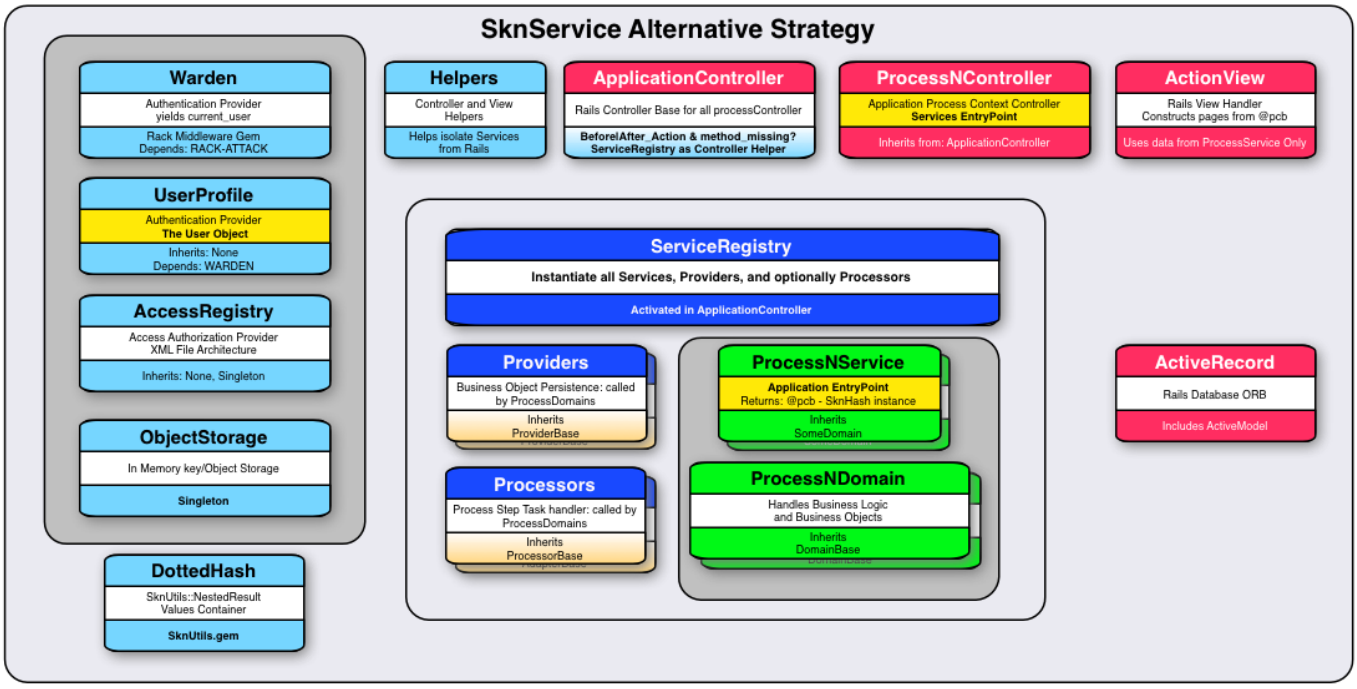
Architectural Details



Design Guidance: Commentary

DO NOT FOLLOW THE RAILS WAY.

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 certa size or features set mix. Understanding these growth issues, it is recommended that you apply a more traditional **Domain Driven Design(DDD)** or **Object Oriented(OO)** approach to building web applications, with Rails isolated as a Web Interface as much as possible.

This would involve 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 results with the user.

However, this is not another Web Framework! It is a thought provoking alternative to the Rails Way. An alternative that still builds applications using the full capabilities of Rails, while keeping your business logic components separate from Rails. This avoids the negative impact of Rails release changes, and enables your ability to easily add or change features.

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 the request. 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 container.

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 container, which would conta 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 container (i.e. NestedResult) 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.

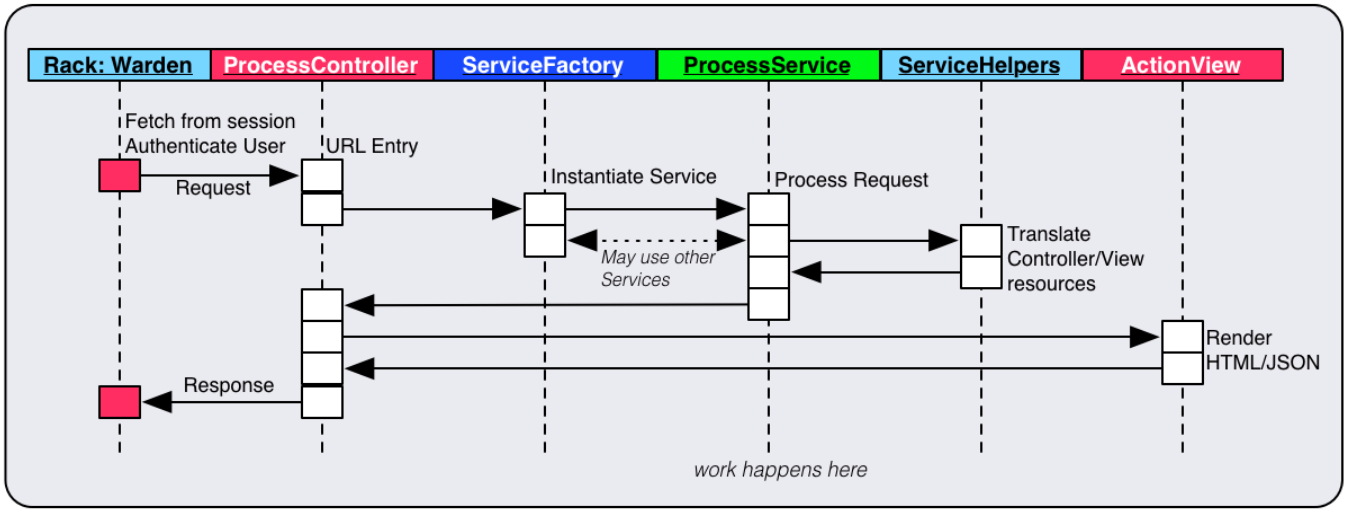
```
ProfileType < ActiveRecord::Base
  has_one :content_profile
  validates_presence_of :name, :description

  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 means 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 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 htm 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 possib 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 are needed, we can make those changes without great concern or constraints being imposed by the delivery mechanism. This also facilitates easi upgrades of the delivery frameworks; since we have reduced the number of touch points between our application the Rails.

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 nee 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 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 strategy.domains themselves are focused on the core business logic of a single process. Then yes, there will be many or several strategy.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 strategy.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. Aga 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 ServiceRegistry < ::Registry::RegistryBase


  ##
  # Application Services used in Controller methods
  ##

  def access_service
    @sf_access_service ||= ::AccessService.new({registry: 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::NestedResult.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::NestedResult.new({
                                  success: false,
                                  message: e.message,
                                  page_users: []
                               })
  end

  ...
end
```

```
 class ContentProfileDomain
```

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 tha 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

UserProfile

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 `Registry::ObjectStorageService`, capable keeping references to memory objects across the users request/response cycle; much like the session does and for the same reasons. i.e some objects a 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_authentica...`	Locate a user with these credentials in whatever user store is available
`:fetch_remembered_us...`	Locate a user using their remember token
`:fetch_cached_user(t...`	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_exp...`	Update the last time user access the system and if expired, revalidate their credentials.
`:authenticate?(unenc...`	Bcryts Authenticate returns self, we need to override that return value to return self instead
`:enable_authenticati...`	After successful login, this method saves the user object into our inMemory storage for later session level retrieval. Also upda :last_access.
`:disable_authenticat...`	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(context)`	Retrieve the list of all users without logging them in. For use with offline operations.
`:page_user(uname, co...`	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)`

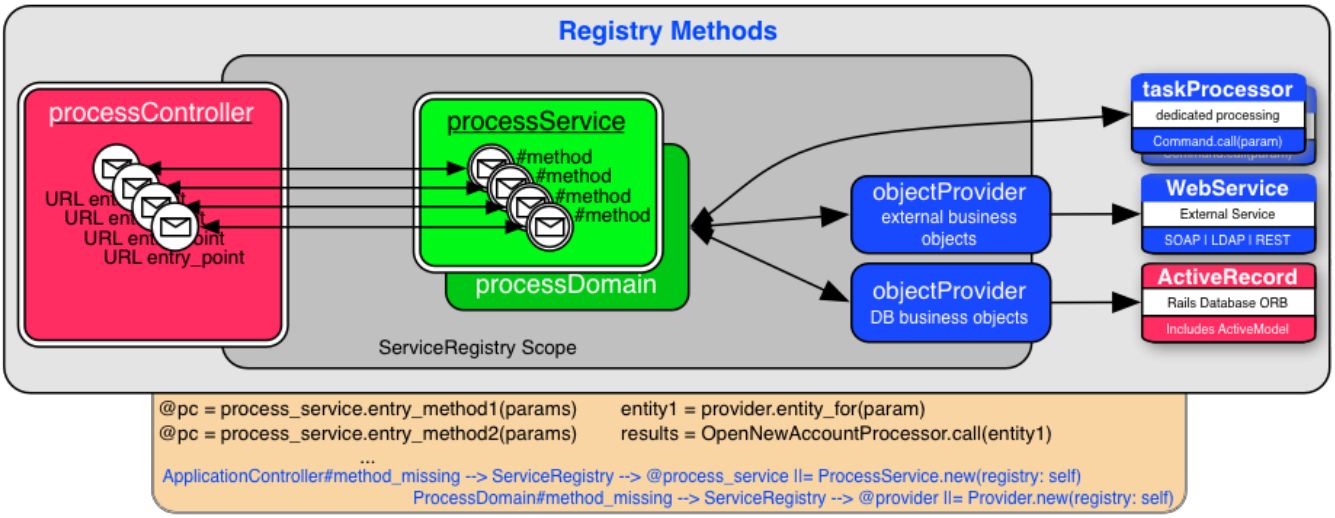
Instance methods support ApplicationHelper methods for use in view and controller actions:

- `: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] ServiceRegistry



ServiceRegistry

Instantiate ServiceRegistry 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 into the session just before the controller exits back to the browser. The ServiceRegistry is lightwieght and would derive no real benefit from being serialized into a session. So it is simply created anew with each request/response cycle.

The ServiceRegistry itself is instantiated as `@service_registry` in the top level ApplicationController, behind a memotizing method `service_registry`. To instantiate a processService use: `process_service.service_method_name(params)`.

The intended use of the `service_registry` is to provide lazy instantiation and memotization of all strategy components classes in the application. And to additionally provide them critical facilites, like `:current_user`, and act as an isolation gateway to the application controller.

Inside a processService, like ContentService or ProfileProviders, reference the service_registry as `:registry` or `:service`, as in: `registry.content_service.some_method` .

Inheritance ServiceRegistry, inherits from `::Registry::RegistryBase` which yields a memory-based Object Storage Interface Module `Registry::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 ServiceRegistry provides Services with the `:current_user`, `:registry` 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_registry devise a solution to their requirements. The 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_registry 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 locate 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 strategy.services and strategy.domains.

Engineering objectives shall include, avoidance of the controller in all strategy.services, strategy.domains, and specialty business strategy.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: `registry.page_action_paths(paths)`

All Components, like ContentService, AccessService, DBProfileProvider, XMLProfileProvider, FileSystemAdapter, and InlineValuesAdapter are instantiated via the service_registry facilities. And primary utility objects can also be supported via the service registry.

Primary Component of this Services Strategy. Providing access to all application strategy.services!

[Services Strategy] processServices

processServices

Instantiate ServiceRegistry provides instantiation strategy.services to any requester in the request/response cycle.

The intended use of processServices 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::NestedResult` group of result value containers.

Inheritance processServices inherit business logic from the `processDomain` class. The processDomain class inherits from the `::Registry::DomainsBase`, which yields common initialization strategy.services like the current_user() and access to the service_registry for interaction with the controller helpers if needed. In particular, the `service_registry` includes object storage strategy.services and session storage strategy.services through its inheritance chain.

Provides processServices present 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` value container, and all exceptions are trapped by this top level method. PageControl value containers are required to include `success: true|false, message: error_message|success_message|empty, and payload: [domainData]` values a minimum.

JSON Api or regular HTML methods are hosted by this service.

Instantiated by: `service_registry()`

[Services Strategy] processDomains

processDomains

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

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

Inheritance processDomains inherit directly from the `::Registry::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 processDomains have logical access to the service registry and all other peer strategy.services the registry manages. Typically you would find two to three levels of methods in a Domain class.

- **Level One** Top level business interface. Methods at this level take responsibility for doing the whole process step; process a order or payment, would be an example of this.
- **Level Two** The notion of the whole process setp, is likely to have component tasks involved. At this level methods are expected to perform one 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. To that end, we include design components to meet the most frequent use cases; name `domainProviders`, and `taskProcessor`.

The Business Process Interface is here.

Inherited by: `ContentService, ...`

[Services Strategy] domainProviders

domainProviders

Instantiate ServiceRegistry provides instantiation strategy.services to any requester in the request/response cycle.

The intended use of a domainProvider object, is to handle all business object persistence related operations for domain context data. It should not interac directly any object aside from the domain object that calls it, and the Database Interface.

Inheritance domainProviders inherit common instantiation service from the `Providers::ProvidersBase` class. The Providers::ProvidersBase yields common initializati strategy.services, and object storage strategy.services.

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

domainProviders by design should handle contextual IO for all or a subset of business objects. Returning results as a attribute hash or ValueBean, never returning the underlying ActiveModel object.

Currently there are two implementations of this component: XMLProfileProvider and DBProfileProvider classes.

Instantiated by: `service_registry()`

[Services Strategy] Processors

Processor

Instantiate ServiceRegistry provides instantiation strategy.services to any requester in the request/response cycle.

The intended use of a Processor is to handle specialized tasks which may include external IO of some sort, for Domains.

Inheritance A task processor inherits directly from the `Processors::ProcessorBase` class. The ProcessorBase yields common initialization strategy.services, and acce to the service registry.

Provides FileSystemAdapter provides logical access to content storage facilities.

taskProcessor by design are expected to handle a complete and single task step of a larger process flow, and have no dependencies. However to comple their assigned task they may engage other task processors, strategy.services, and providers; thus they do benefit from access to the

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

Instantiated by: `service_registry()`. However, they are capable of operating independently of the

[Services Strategy] NestedResult or DottedHash Value Containers

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

Initialized with a Hash of key value pairs: `res = SknUtils::NestedResult.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, arrays of hashes, and arrays of array of hashes.

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::NestedResu can be done using `dot` notation, or `hash` notation: example: `res.one #=> 1`, `res.two.first.two #=> 'two'`

If you need to unwrap the container 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::NestedResult Used directly to transport values to controller method.

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.

It is initiated by an `ApplicationHelper#do_page_actions` and assumes it input array is in either `@page_controls.page_actions or .package.page_actior`

In your strategy.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

The directory `spec/support` has a collection of modules and classes to augment testing:

- `ServiceRegistryMockController` A class used to replace the need for a ApplicationController to initialize the ServiceRegistry with. It contains methods that Service or Domain objects might call on the ApplicationController, like: current_user, page_actions, menu_active, etc. The use of this mock controller has a significant speed increase compared to alternative **Integration/Controller** tests.
- `TestUsers` contains a collection of methods that will return a testable user_profile object, either fully enabled as the current_user, or simply loaded as an independent record.
- `TestDataSerializers` contains a couple of methods that allow you to save objects to a separate yaml file. Can be used to capture a `@page_controls` object to be used latter as mocking data when testing views.

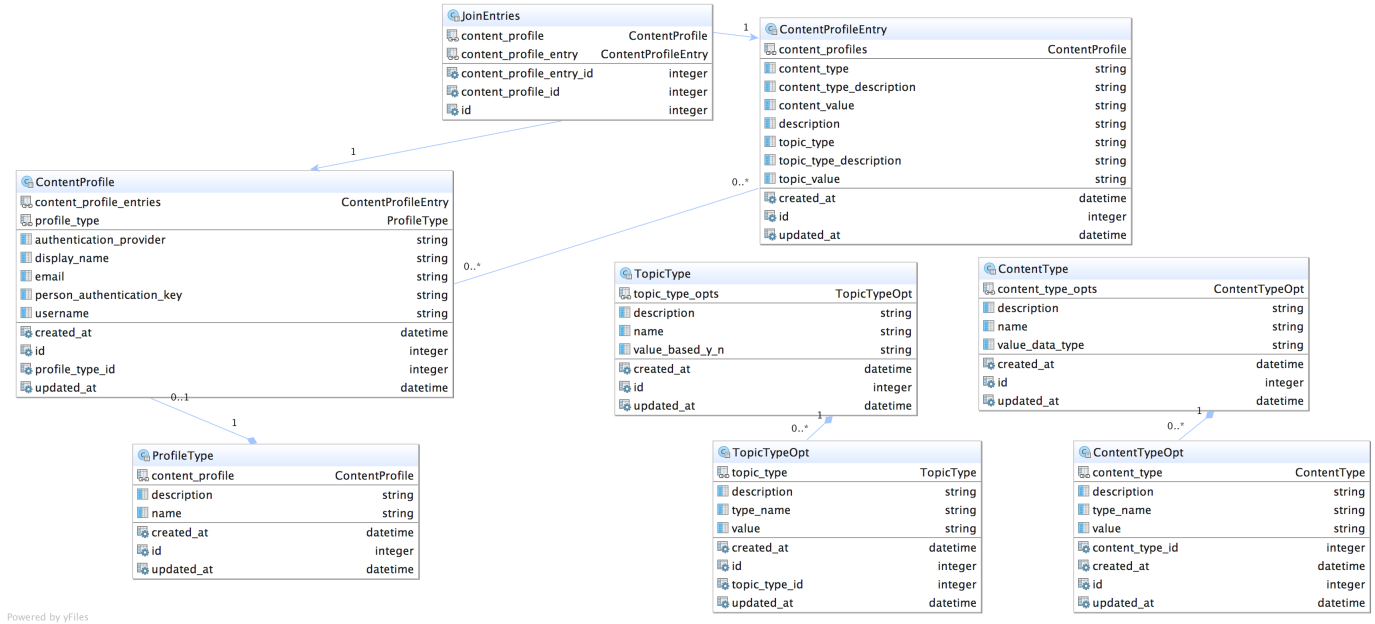
Testing Services with RSpec looks like this:

```
RSpec.describe ContentService, "Service routines of ContentService." do
  let!(:user) {user_bstester}

  let(:mc) {ServiceRegistryMockController.new(user: user)}
  let(:service_registry)  { ServiceRegistry.new({registry: mc}) }

  let(:service) {service_registry.content_service}

  context "Content Profile Management methods return proper results. " do

    scenario "#handle_content_profile_management prepares a page package of all users" do
      result = service.handle_content_profile_management({})
      expect(result).to be_a(SknUtils::NestedResult)
      expect(result.success).to be true
      expect(result.message).to be_blank
    end
```

Finished in 10.76 seconds (files took 3.44 seconds to load)
223 examples, 0 failures
Coverage report generated for RSpec to ./coverage/index.html 1275 / 1583 LOC (80.54%) covered.

Authorization Data Models

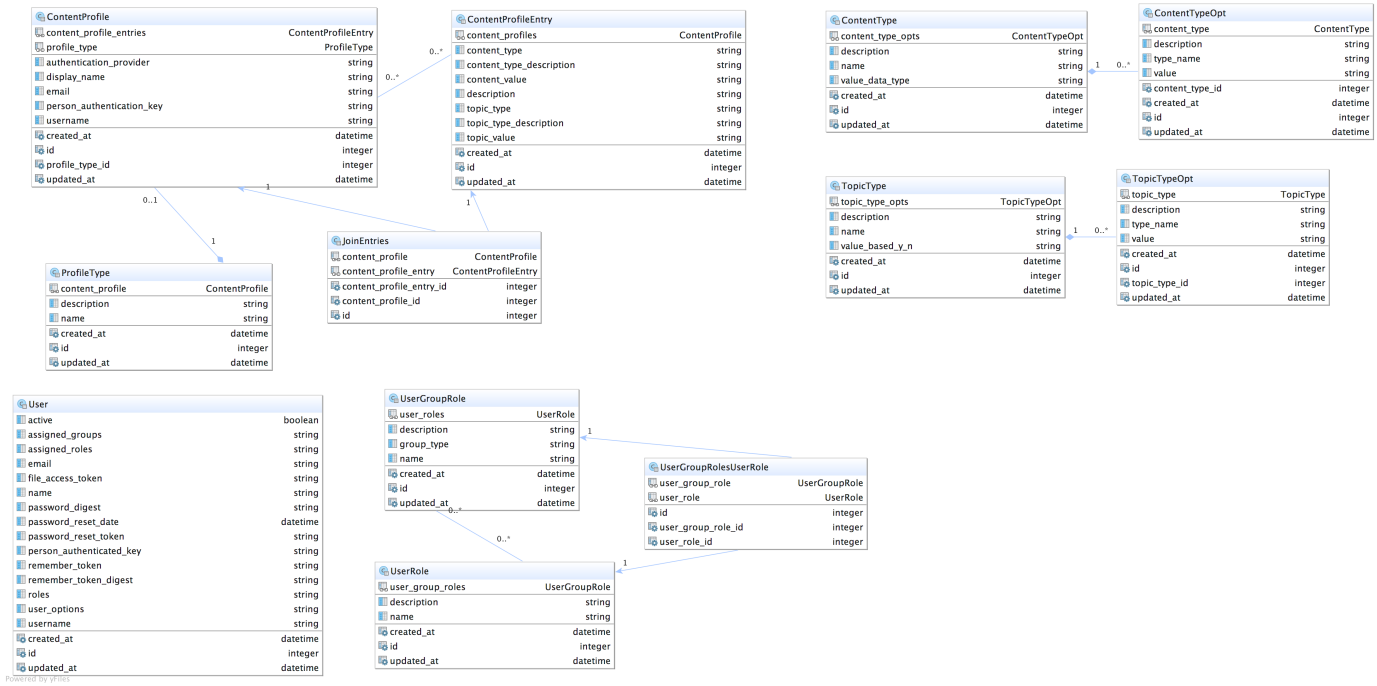
Authorization Data Models



Authorization Data Models

Application Data Model

Application Data Model



Authentication and Authorization Data Models

To Do List

To Do

CommandBase Processors have two types of implementations based on their intended purpose: Single or Multi Method mode. Mode is implementation via inheritance; `Processors:ProcessorBase` class is for multi-method, `Processors::CommandBase` class is for single-method implementation. Single mode Processors wou live outside of the ServiceRegistry umbrella and be invoke directly by Domains when needed; they are desiged to be invoke via: `myCommandProcessor.call(params_hash)`

Profile Runtime A new class to handle runtime requests for user ContentProfiles and authorization queries, rather than bolting these methods on the UserProfile. While the features are present in the code, I'd like to organize them better.

AccessRegistry There are a lot of direct calles to the `Secure:AccessRegistry...` class which I would like to corral into a AccessRegistry provider or processor.

Clarify There are three significant and wonderful components in this codeset; as depicted on the homepage.

1. Deep authentication implementation via UserProfile class, backed by Warden and Rack-Attack.
2. Deep authorization coverage of all processes and accessible elements via AccessProfile classes, backed by Warden and ApplicationController.
3. Deep authorization coverage of secured-content based resources via ContentProfile XML or Database classes.

For which I need to expose as usable features in your app.

Todo List: Items I want to spend more time on