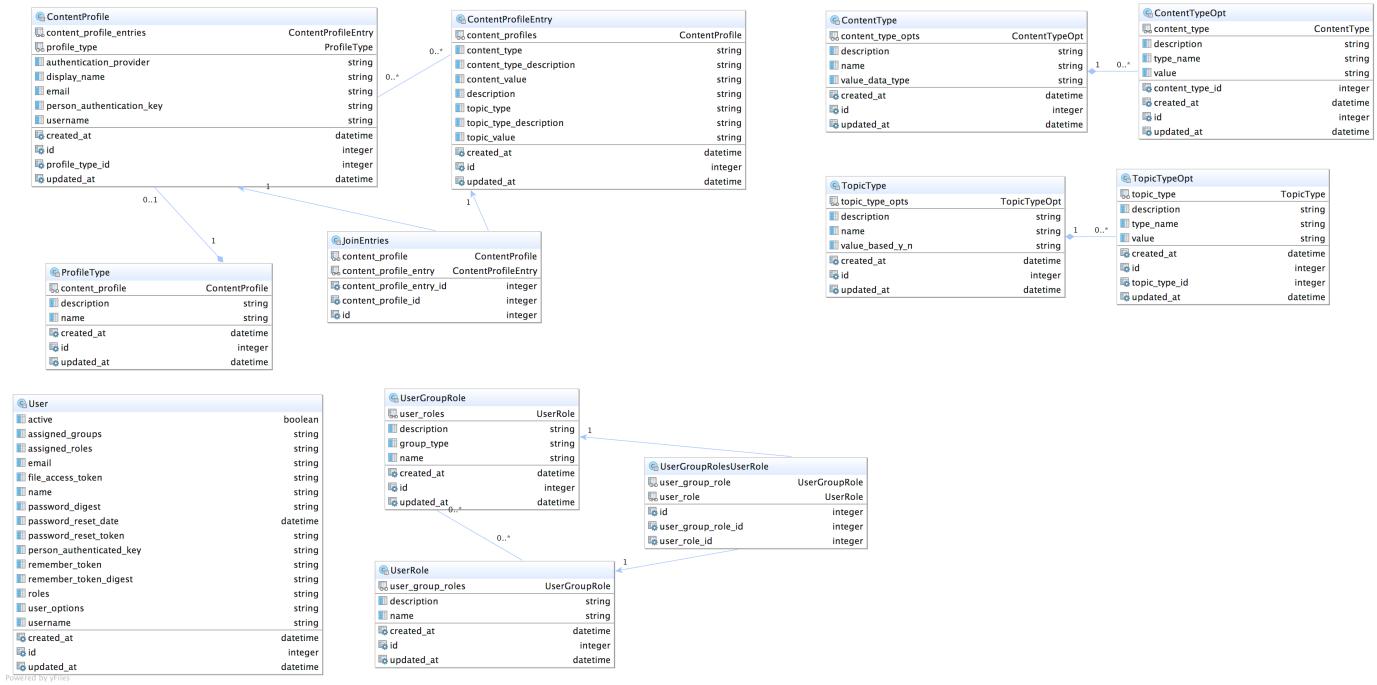
# Authorization Data Models




Authorization Data Models

### Application Data Model

# Application Data Model




Authentication and Authorization Data Models