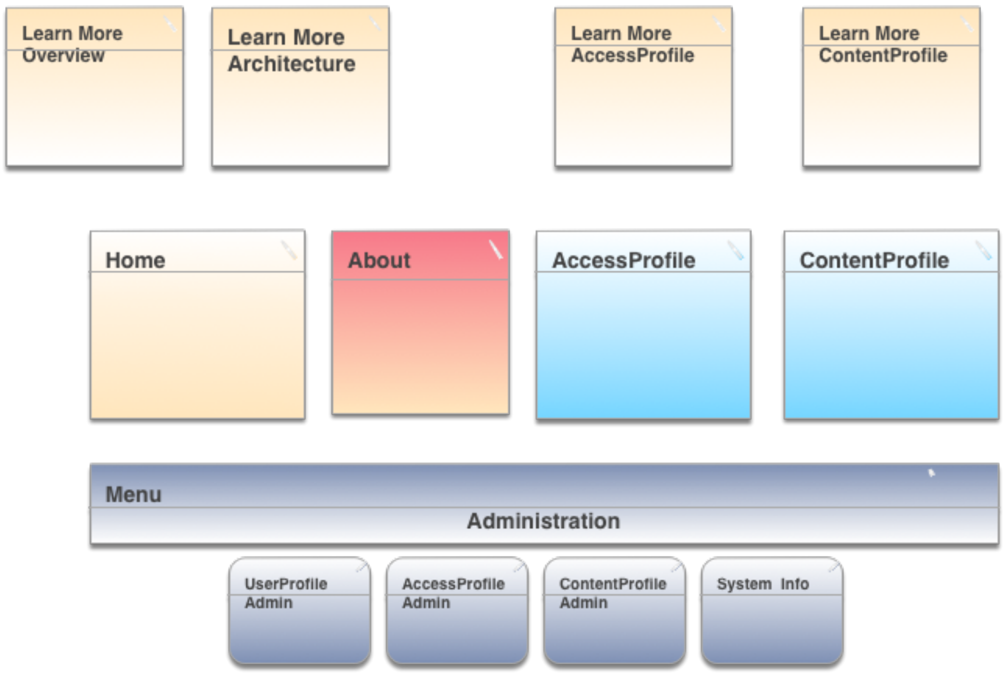
About SknServices



The SknServices project exists to explore Authentication and Authorization services. With special focus on content-based authorization using expanded Permissions properties. It also implements a **Services** implementation strategy that lessens it's dependancy on Rails.

SknServices Strategies

Authentication — **UserProfile**

The Warden Gem is used for its highly secure and flexible rack-based security engine.

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 Strategies capable of user/password, http basic auth, single-sign-on tokens, and several more.

Remember me, secure email based password-reset, and session timeout is also enabled.

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.

Authentication is evaluated with every page request, except for images, javascript, and css script files.

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

Authorization: Access — **AccessProfile**

AccessRegistry.xml, registers and/or assigns a URI to every page, action, and information element. All secured page elements are assigned an unique id, called URI, 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 be 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 easily represented by combining the controller name and controller method name; or URL path. An example URI for the index page in the HomeController would be `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, dynamically assigned to 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*. Page Access authorization is automated by examining the the controller name and method to compose the URI for the page. Clickable (menu items, buttons, etc) authorization requires the developer to use programming helpers to implement authorization. Showable Data Elements have two levels of authorization available; Clickable, and content-based authorization using expanded permissions, both use the AccessRegistry.

Authorization: Content

ContentProfile

A ContentProfile contains a the collection of permission entries for a user identified via their PAK id. Each user is optionally assigned a content profile, which will grant access to its associated 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 CPEs can be associated with one ContentProfile belonging to one identified user.

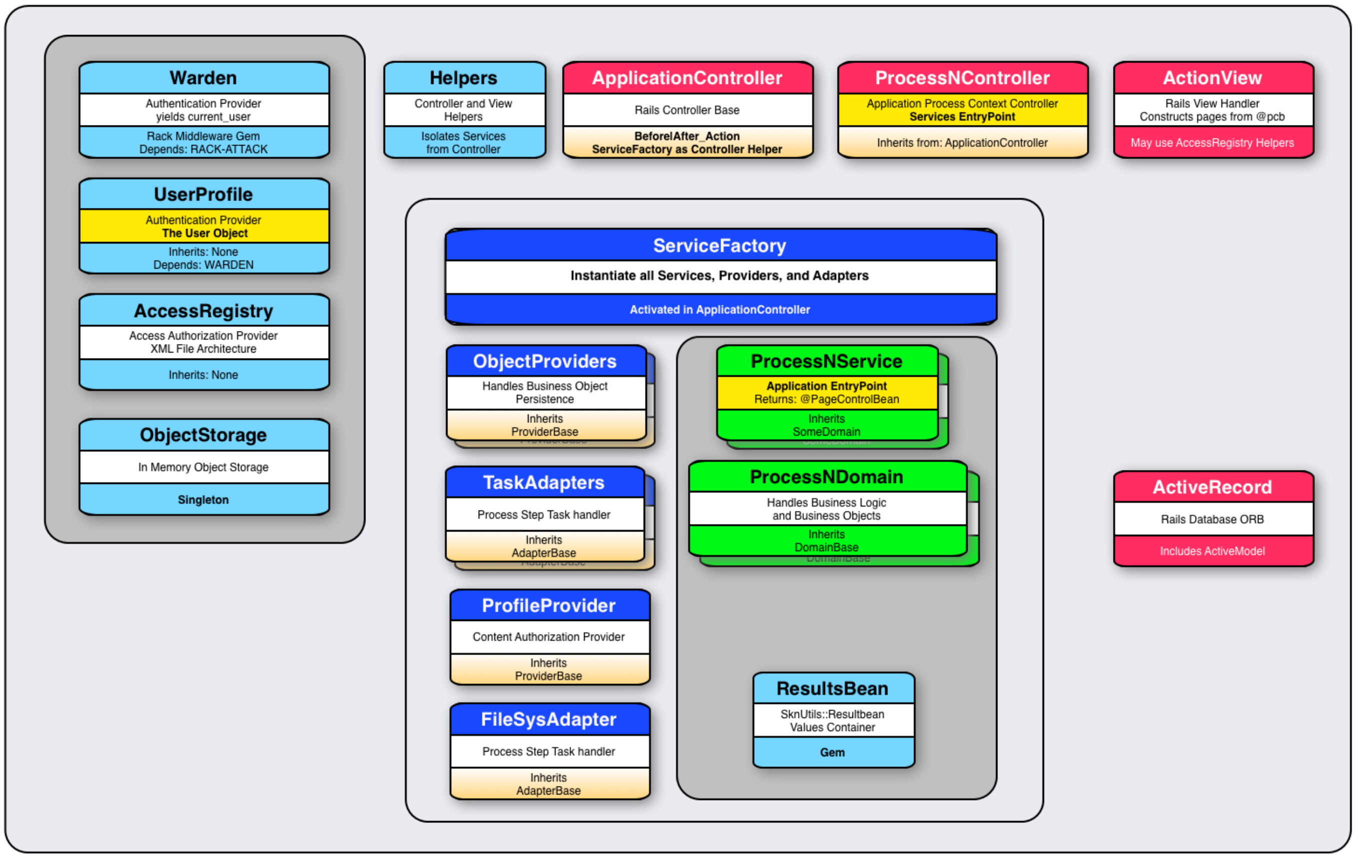
The First factor key is TopicType, it categorizes the type of entity domain being secured; like Account, Branch Office, Product, Operational-Info, or other similar entities. Each TopicType key uses its value attribute to specifically identify it; a k,v example being key=Account, value=#account number.

The second factor's key is ContentType, categorizes the type of asset or resources being secured; like Contracts, Commission Statements, Licenses, or other 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 these 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.


Warden
Authentication Provider yields current_user
Rack Middleware Gem Depends: RACK-ATTACK
UserProfile
Authentication Provider The User Object
Inherits: None Depends: WARDEN
AccessRegistry
Access Authorization Provider XML File Architecture
Inherits: None
ObjectStorage
In Memory Object Storage
Singleton
Helpers
Controller and View Helpers
Isolates Services from Controller
ApplicationController
Rails Controller Base
Before/After_Action ServiceFactory as Controller Helper
ProcessNController
Application Process Context Controller Services EntryPoint
Inherits from: ApplicationController
ActionView
Rails View Handler Constructs pages from @pcb
May use AccessRegistry Helpers
ServiceFactory
Instantiate all Services, Providers, and Adapters
Activated in ApplicationController
ObjectProviders
Handles Business Object Persistence
Inherits ProviderBase
TaskAdapters
Process Step Task handler
Inherits AdapterBase
ProfileProvider
Content Authorization Provider
Inherits ProviderBase
FileSysAdapter
Process Step Task handler
Inherits AdapterBase
ProcessNService
Application EntryPoint Returns: @PageControlBean
Inherits SomeDomain
ProcessNDomain
Handles Business Logic and Business Objects
Inherits DomainBase
ResultsBean
SknUtils::Resultbean Values Container
Gem
ActiveRecord
Rails Database ORB
Includes ActiveModel

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 use a strategy influenced by Domain Driven 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 library or web adapter.

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.

We believe the resulting application will be easier to maintain, won't require us to test controllers and code we didn't write, 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 that support one or more controllers.

A Services object is normally contextual, attempting to follow the flow of a business process. It's primary role it to map or transform request/response data to the Controller. The data is trasnforms typically comes from a Domain Class, which it inherits. All services object return one bundle of results in a helper object call `ResultsBean` . Views and Controllers us the contents of thei result bean 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 concentrate their methods on the broader business process; attempting to collect related methods that handle all IO for the process, and the logic needed to ensure each process step is successful.

Domain classes call on or instantiate Adapter objects, and other helper objects to as needed to further condense business process tasks.

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

Methods in a domain or service class should minimize their user of controller based methods and helpers where ever possible to maintain separation and improve test outcomes. However, access to the controller is implemented and available from every service and domain class, through the inheritance chain: Domains inherit from DomainsBase class. DomainsBase keep a initialization reference to the ServiceFactory that instantiated it `#service` , and the ServiceFactory keeps a reference to the controller that instantiated it `#factory` . Thus from any service or domain `#service.factory` is a direct reference to the Controller.

ServiceFactory **Business Logic Propagation**

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

- `#service_factory` module contains controller helpers designed to be used by Services to gain access to standard controller capabilities and other Services; these methods can easily be mocked, allowing the Services to be tested without a Controller being active in many cases. It wraps all Service/Domain objects and standardizes their initialization params.
- `#method_missing` method routes Services method calls to the service_factory, relieving us from defining every Service as a Controller Helper.

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

Object Storage Services

The ServiceFactory provides controller services to the Services Object and it also provides and in-memory storage facility where objects can be stored across multiple request/response cycles if needed.

Domains **Domain Objects**

Domain classes contain methods related to the business process it is responsible for. PasswordResetDomain would have methods structured to handle each 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 service. 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 50 lines), a ProcessStep class should be created the handle that step for the Domain class

Services **Service Objects**

Service objects inherit a related Domain Class and have the single respobsibility to transpose the Domains information to a data bean useable by the 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 object are directly invoked by a Controller route-based method, like: HomeController#index. The implementation of the :index method looks similar to: `page_controls = service_factory.home_service.homes_list()`

The return value from the above call is a ResultsBean object. All Services return a bean of this type containing all the data or information elements relevant to this methods response. This bean package can be converted to JSON or a Hash bundle as needed by the View handler or API Response. The only required elements for this type of bean is :success, and :message, so each service methods includes these elements allowing the controller 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 method.

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

ResultsBean **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.

Controller & View Helpers

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

Testing **Directly test the code you write**

To implement an application using these strategies you will create Domains, ProcessStep, Services, and Helpers Classes; along with the necessary Views.

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

Rather than test a Rails Controller to verify business logic, test the Service#method that's invoked in that controllers method. Only during LIVE Integration or Feature tests should concerned about what the Rails Controller is doing.