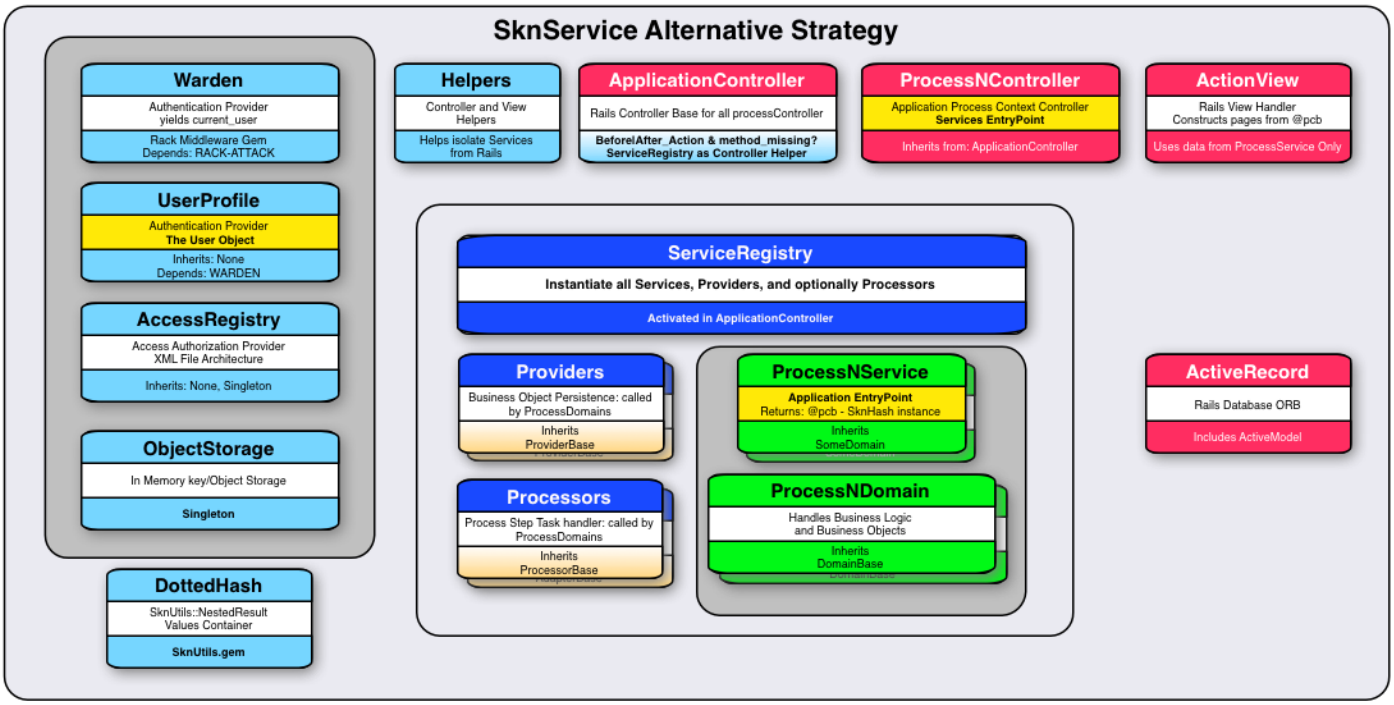
Architectural Details



Design Guidance: SknStrategy

DO NOT FOLLOW THE RAILS WAY.

The MVC model is a classic and used with success, as is, for many applications. However, when the project's code-set exceeds a certain size or feature-set mix, your ability to grow th codebase by adding new features and/or upgrading to the next release of Rails can be severely hampered. This is no surprise as MVC was conceived for small Web Applications.

Understanding these growth issues, it is recommended that you apply a more traditional approach to building applications that use the Web, by isolating or treating Rails as a Web Interface as much as possible.

Although there are many application architectures and patterns to choose from, isolating Rails APIs yields the greatest value. These five actions will deliver that value:

Remove from Control…	Create Service or UseCase classes to contain the code you would otherwise put into Rails Controller URL-Methods.
Remove from Models	Collect all ActiveRecord invocations and methods into a one or more classes which follow the Repository or Provider pattern, suitable for dependency injection.
Remove from Views	Limit controller instance variables to one nested hash of pure ruby values. UseCase methods should return this object as the sole results object to be rendered as HTML or JSON to requester. Views must never cause the execution of a non-helper method or follow the dotted-path of an ActiveRecord object, the data available to a view should be a value. As an example, views should format a time object for display but they should not execute `Time.now` to get the object to format.
Wrap APIs in helpers	Create a few View helpers to aid in the HTML rendering or controller interface; to minimize Rails View API changes and the number of partials needed.
Use seperate Directory	Establish a namespaced directory structure to contain your application components; possibly `app/strategy`

Having your business or application logic in PORO class containers, improves your understanding of its components and your ability to add or enhance features. Depending on how successful you were in separating your application code from Rails you may be able to run your full test suite by injecting memory-based dependencies for testing rather than Rails based dependencies; and never even startup Rails!

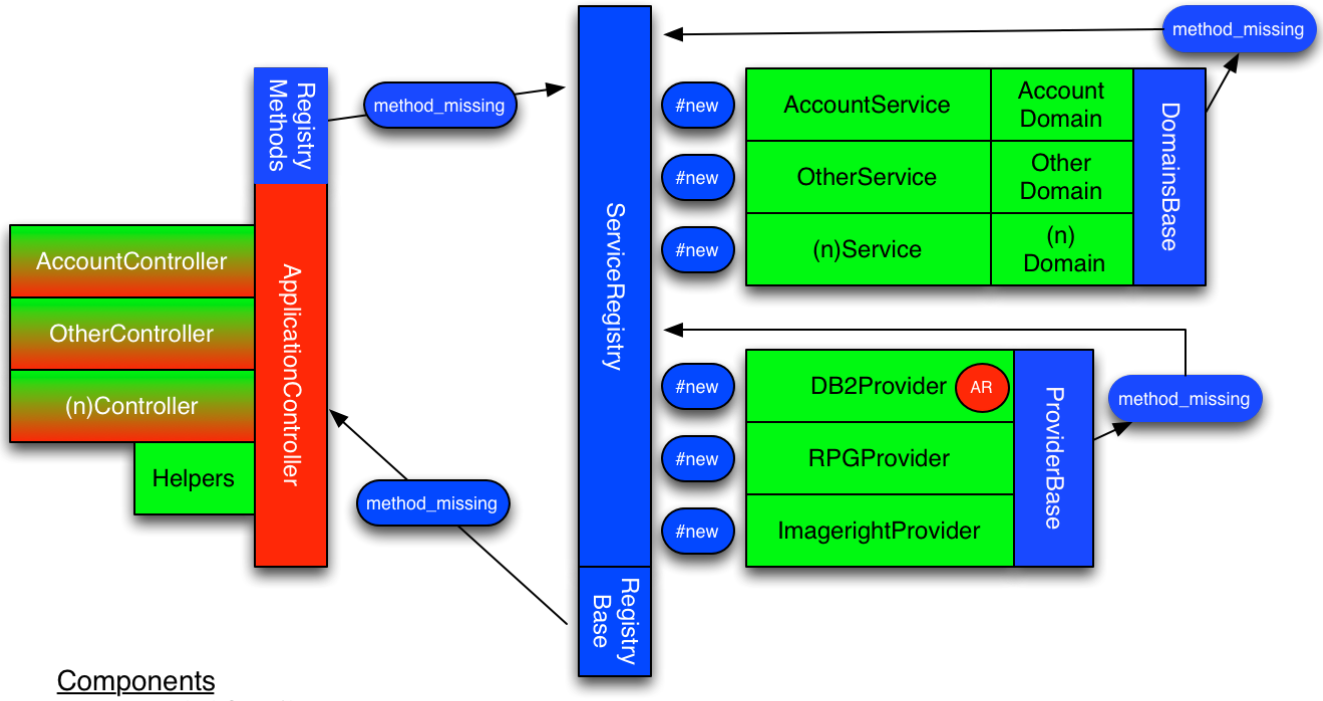
However, the **SknStrategy** is not another Web Framework! It is a thought provoking alternative to the Rails Way. An alternative that still builds applications using the full capabilities of Rails, while keeping your business logic components separate from Rails; and enables you to easily add or change features.

SknServices serves as a model for how Ruby applications can be engineered to use the Web 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: Containers

SknStrategy offers six loosely-coupled structures to contain your application code and interconnect Rails and other DataSources.



Components

* URL Entry Method: Start of business logic request.
* ServiceRegistry: Instantiates all Services and Providers that represent core business logic.
* AccountService: Handles request validation, invokes core logic, and packages response.
* AccountDomain: Contains core business logic, collects info from Providers as needed. Inherited by Services.
* DB2Provider: Contextual collection of business entities, grouped by domain if required.
* Helpers: Rails controller apis wrapped by methods, method_missing routing provides access to these helpers from Service objects.

ServiceRegistry Class

A simple PORO with named-methods that will instantiate each UseCase, Provider, and Processor class on demand and with any mix of dependencies needed. Many applications can have hundreds of entry points or controller URL methods. ServiceRegistry centralizes the instantiation of classes to avoid coding those initiations inline to the controllers method. Additionally, each class can be initiated with a reference to the registry as a parallel to dependency injection.

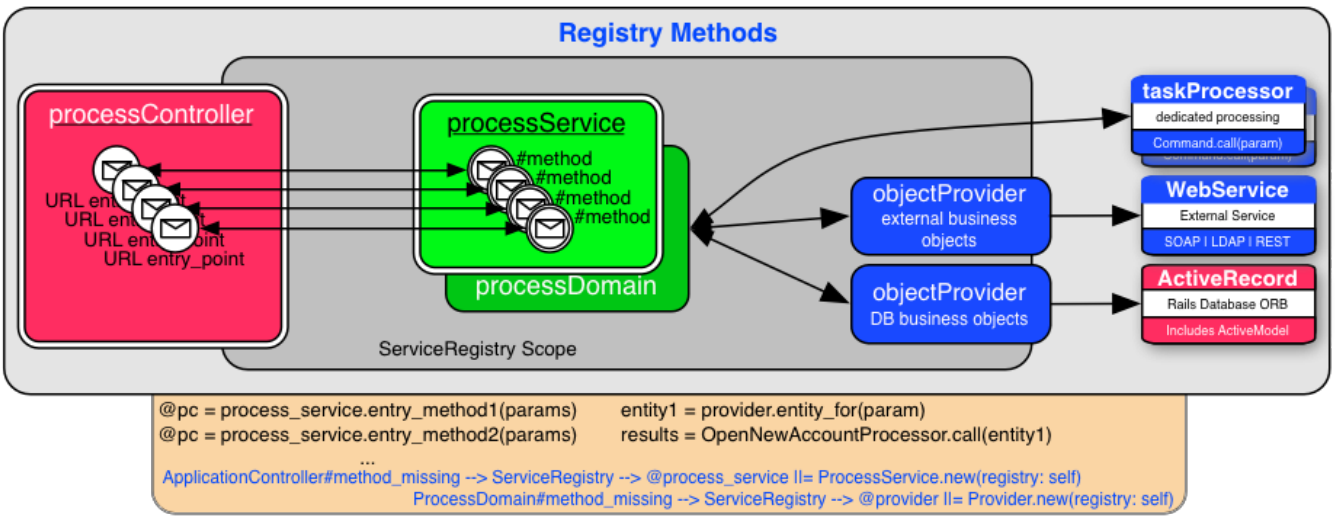
Finally, theServiceRegistry is wired into the controller via `#method_missing`, which enables the controllers url-method to use a registry-method-name of the desired class. There will be only one ServiceRegistry per Application or Engine.

```
class ApplicationController < ActionController::Base

  def service_registry
    @service_registry ||= Services::ServiceRegistry.new({registry: self})
  end

  # Easier to code than delegation, or forwarder
  def method_missing(method, *args, &block)
    Rails.logger.debug("#{self.class.name}##{__method__}() looking for: #{method.inspect}")
    if service_registry.public_methods.try(:include?, method)
      block_given? ? service_registry.send(method, *args, block) :
          (args.size == 0 ?  service_registry.send(method) : service_registry.send(method, *args))
    else
```

```
      super
    end
  end

  ...
end


class ProfilesController < ApplicationController

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

  # Or with Html Wrapper
  def in_action_admin
    wrap_html_response( content_service.handle_in_action )
  end

  # Or with JSON wrapper
  def api_accessible_content
    wrap_json_response( content_service.handle_api_accessible_content(params.to_unsafe_h) )
  end

  ...
end


module Services
  class ServiceRegistry < ::Registry::RegistryBase

    def content_service
      @sr_content_service ||= ::ContentService.new({registry: self})
    end

    def password_reset_use_case
      @sr_password_reset_use_case ||= ::UseCases::PasswordResetUseCase.new({registry: self})
    end

    def db_profile_provider
      @sr_db_profile_builder ||= ::Providers::DBProfileProvider.new({registry: self})
    end

    def content_adapter_file_system
      @sr_content_adapter_file_system ||= ::Processors::FileSystemProcessor.new({registry: self})
    end

    ...
  end
end


module Registry
  class RegistryBase
    include Registry::ObjectStorageService

    attr_accessor :registry

    def self.inherited(klass)
      klass.send(:oscs_set_context=, klass.name)
      Rails.logger.debug("#{self.name} inherited By #{klass.name}")
      nil
    end

    def initialize(params={})
      params.keys.each do |k|
        instance_variable_set "@#{k.to_s}".to_sym, nil
        instance_variable_set "@#{k.to_s}".to_sym, params[k]
      end
      raise ArgumentError, "#{self.class.name}: Missing required initialization param!" if @registry.nil?
```

```
      nil
    end

    # Not required, simply reduces traffic since it is called often
    def current_user
      @current_user ||= registry.current_user
    end

  private

    # Easier to code than delegation, or forwarder; @registry assumed to equal @controller
    def method_missing(method, *args, &block)
      Rails.logger.debug("#{self.class.name}##{__method__}() looking for: #{method}")
      if registry.public_methods.try(:include?, method)
        block_given? ? registry.send(method, *args, block) :
            (args.size == 0 ?  registry.send(method) : registry.send(method, *args))
      else
        super
      end
    end

  end
end

module UseCases
  class UseCaseBase

    attr_accessor :registry

    def initialize(params={})
      params.keys.each do |k|
        instance_variable_set "@#{k.to_s}".to_sym, nil
        instance_variable_set "@#{k.to_s}".to_sym, params[k]
      end
      raise ArgumentError, "#{self.class.name}: Missing required initialization param!" if @registry.nil?
    end

    def self.inherited(klass)
      Rails.logger.debug("#{self.name} inherited By #{klass.name}")
    end

  private

    # Easier to code than delegation, or forwarder
    def method_missing(method, *args, &block)
      Rails.logger.debug("#{self.class.name}##{__method__}() looking for: #{method}")
      block_given? ? registry.send(method, *args, block) :
          (args.size == 0 ?  registry.send(method) : registry.send(method, *args))
    end

  end
end

# Services Base thru Inherited Domain
module Domains
  class DomainsBase

    attr_accessor :registry

    def initialize(params={})
      params.keys.each do |k|
        instance_variable_set "@#{k.to_s}".to_sym, nil
        instance_variable_set "@#{k.to_s}".to_sym, params[k]
      end
      raise ArgumentError, "#{self.class.name}: Missing required initialization param!" if @registry.nil?
    end

    def self.inherited(klass)
      Rails.logger.debug("#{self.name} inherited By #{klass.name}")
    end

  private
```

```
    # Easier to code than delegation, or forwarder
    def method_missing(method, *args, &block)
      Rails.logger.debug("#{self.class.name}##{__method__}() looking for: #{method}")
      block_given? ? registry.send(method, *args, block) :
          (args.size == 0 ?  registry.send(method) : registry.send(method, *args))
    end

  end
end
```



UseCase/Services Class

UseCase classes are assumed to have multiple entry-point methods related to the page/request-oriented steps of the usecase. Service classes are generally expected to have only on entry-method, which is suggested to be `#call`. Whichever pattern you choose the entry-method is *required to*:

1. catch any exception thrown in this code path and guarantee the controller a structured response
2. do the work necessary to compose an appropriate response to the the request, using any Ruby pattern deemed feasible.
3. return a nested-hash response object with these minimum keys `:success, :message, :payload`

Services and/or UseCase classes should consider using inheritance from a domain class to facilitate code reuse between entry-point methods. Method calls to other peer UseCase or Services objects is encouraged. An application will have many UseCases and/or Services classes contextually named, such as PolicyManagementUseCase, or NewQuoteService.

```
module UseCases
  class PasswordResetUseCase < UseCaseBase

    def empty_user_to_get_username
      user = User.new
      SknUtils::NestedResult.new({success: user.present?, user: user, message: ""})
    rescue => e
      SknUtils::NestedResult.new({success: false, user: nil, message: e.message})
    end

    ...
  end
end

module Services
  class ContentService < Domains::ContentProfileDomain

    def handle_in_action_admin(params={})
      package = get_page_users(PROFILE_CONTEXT)
      SknUtils::NestedResult.new({success: package.present?, message: "", page_users: package})
    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
end

module Domains
  class ContentProfileDomain < DomainsBase

    def member_update_package(params)
      choices = db_profile_provider.parse_member_update_params(params)
      success = db_profile_provider.apply_member_updates(params['id'], choices)
      {
          success: success,
          message: (success ? "Update Completed" : "Update Failed"),
          package: {}
      }
    end

    ...
  end
end
```

Provider/Repository

Like UseCases and Services classes, Provider and Repository classes support different expectations and are usually implemented differently with the exception that both provide a boundary between application code and ActiveRecord or an external subsystems. The Repository pattern may have a one-to-one mapping to ActiveRecord tables with the CRUD-like semantics. The Provider pattern is slightly more aggressive in that it uses the semantics of the UseCases business language and does all the work that step-element needs done; both exists to isolate ActiveRecord calls to one ruby module and be the provider of data operations over SQL or WebService subsystems. The SknStrategy encourages the containment of external subsystems calls in a module separate from your core application logic; you are free to use any pattern that achieves that goal. An application will have many Providers contextually named, such as PolicyProvider, or QuoteProvider.

```
module Providers
  class DBProfileProvider < ProvidersBase

    def create_content_profile_entry_for(cpe_desc, topic_choice, content_choice)
      cvs = content_choice.is_a?(Hash) ? content_choice : content_choice.first
      tvs = topic_choice.is_a?(Hash) ? topic_choice : topic_choice.first
      ContentProfileEntry.create!({
          description: cpe_desc,
          content_value: cvs[:opts].map {|v| v["value"] }.flatten,
          content_type: cvs[:type]["name"],
          content_type_description: cvs[:type]["description"],
          topic_value: tvs[:opts].map {|v| v["value"] }.flatten,
          topic_type: tvs[:type]["name"],
          topic_type_description: tvs[:type]["description"]
      })
    end

    ...
  end
end
```

Processor

This class does the heavy lifting for UseCases or Services classes. In the course of preparing a response to the URL request a UseCase object may need several specific tasks to be completed. Rather than have those statements inline to the UseCase class, the Processor class can implement those statements. Thinking about how a UseCase might break its work into multiple steps, the processor can extend the UseCase by implementing one or more steps. Processors often perform their operations against external Web/Restful APIs, EMailers, Document Storage, or other subsystems. An application will have many Processors contextually named, such as AttachmentsProcessor, or RatingProcessor.

```
module Processors
  class FileSystemProcessor < ProcessorBase

    def preload_available_content_list(cpe)
      result = []
      catalog = {}
      content_type = cpe[:content_type] || cpe["content_type"]  # should always be an array

      content_list(cpe, catalog, result)
      update_storage_object("#{PREFIX_CATALOG}-#{content_type}-#{cpe[:pak]}", catalog)
      Rails.logger.debug "#{self.class}##{__method__} CATALOG: #{catalog.keys},   Result: #{result.present?}"

      result
    rescue Exception => e
      Rails.logger.warn "#{self.class.name}.#{__method__}() Klass: #{e.class.name}, Cause: #{e.message} #{e.backtrace[0..4]}"
      []
    end

    ...
  end
end
```

Helpers

Rails Controller Helpers, are written to increase readability and exploit the code consistency at Rails API touch points, on behalf of UseCases and also view based operations. Examine the `#wrap_html_response()` later for the best example. For every url-entry-method the controller expects to generate a html/json response or redirect to a different view. UseCases always return one nested hash object as the data payload, a ruby wrapper has been created to issue the render or redirect command to the controller passing the payload an notificati message if available

The SknUtils gem contains a NestedHash class which add dot notation to the hash returned by UseCases to the controller. There are several similar gems available to package and validate the request params, responses, and to create value objects.

```
module Registry
  module RegistryMethods

    protected

    def wrap_html_response(service_response, redirect_path=root_path)
      @page_controls = service_response
      flash[:notice] = @page_controls.message if @page_controls.message.present?
      redirect_to redirect_path, notice: @page_controls.message and return unless @page_controls.success
    end

    def wrap_html_and_redirect_response(service_response, redirect_path=root_path)
      @page_controls = service_response
      flash[:notice] = @page_controls.message if @page_controls.message.present?
      redirect_to redirect_path, notice: @page_controls.message and return
    end

    def wrap_json_response(service_response)
      @page_controls = service_response
      render(json: @page_controls.to_hash, status: (@page_controls.package.success ? :accepted : :not_found), layout: false, content_type: :json) and return
    end

    ...
  end
end
```

Directory Layout

The objective is to have physical separation for Rails traditional directories, and to have each SknStrategy component be uniquely namespaced.

Any directory under 'app' will receive Rails full treatment of eager-loading and auto-loading. I've tried other directories to anchor these new containers only to be forced into fixing or working around Rails Autoload feature. Anchoring at 'app/strategy' resolves all the autoload issues, and gives physical separation from the regular Rails directories.

In terms of namespacing, 'app/strategy' is a freeby; Rails does not consider these directories as a part of an eventual namespace. 'app/strategy/use_cases', the use_cases portion is a part of the namespace; `UserCases::SomeModule`. The following directory layout is what I have used without conflict.

```
app/
    ├── assets
    ├── controllers
    │   ├── application_controller.rb
    │   ├── pages_controller.rb
    │   └── profiles_controller.rb
    ├── helpers
    │   └── application_helper.rb
    ├── strategy
    │   ├── domains
    │   │   ├── access_profile_domain.rb
    │   │   ├── content_profile_domain.rb
    │   │   └── domains_base.rb
    │   ├── page_actions_builder.rb
    │   ├── processors
    │   │   ├── file_system_processor.rb
    │   │   ├── inline_values_processor.rb
    │   │   └── processor_base.rb
    │   ├── providers
    │   │   ├── db_profile_provider.rb
    │   │   ├── providers_base.rb
    │   │   └── xml_profile_provider.rb
    │   ├── registry
    │   │   ├── registry_base.rb
    │   │   └── registry_methods.rb
    │   ├── services
    │   │   ├── access_service.rb
    │   │   ├── content_service.rb
    │   │   └── service_registry.rb
    │   └── utility
    └── views

spec/
    ├── rails_helper.rb
    ├── spec_helper.rb
    ├── strategy
    │   ├── domains
    │   │   └── content_profile_domain_spec.rb
    │   ├── page_actions_builder_spec.rb
    │   ├── processors
    │   │   ├── file_system_processor_spec.rb
    │   │   └── inline_values_adapter_spec.rb
    │   ├── providers
    │   │   ├── db_profile_provider_spec.rb
    │   │   └── xml_profile_provider_spec.rb
    │   ├── services
    │   │   ├── access_service_spec.rb
    │   │   └── content_service_spec.rb
    │   └── utility
    └── support
        └── service_registry_mock_controller.rb
```

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

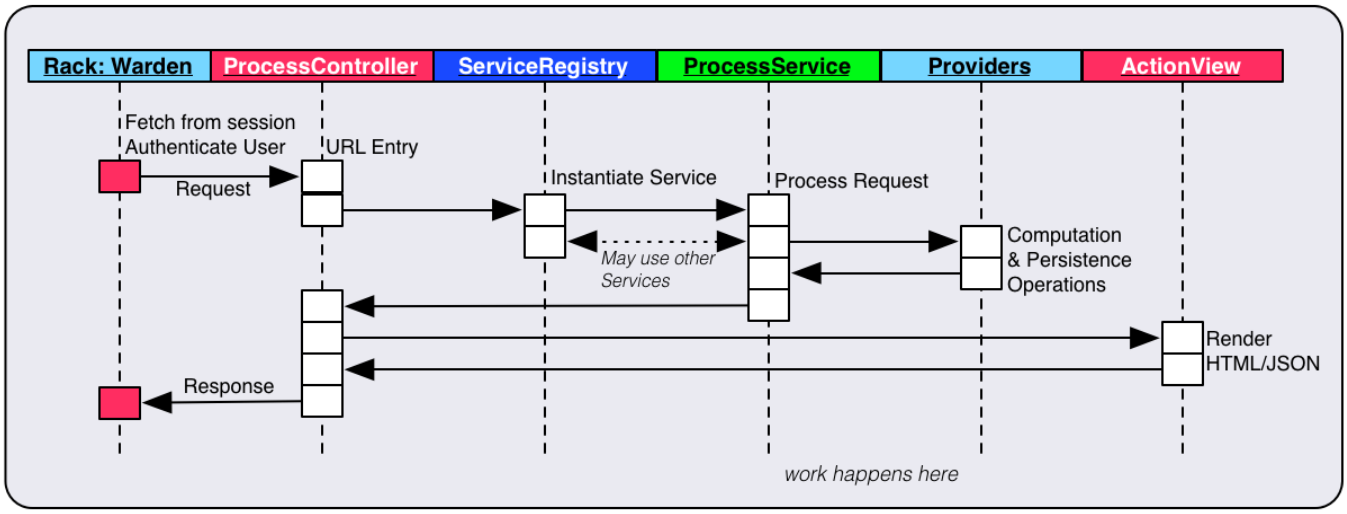
```
class ContentProfile < ApplicationRecord
  belongs_to :profile_type
  has_and_belongs_to_many :content_profile_entries, inverse_of: :content_profiles, :join_table => :content_profiles_entries
  accepts_nested_attributes_for :content_profile_entries, :profile_type, allow_destroy: true, reject_if: lambda {|attributes| attributes['description'].blank
  validates :person_authentication_key, uniqueness: true, on: [:create, :update]

  ...

  def entry_info
    {
        pak: person_authentication_key,
        profile_type: profile_type_name,
        profile_type_description: profile_type_description,
        provider: authentication_provider,
        username: username,
        display_name: display_name,
        last_update: updated_at.strftime("%Y-%m-%d %I:%M:%S %p"),
        email: email,
        entries: content_profile_entries.order([:topic_type,:content_type]).collect(&:entry_info)
    }
  end
end
```

In the case of models this means only short methods to help with form option lists, the model's own validation methods, and `:entry_info` like methods that return the logical entity without AR identifiers. Nothing more!

SknStrategy organizes your application code so you can focus on engineering the application using software patterns you think appropriate.

Design Guidance: Message Flows

Transpose your UseCases into a sequence of url-request steps. Each url-request will flow through your application code in a fashion similar to the following.



This strategy provides basic containers for your application logic, by making instances of application classes available from the ServiceRegistry, centralizing the initialization of core classes, and establishing a directory structure. However, you have the freedom to implement your business logic using any application architecture you choose; Clean, Hexagonal, Domain-Driven, etc. This strategy should be considered basic wiring for interconnecting your application to the Rails Web Framework.

Instantiating objects through the ServiceRegistry, which all base classes have access to, potentially removes the need for Dependency Injection; since each UseCase or Services class can invoke methods on any peer by using its registry shortname. Peer to Peer calls by UseCases or Services do raise a potential packaging issue. Services invocation typically return a nested-hash wrapped in a dotted-hash class. This is certainly usable as is, but maybe not your preference. However, consider why you might want to make a peer-to-peer call if you could reach their reusable processor tasks and non-critical data operations directly through a Provider or Processor. Peer-to-Peer traffic will be extremely low or non-existent in practic and should only occur when you need the change the state of the targeted object, all query-based requests should be directed to the appropriate provider or processor.

I've looked long and hard at the Repository pattern as an interface to the SQL subsystem or ActiveRecord and decided each time to use the SknStrategy's Provider pattern. Most repository implementations I've reviewed tend to use the CRUD semantics and handle only one record at a time. In practice I've found that I often need to process collection or Aggregate of objects at once and all the objects may not be from the same SQL Table. The Provider pattern has the freedom to treat that request as a single unit of work and complete in one call from the UseCase.

Larger application might provide fifty to one-hundred controllers, if each has ten url-entry methods that will result in five-hundred different entry-points into you application logic. This scaling issue fits well into the SknStrategy. UseCase classes or Services classes are the targets for those five-hundred url-entries, code reuse opportunities will be represented in the Domain classes the Services and UseCases inherit from along with the contents of Providers and Processors. It is hoped that reasoning about where a particular feature is located will be easier to follow, once the basic of this strategy have been adopted.

Rails Engines and Ruby Gems as a packaging strategy for application code. My experiences, thus far, highlights **NameSpacing** as the most valuable property of large applications usi Rails. Ruby Gems come in second as a valid option for code separation, if they too are namespaced. My current corporate apps use multiple engines as a legacy artifact and as technical debt from co-existing with Rails for years. I would not choose engines again, knowing what I know about NameSpacing (a Ruby Feature BTW). Looking forward from today, i the application becomes too large for one Rails app -- at that point I would recommend breaking the one into two parts; but only when the time is at hand. *In no scaling situation would suggest switching from Rails to another Ruby Web Framework*, if you started with its Rails best to stay with Rails.

However, if you have the opportunity to start a new web-server based application I would give serious consideration to the Roda (http://roda.jeremyevans.net) gem as your web-interfa component, with Dry-Rb (http://dry-rb.org) and the Rom-Rb (http://rom-rb.org) gems included in your application components. I have a complementary starter program that you can evaluate which uses Roda, SknStrategy, and Rom-Rb called SknBase (https://skoona.github.io/skn_base/) on github.

SknServices implements the SknStrategy as an example application.

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

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

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 14.94 seconds (files took 22.17 seconds to load)
244 examples, 0 failures
Coverage report generated for RSpec to ./coverage/index.html 1803 / 1967 LOC (91.66%) covered.

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, Providers, or Processors when needed; they are desiged to be invoke via: `myCommandProcessor.call(params_hash)`

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

Clarify There are three significant and advanced 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

Architectural Information (/pages/details_architecture)

SknServices Release Version: 2.3.0