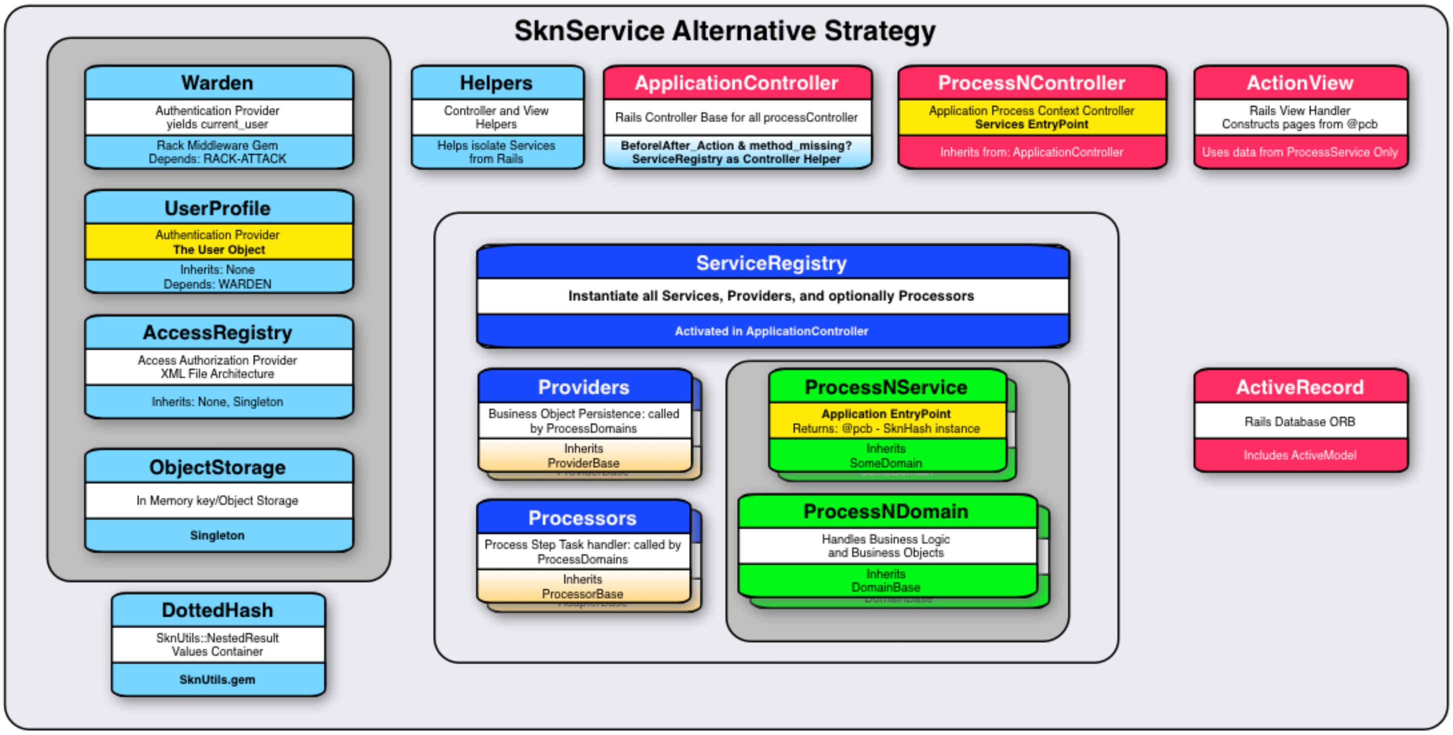
About SknServices



The SknServices project exists to explore Authentication and Authorization strategies. With special focus on content-based authorization using expanded Permissions properties. It also implements the **SknService Development Strategy** for the target application which lessens the dependency on Rails.

Core Capabilities

Authentication

UserProfile

The Warden Gem is used for its highly secure and flexible rack-based security engine. Rack-Attack gem augments Warden's security management features.

A unique ID, Personal Authentication Token(PAK), is assigned to each user and persisted internally forever. This id is used to coordinate the collection of user profile attributes and permissions used to enable the users access to the system once they are authenticated. Authentication is handled by Warden Strategie capable of user/password, http basic auth, single-sign-on tokens, and several more.

Remember me, secure email based password-reset, and session timeout are also enabled; but considered application-based features or optional.

The UserProfile class centralizes authentication and authorization methods and/or services into a single user class. The current user is locally available from every Rails controller as :current_user. Additionaly, a UserProxy class exists for use by the UserProfile and is responsible for interfacing the the authenticatio provider. The application considers the UserProfile as the *user*!

Authentication is evaluated with every page request, except for images, javascript, and css script files; through automation in the ApplicationController.

The UserProfile coordinates the services of the AccessProfile and the ContentProfile, providing controlled and consistent access to business information.

Authorization: Access

AccessProfile

AccessRegistry.xml, encodes and/or assigns a URI to every page, action, and information element for which security is intended. All secured elements are assigned an unique URI id by a developer.

AccessRegistry class maps internal URIs to external String Roles; which can be managed in LDAP or similar systems. Users having a particular role would b allowed to interact the associated resource, using RESTFul or CRUD or semantics.

Access control methods are attached to the UserProfile object and thus available in all contexts of the application via :current_user. Application Pages are ea represented by combining the controller name and controller method name; or URL path. An example URI for the index page in the HomeController would b `home/index` . Actions which may exist on menus or page action buttons may be given URL labels like: '#Print', '#ManageAccount', '#ManageContacts', etc On-page information elements URIs may be '_RecentOrders', '_CanceledOrders', or '_OrderHistory'

Access permission roles collected for each user at login, might map URI='#ManageContacts' to Role='App.Manage.Contacts', giving that user the ability to manage contacts. The AccessRegistry method call would be: `userProfileObject.has_access?(URI) # ==> true`.

Access authorizations are evaluated for each *page access*, *clickable control*, and for each *showable data element* when the service is building the response. service omits info elements the user is not authorized for. This allow the view handler to blindly display all info it is given, knowing the service has removed n authorized info. Page Access authorization is automated by examining the the `#controller_name` and `#action_name` methods to compose the URI for t page. Clickable (menu items, buttons, etc) authorization requires the developer to use programming helpers to evaluate authorization. Showable data elemer have two levels of authorization available; *Clickable*, and *Content-based* authorization using expanded permissions, both use the AccessRegistry.

Authorization: Content

ContentProfile

A ContentProfile contains the collection of permission entries for a user identified via their PAK id. Each user is optionally assigned a content profile which wi grant access to its guarded or secure resources.

ContentProfileEntries

A data model is the preferred container for the ContentProfile, although the access registry xml file could contain the needed modeling without the use of a database. This model registers each secured resource using two-factors with each factor being composed of a key-value pair of attributes. A ContentProfileEntry(CPE) wraps these two factors to specify a user's access to this type of content. The CPE itself represents one permission and many CPI can be associated with one ContentProfile belonging to one identified user.

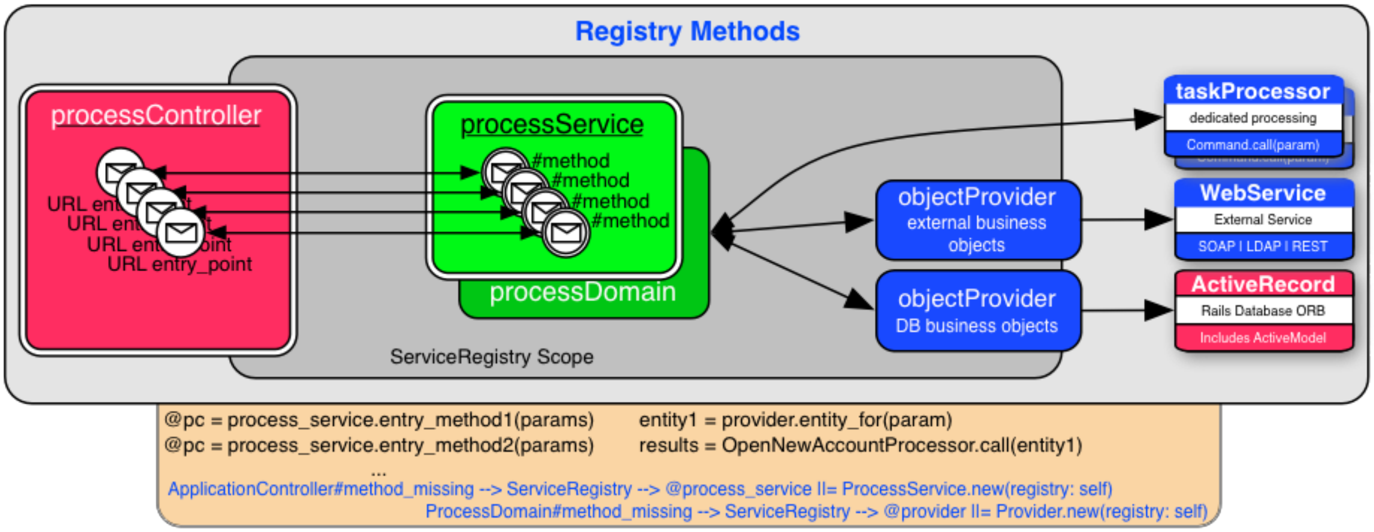
The First factor key is TopicType, it categorizes the type of entity domain being secured; like Account, Branch Office, Product, Operational-Info, or other simi entities. Each TopicType key uses its value attribute to specifically identify it; a k,v example being key=Account, value=#account-number.

The second factor's key is ContentType, categorizes the type of asset or resources being secured; like Contracts, Commission Statements, Licenses, or othe similar assets. Each ContentType uses its value attribute as a precise identifier or search term that can be used to select a certain set of items; presuming th items are stored a filesystem or document management system. The identifier is used to retrieve items matching.

Here is an example of how a CPE could be applied to a file system storage container.

- syntax: //storage/TopicType/TopicValue/ContentType/ContentValue
- populated: //storage/Account/160032/Contracts/*_purchase.*
- populated: //storage/Account/160032/Images/*.jpg
- populated: //storage/Account/160032/Commission/*_commisson.pdf
- populated: //storage/Agency/4014/Licenses/*_license.txt
- populated: //storage/Product/MDL-8433167/Specifications/{"docId":"810", "folder": "model_specs"}

Content-based Authorization goes deeper than the normal show/hide or access/no-access authorization outcomes. It first identifies an authentication user's access authorization to a specific business entity. Then specifies access, by type, to secured assets related to that business entity.



```
@pc = process_service.entry_method1(params)       entity1 = provider.entity_for(param)
@pc = process_service.entry_method2(params)       results = OpenNewAccountProcessor.call(entity1)
...
ApplicationController#method_missing --> ServiceRegistry --> @process_service ||= ProcessService.new(registry: self)
                    ProcessDomain#method_missing --> ServiceRegistry --> @provider ||= Provider.new(registry: self)
```

SknServices Strategy DDD Lite Strategy

Web Frameworks

Are not Application Servers

Rather than building the application the Rails (MVC) Way. We took a step back from MVC, and opportunistically used a strategy influenced by Domain Driver Design principals. Our intent was to separate as much of our application logic from the direct mechanics of Rails as possible; treating Rails more like a librar web adapter. Where/When it is required to make a Rails API call we wrap those calls in a semantically named method of our choosing; often inside our Provi class.

This approach does not intend to discount the usability or value of Rails as a Web Framework, it's excellent as a Web Framework; but it is not an Application Server similar to TomCat, or Jetty.

The resulting application will be easier to maintain, test, and it will be easier to upgrade Rails versions.

Service Strategy

If not MVC where does code go?

Following the Rails MVC Way, code that would be in the View and Controller is placed in a Services object grouping that supports **controller entry-point methods**. We are presuming a Services object will support more than one controller entry method, following the notion of a business process being implemented. Because of this **more than one** decision items like the ServiceRegistry and ProcessDomain classes exists to simplify the initialization of the m method service object themselves and their access to providers and processors.

A Services object is normally contextual, attempting to follow the flow of a business process. It's primary role is to map or transform request/response data f the Controller. The data it transforms typically comes from a Domain Class, which it inherits. All services object return one bundle of results in a DottedHash object called `SknHash or NestedResult`. Views and Controllers use the contents of their nested result to complete their operations; typically named `@page_controls`, this reduces the total number of instance variables being defined in controllers - again making testing easier.

Domain Classes contain the code that would otherwise be embedded in a Model, using the Rails MVC understanding. However, Domain Classes concentrat their methods on the broader business process; attempting to collect related methods that handle all operations for the process, and the logic needed to ens each process step is successful. Methods contained in the Domain class also facilitate cross-domain sharing or reuse of process methods; since another service/domains could require a function contained in a different domain, they could call straight through that process_service to use a domain method.

Domain classes call on or instantiate Provider and Processor objects, and other helper objects to as needed to further condense business process tasks; wh reduces the code size of the domains and their methods.

A ServiceRegistry is used to collect all services, providers, and optionally processors defined in the application, it is the sole provider of instances of these services. ServiceRegistry expects both controllers and other services to be making requests for a particular service; which is memotizes to speed up second access.

Methods in a domain or service class should minimize their user of controller based methods and helpers to maintain separation and improve test outcomes However, access to the controller is available from every service and domain class, through the inheritance chain: Domains inherit from DomainsBase class. DomainsBase keep a initialization reference to the ServiceRegistry that instantiated it `#registry`, and the ServiceRegistry keeps a reference to the controll that instantiated it `#registry`. Thus from any service or domain `#registry.controller_name` is an in-direct or direct reference to the Controller. By implementing a ServiceRegistry and associated #method_missing methods, we avoid the burden of populating a #delegate or #forwardable method requirer

Helpers are used to wrap controller methods that are required from Services or Domains. Wrapping controller methods with a helper facilitates test mocking localizes the Rails syntax requirements to the Rails helper.

ServiceRegistry **Business Logic Propagation**

Rails ApplicationController is augmented with a `#login_required` method to support the Authentication/Authorization processes. In support of business lo three more methods/components are added to the base ApplicationController for use by all controllers that inherit from it.

- `:ApplicationHelper` module contains controller helpers designed to be used by Services to gain access to standard controller capabilities and oth Services; these methods can easily be mocked, allowing the Services to be tested without a Controller being active in many cases.
- `#service_registry` wraps all Services and Provider classes and standardizes their initialization params.
- `#method_missing` method routes Services method calls to the service_registry, relieving us from defining every Service as a Controller Helper or populating the syntax of #delegate and/or #Forwardable.

You can think of the ServiceRegistry as a classic Rails ControllerHelper. However, all our business logic and data generation logic is defined in Service Class memorized by the ServiceRegistry. For testing this allows the ServiceRegistry replace the controller, in many cases.

Object Storage Service **Session like object storage**

The ServiceRegistry provides a in-memory object storage service for use by Services and Provider classes where expensive objects can be stored across multiple request/response cycles if needed.

Domains **Domain Classes**

Domain classes contain methods related to the business process it is responsible for. PasswordResetDomain would have methods structured to handle eacl step in the process of resetting a user password, and all the related IO routines involved. This a base class for its related Services object, and should not be directly instantiated.

A Domain's class single responsibility is to have methods to complete the whole process of a business sub-domain. It has no concern for how its resolved information will be presented or displayed.

Domain classes do make IO calls to ActiveRecord Models, and by this strategies convention the only class to make these AR calls. However, the AR Models have no business logic and minimal validations and callbacks implemented.

If the work of a particular step in the business process gets deeply involved (more than x lines), a taskProcessor class should be created the handle that step the Domain class

Services **Service Objects**

Service objects inherit a related Domain Class and have the single responsibility to transpose the Domain's information to a data value package useable by t controller or view. It may invoke some controller methods to format certain values, like a named URL; however such calls are supported by Controller/View Helpers which can be mocked for testing.

Service objects are directly invoked by a Controller route-based method, like: HomeController#index. The implementation of the :index method looks similar `@page_controls = home_service.homes_list()`

The return value from the above call is a DottedHash object. All Services should return a value container of this type containing all the data or information elements relevant to this methods response. This value container can be converted to JSON or a Hash bundle as needed by the View handler or API Respor The only required elements for this type of value container is :success, and :message, so each service methods includes these elements allowing the control method coding to flash a message and/or redirect as needed.

All the statements you would normally see in a MVC controller method are actually contained in the Service object's structure.

Service objects should perform any authorization checks needed before packaging domain data into a response. This prevents the View handler from being required to do it, and provides better protection of the applications data.

DottedHash SknUtils::NestedResult class, **Data exchange between Service and Rails MVC classes**

SknUtils Gem is utilized to provide a flexible container to hold all data returned by a Service. Dot notation or hash notation is supported by this object, again making it easy to capture test data.

Helpers **Controller & View Helpers**

`ApplicationHelper` module and the `ControllerAccessControl` module contain security helpers and helpers designed for views to operate on the dat only results returned by all Services, and to authorize on-page elements.

The design guidance for html views, and other view handlers, is to not do access authorization in the view. Instead allow the service objects to do the authorization and return either an empty value or a positive value.

Testing **Directly test the code you write**

To implement an application using this strategy you will create Domains, Providers, Processors, Services, and Helpers Classes; along with the necessary Vie

Using the ServiceRegistry as a stand in for a Controller, many of the Services and Domains can be tested without engaging the full Rails stack, which might r tests run faster.

Eighty plus (80+%) percent test coverage can be achieved by testing each Services method alone. Testing the Service#method that's invoked in that control method with a range of valid and invalid params will exercise the majority of paths through your business logic.

Rails Upgrades **Did we wrap all the Rails touch points?**

- Module: ApplicationHelper, contains all the Rails methods call by a Service, Domain, or Provider class.
- Provider class, contains all the ActiveRecord methods called. They were concentrated in a providers to remove sprawl!
- Services and Domain classes, remove all business logic from Views and the Controller proper.

The result will be upgrades which can be easily **evaluated and applied with minimal effort!**

Architectural Information (/pages/details_architecture)
System Information (/pages/details_sysinfo)

SknServices Release Version: 1.3.1.2

Top
Home (/pages/home