Oracle® Agile Product Lifecycle Management for Process

Custom Portal Implementation Guide

Extensibility Pack 3.0

E38005-01

March 2013

ORACLE®

Copyrights and Trademarks

Agile Product Lifecycle Management for Process

Contents

Introduction

Custom Portal is an extensible web portal framework for customers to build web pages that query for Agile PLM for Process objects, display the search results, and print the result details in various formats.

This document details the Custom Portal framework, including the administration of portal pages and views, the technical implementation requirements for extending the portal, and the existing reference implementation.

Overview

The Custom Portal is an extension of the Agile PLM for Process (PLM4P) application suite. It allows customers to implement various integration solutions that leverage the PLM4P data and capabilities without using the core application. Its primary usage is to provide a framework for searching, filtering, and displaying PLM4P data, and gives solution implementers the ability to customize each of those aspects.

Custom Portal pages can be built to give users (who would not typically access PLM4P) very specific access to certain data. Views of that data can be tailored to meet specific business needs, such as providing business partners with custom views into their specifications.

Possible Uses:

1. **Grant read only access to your individual plants.** Plant users are a very different audience compared to the average GSM specification user. Plant users need to see a read only view of the entire finished good specification. This could be a combined view of data spanning attributes from the trade, nutrient profile, formulation and raw materials.
2. **Grant read only access to internal departments in a format they are used to seeing the data.** For example, you can grant the Marketing department access to Product Fact Sheet reports for only approved finished goods. This would allow them to see nutritional fact panels and label claims pertaining to a particular finished good without granting them access to the entire Nutrient profile and Trade specifications.

Component Overview

Custom Portal consists of several core PLM4P components, Custom Portal framework components, and custom code.

Data Administration Screens

Data Admin screens in the core PLM4P application allow for the configuration of the Custom Portal. Users configure a Portal with a name and status, and then configure the individual Views available in each Portal.

Each View is a separate web page that provides search, filtering, and display capabilities. Users configure a View with a name, status, and whether the individual search results should appear in a new pop-up

window or in the same window. Additionally, the search, filtering, and display capabilities of the View are declared in the configuration by specifying custom Plugins and Filter Controls.

Details of using the Data Admin screens are specified in the Configuring Portal Pages and Views section. Technical details of the plugins used by the Views are specified in the Technical Overview and Technical Implementation sections.

Portal Management Screens

Portal Management screens, accessible in Custom Portal, enable Profiles to be created for each Portal. A Profile is a way to associate a View to various participants (users) and apply filters to the search results that the View provides.

> For instance, a Profile can be configured to allow users of a certain Facility to search using a specific View, but only see A*pproved* Specifications as search results, while another Profile could be granted to one specific user for the same View that would not restrict the search results.

When a user logs into a Portal page, the left navigation is populated by whichever Profiles in that Portal the user has access to.

Details of configuring Profiles using the Custom Portal Admin screens are specified in the Configuring Portal Pages and Views section.

Custom Portal Web Application

CustomPortal is the web application that acts as the host for the portal pages, portal management screens, and user authentication. User interface customization can be made in the Custom Portal web application to personalize the branding, header, navigation, and footer sections of the portal pages.

The Custom Portal web application is provided and maintained as an extension by Oracle. Changes made to this web application should be minimized (primarily stylesheet and UI changes) to avoid maintenance issues. Clients adding their custom portal pages may do so in their own web application that is then set up as an IIS virtual directory within Custom Portal.

Custom Plugins and Filter Controls

To create custom portal views which implement the search, filtering, and display behaviors, clients must create individual Search Plugins, Rendering Plugins, and Filters Plugin controls. These may be located in the client's custom web application or assembly under the Custom Portal web application. These Plugins and controls are then configured for each View in the Data Admin screens. The Profile Management screens can then be configured to use the filters to limit data results when setting up the various Profiles, as discussed above.

Details of creating Search, Rendering, and Filter Plugins are discussed the Technical Implementation section below.

Portal Interfaces

Each custom Search, Filter, and Rendering Plugin must implement specific interfaces which are provided in the CustomPortalInterfaces assembly.

Technical detail of each interface is discussed in the Class Structure and Interfaces section below.

Technical Overview

Custom Portal is a web application that must be installed in an existing Agile PLM for Process environment. It contains the portal management screens, page layout, security, and the pluggable framework that is used to develop custom search, filter, and display functionality. It relies on the Interfaces located in the CustomPortalInterfaces assembly, which define the class structure required when using the Search, Render, and Filter Plugins.

Client implementations that use the Agile PLM for Process Web Services API will require that the Web Services API is installed in an accessible environment.

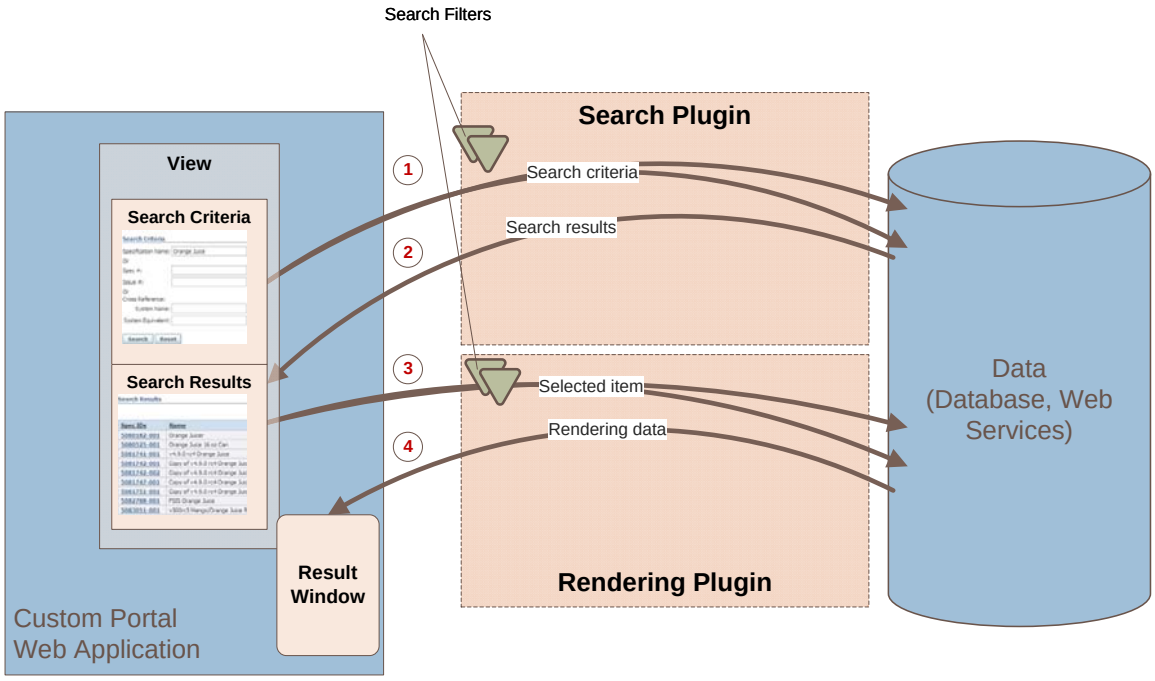
Custom Portal will also host the client's own web application or assembly in which most of the customized plugins, forms, and other implementation code should be located.

For more details, see the Installation and the Technical Implementation sections below.

Search Process Overview

The Custom Portal hosts a View, which is a web page that contains the search criteria and search results. Once a search is performed and search results are displayed, clicking on a result entry will display the final detailed results in the chosen output format.

The following diagram illustrates the processes of a custom portal View:



The numbered steps in the diagram above describe the process as follows:

1. The search criteria are entered and the Search button is clicked. This calls the configured Search Plugin and passes it the Search Filters that restricts the result set. The Search Plugin queries the database or calls web services for the needed data.
2. The search results are returned and displayed in the search results section. These results include information about the columns that should be displayed as part of the result set.
3. A result entry is selected by clicking on it. This calls the configured Rendering Plugin, which will likely make its own database and/or web service calls to retrieve additional data needed for the final display. It can also use the filters to restrict any data returned as needed.
4. The Rendering Plugin will then process the results and display them in the desired output format, such as HTML or PDF, either in the same window or by launching a new window.

Solution Implementation Overview

Implementing a Custom Portal solution will require different strategies depending on the business requirements and the available integration options.

Solutions will be based on approaches to the following:

How to ***access*** the data used for searches and rendering
How to allow for ***search*** capabilities
How to ***filter*** data to restrict access
How to ***render*** the data results

Accessing Data

Implementing the search and rendering capabilities will require a way to access the Agile PLM4P data. This may be achieved use existing web services, direct database queries, and/or PLM4P internal data objects and services.

Web Services API

A common approach to implementing some of the search behavior and retrieving specific data for rendering is to use the Agile PLM for Process Web Services API. The Web Services API provides over 40 web services that can be used for various data retrieval needs, such as retrieving nutrient profile information, custom section data, and related specifications information, as well as for performing searches for GSM specifications or eQuestionnaire items based on specific criteria.

Typically, a Search or Rendering Plugin will call one or more web services to achieve its required outcome. Each web service is granular in nature, achieving a very specific business need. Therefore it will be common to compose several web service calls together, using the results of one web service to call other web services as needed.

> For instance, you can call the GetSpecNumbersForCriteria web service to enable a GSM Spec search for Ingredient Specifications created in the past 30 days that are in an Approved status. The result specifications can then be used to call the GetNutrientItemsPer100g web service to get the nutrient items list of each spec, and use those results to find if any of those contain a specific amount of given Nutrient Item.

Advantages of using the Web Services API include leveraging user authentication, user authorization, and the existing business logic in the system. Additionally, the Web Service API is supported and maintained by Oracle.

Details of the Web Services API can be found in the Extensibility Pack's Web Services API User Guide, including a listing of all of the current web services and detailed documentation of each web service.

A further discussion of using web services in the Custom Portal can be found in the Technical Implementation section.

Database Queries

Required data that is not available via the current Web Services may be retrieved using custom database queries. Read access to the database and knowledge of the database structure will be required.

While using direct database queries to access required data will be possible, and at times, the only way to retrieve the desired data, it does have some drawbacks:

> Implementing user security/permissions checks in the database queries may be complex
> Database schemas may change between releases – requires maintenance

Details of the Agile PLM for Process database schemas can be found in the Extensibility Pack in the Docs\DbSchema directory.

Internal Data Objects

Some data may be accessed by using the internal data object and accessing its properties. Typically, the data object's primary identifier (the PKID) is required to load the object and access its properties. The PKID can be returned via database queries (the Web Services typically do not expose the PKID), and then used to load the data object using internal PLM4P Services.

Searching Data

Setting up a search utility that specific users can utilize will require the use of a Search Plugin. The Search Plugin will provide the search ability (search fields, predefined search inputs) and determine the search result format, such as the result columns. It must also implement the actual search, either by calling one or more of the existing web services, or by using direct database queries, or both. The search filters configured for the View are used by the Search Plugin to filter the result set (see Filtering Data). A list of error messages may be populated in the Search Plugin and then displayed to the user.

Filtering Data

Search data returned will likely need to be filtered based on the Portal Profile or the users accessing the results. Code is used to limit the results or the queries so that only the right data is exposed.

Filtering search result data is done using two filter controls, which are specified in the Data Administration screens when configuring a View:

1. An Admin Filter control is used in the Profile configuration to automatically filter results for that view.
2. An "in-search" filter control is used during the search and allows the user to specify the filtering during the search process.

Rendering Data

Rendering data involves displaying the desired output results once the search results are displayed and a result item is selected. The Rendering Plugin configured in the Data Admin screen for a View is called to display the result, either in the same window or in a pop-up.

Rendering Plugins can display the data in whichever format is created, such as HTML, PDF, etc. Additionally, the Rendering plugin can also access additional data need for the display of the item selected using the various data access methods specified above.

> For instance, if the search results display a list of Ingredient Specifications that contain a specific Nutrient Item, clicking on one of the specifications returned could display more detail about that ingredient spec, such as the category, subcategory, and group, the ingredient statement, and the full list of nutrient items. The Rendering plugin would call specific web services to retrieve this desired data, and then could display it as a PDF document.

Configuring Portal Pages and Views

Data Administration Screens

Data Admin screens in the core PLM4P application allow for the configuration of the Custom Portal. Users configure a Portal with a name and status, and then configure the individual Views available in each Portal.

Only users with the role "[CP_SYSTEM_ADMIN]" may access the Portal and View screens.

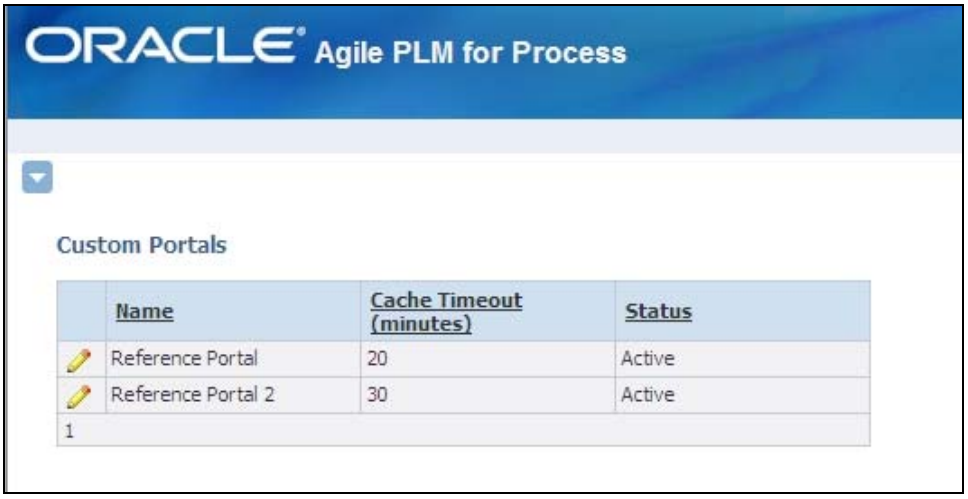
Changes to the Portals or Views are available after a Cache Flush event or IIS reset.

Custom Portals Screen

The Custom Portals screen defines a portal used by the Custom Portal Web application. Each portal is implemented as a separate page (.aspx) in the Custom Portal.

Each entry has the following fields:

Name: defines the portal name
Cache Timeout: the number of minutes that the search results are cached
Status: status of the portal (Active, Inactive, or Archived)



Custom Portal Views Screen

Each View is a separate web page that provides search, filtering, and display capabilities. Users configure a View with a name, status, and whether the individual search results should appear in a new pop-up window or in the same window. Additionally, the search, filtering, and display capabilities of the View are declared in the configuration by specifying custom Plugins and Filter Controls.

Each entry has the following fields:

Name: defines the view name, which is displayed as a page header and left navigation item.

Search Plugin: the class URI defining the custom search business logic, as implemented by a custom class implementing an ISearchPlugin interface.

- Ex:
  `Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSearchPlugin,MockCustomPortalPlugins`

Rendering Plugin: the class URI defining the custom rendering/display logic, as implemented by a custom class implementing an IRenderingPlugin interface.

- Ex:
  `Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSummaryRenderingPluginFactory,MockCustomPortalPlugins`

Filter Control: there are two filters controls specified here, separated by a semicolon. Each entry should be the relative path to the filter control (ex: `/DefaultSearchPlugin/DefaultSearchFilter.ascx`)

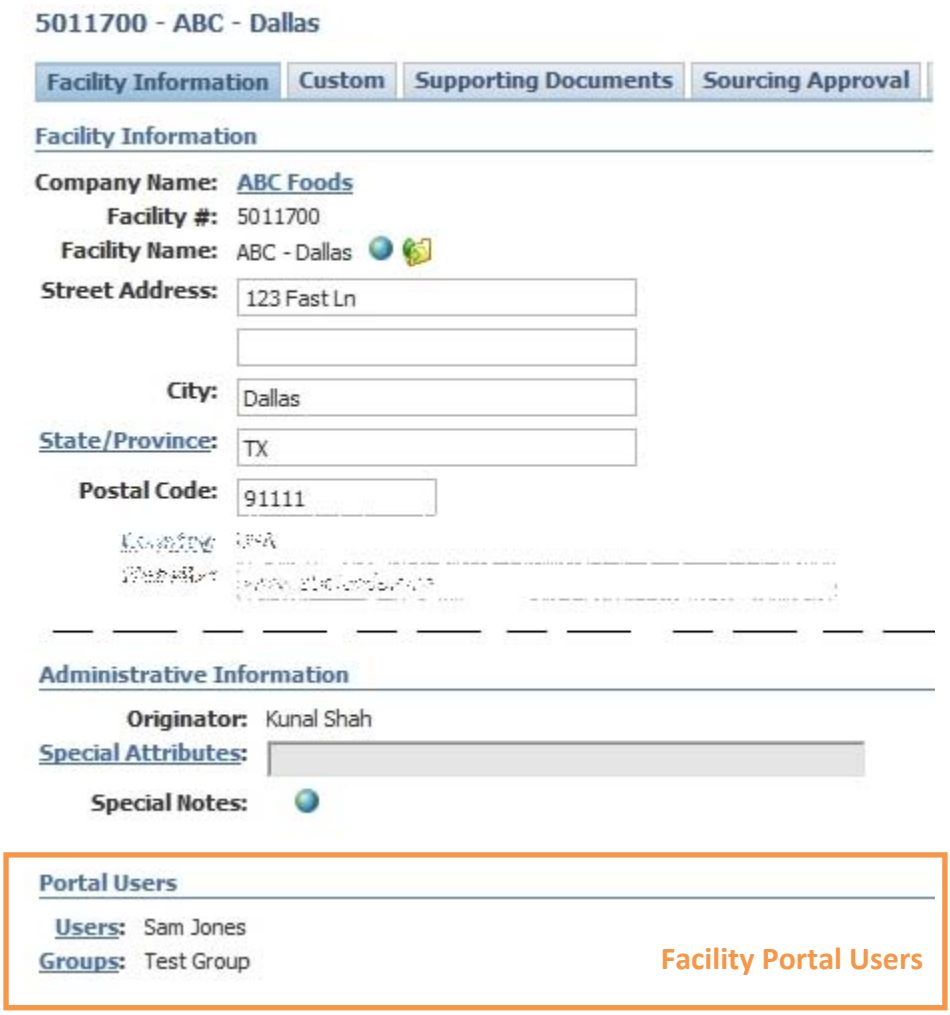
- The first is used in the search control on the default search page
- The second is used in the Custom Portal Management screen to limit results hidden from end users.

Open New Window: checkbox value determines if the search result entry, when clicked, is displayed in a new pop-up window or not.

Status: status of the view (Active, Inactive, or Archived)

SCRM Facility Screen

UGM users and groups can be associated to SCRM Facility profiles for use in the Custom Portal. If that Facility is selected in the Profile Participants section, access to the View is granted to these users. A new section, title Portal Users, is now available in the SCRM Facility screen.



To view the Portal Users section on an SCRM facility profile, make sure the following configuration entry is set to true: `CustomPortal.SCRMFacility.Participants.Enabled`

To *edit* the Portal Users section on an SCRM facility profile, the user must have the "[SCRM_PRINCIPAL_EDITOR]" role.

Portal Management Screens

Portal Management screens, accessible in Custom Portal, enable Profiles to be created for each Portal. A Profile is a way to associate a View to various participants (users) and apply filters to the search results that the View provides.

Only users with the role "[CP_ACCESS_ADMIN]" may access the management screens.

Searching for a Profile

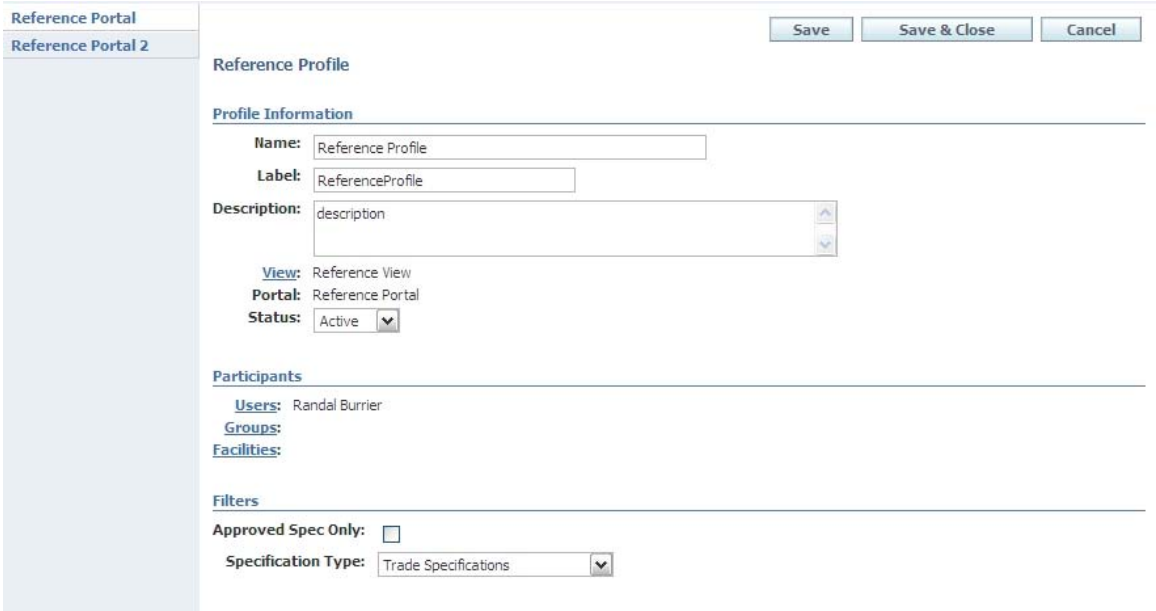
Access the portal management screens via http://<servername>/customPortal/Admin/PortalAdmin.aspx and select the Portal to configure profiles for from the left navigation menu.

This will display a Profile search screen to search for Profiles for the selected Portal, or create a new Profile.

Creating/Editing a Profile

Note that to configure a Profile, the relevant View must exist along with the actual Plugins and Filters. In other words, a Profile cannot be configured until the Plugins are set up in Data Admin.

This page allows users to configure a Profile for a View, the display name, the users/participants, and the admin filters.



The Profile Information section contains the following fields:

Name: the name of the profile
Label: the name to display on the portal page's left navigation bar and search title
Description: optional description
View: a pop-up selection of all Views
Portal: a read-only display of the current Portal being configured
Status: status of the profile (Active, Inactive, Archived)

The Participants section is where permissions are set for the Profile. It contains the following fields:

Users: selection of individual users
Groups: selection of user groups
Facilities: selection of SCRM facilities. UGM users can be associated to SCRM facility profiles. When you select a facility here it will use the UGM user collection stored on that facility. See the SCRM Facility Screen section for details.

The Filters section will differ by the selected View, as it will load the admin filter control specified in the Data Admin configuration screen for that View. The filters section shown above is based on a reference implementation: it allows the search filter to be restricted to Approved specifications only and to a particular specification type.

Search Screen

When users access any portal page, they are prompted to log in using the regular PLM4P login screen.

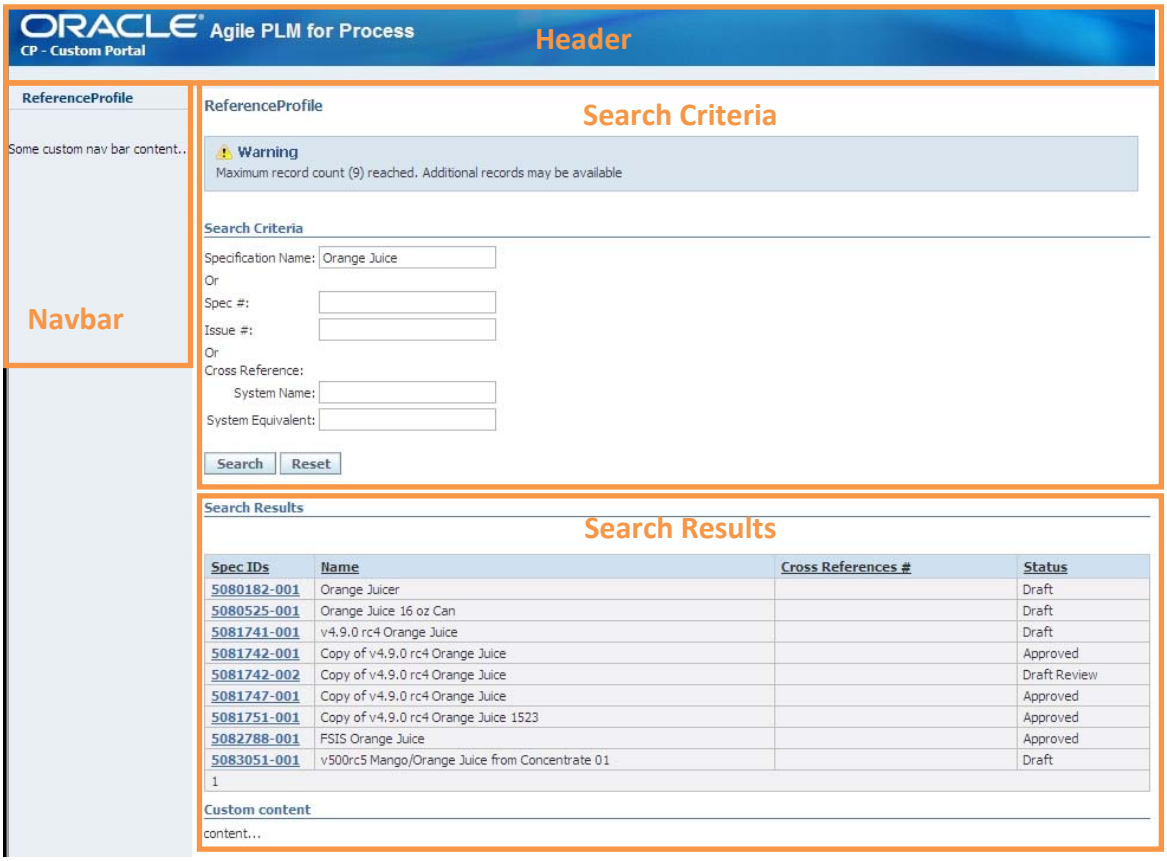
The location (URL) of the particular search screen will depend on the deployment implementation, namely where the Portal web control is being implemented.

More details can be found in the Technical Implementation section.

A web control (CustomerPortal.ascx), provided in the Custom Portal web application, must be defined for each Portal that is configured. The control defines the appearance and functionality of the Portal.

Each portal page has four sections:

1. **Header**: contains branding information
2. **Navbar**: lists any Views available to the user for the given Portal
3. **Search Criteria**: displays the custom search behavior (as well as the View name and any warning/error messages)
4. **Search Results**: displays the custom search results (as well as a page footer placeholder). The search results are cached for a set time, as configured in the Portal Data Admin screen.



Installation

Custom Portal Web Application Setup

To install the Custom Portal web application, you must do the following:

1. From the ExtensibilityPack release, copy the CustomPortal web application into your Agile PLM for Process application's \web folder.
2. Create a new virtual directory in IIS:
    a. Set the name to customPortal (this can be changed to a different name if needed).
    b. Set the path to the customPortal directory in the web folder.
    c. Add new virtual directories underneath the new customPortal virtual directory. [Be sure to remove the ApplicationName value from the Virtual Directory tab in the Application Settings section.]
        i. name: css, path: <PLM4P directory>\Web\css
        ii. name: images, path: <PLM4P directory>\Web\images
        iii. name: WebCommon, path: <PLM4P directory>\Web\WebCommon
3. **If using the Web Services API**, please read the document "Agile Product Lifecycle Management for Process Application Programming Interface User Guide for Extensibility Pack 2.4.pdf" in the ExtPack 2.4 package on how to install the API and modify the configuration files.

4. Make sure your user has the [CP_SYSTEM_ADMIN] role to configure the Custom Portal in the data admin menu. See Configuring Portal Pages and Views for details.
5. Restart IIS

Typical Custom Implementation Setup

Your custom implementation code (either packaged as a web application or an assembly) will also need to be installed. Typically, you will do the following:

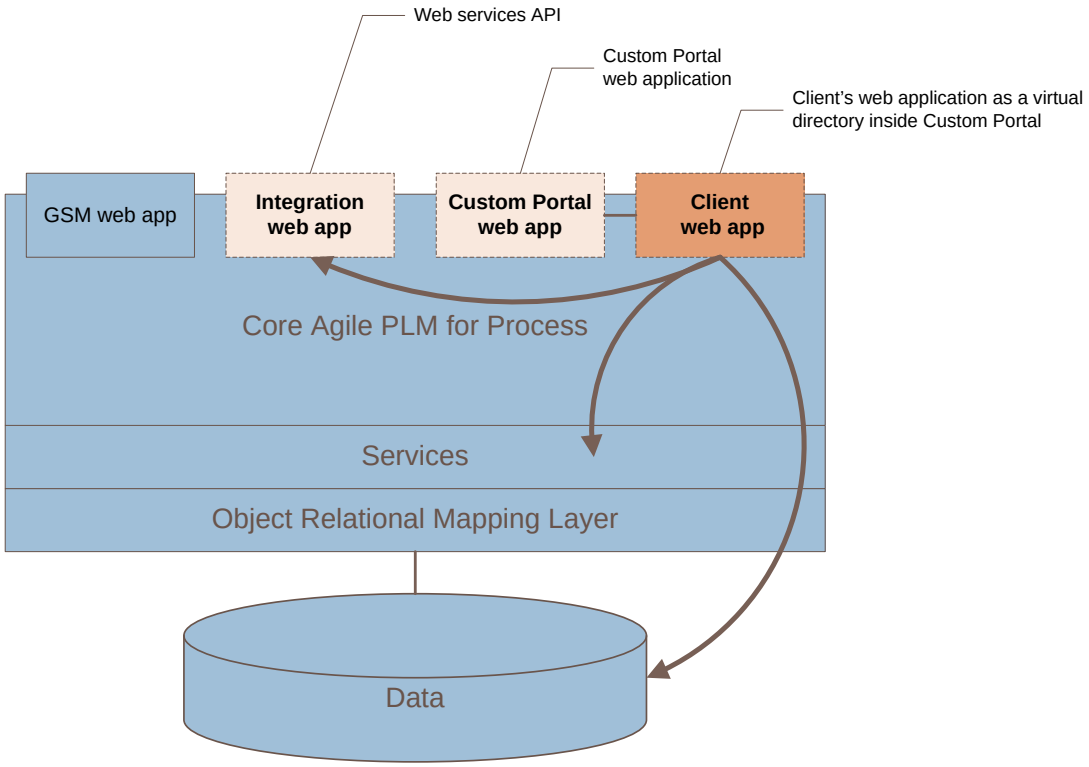
1. Place your custom web application or library in the <PLM4P directory>\Web directory.
2. Copy the .dll of your custom library or web application into the CustomPortal\bin directory.
3. Create a new virtual directory within the CustomPortal virtual directory that points to your custom web application.

More detailed installation procedures can be found in the Reference Implementation section.

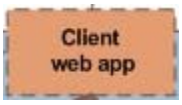
Technical Implementation

Architecture Overview

The following diagram presents a simplified model of the architectural components involved in implementing a Custom Portal solution.



The Custom Portal web application hosts the portal pages, portal management screens, branding, stylesheets, and user authentication.



The Client web application (or assembly), accessible via a virtual directory in IIS, implements the search, rendering, and filtering requirements by accessing the Web Services API, the database through direct queries, and possibly internal PLM4P services and data objects.

The Integration web application is used to host the Agile PLM for Process Web Services API.

Each of these modules must be installed and configured into an existing Agile PLM for Process environment.

Class Structure and Interfaces

This section describes the class structure for the Search and Rendering Plugins, and the Filter controls. The Plugins adhere to the interfaces defined in the CustomPortalInterfaces.dll, under the namespace `Xeno.Prodika.CustomPortalInterfaces`.

Filter Plugin Structure

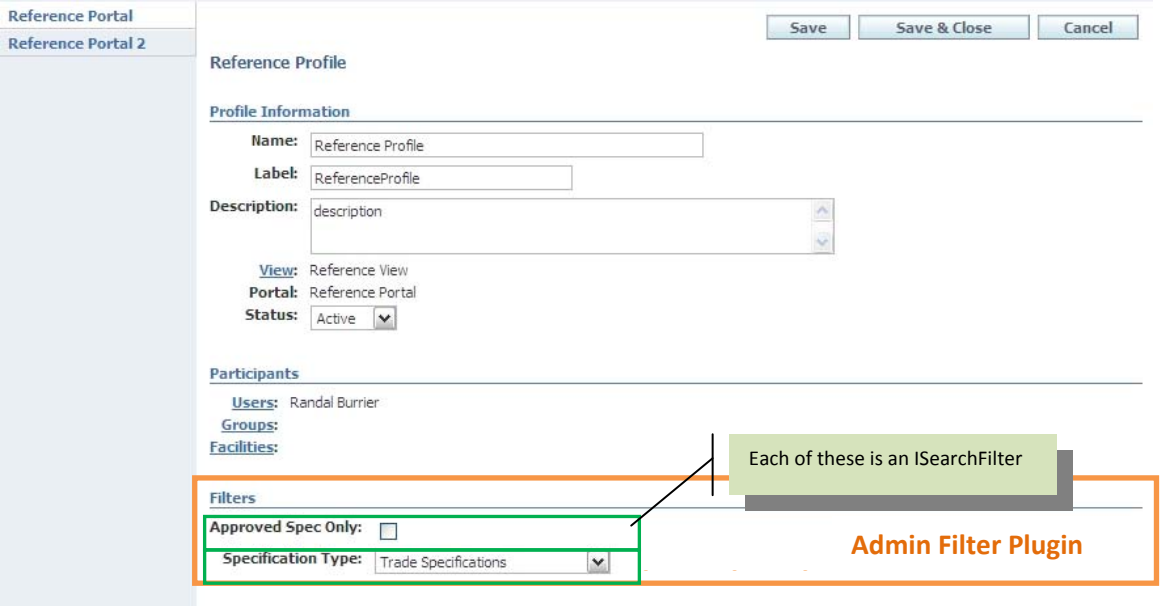
Filtering search result data is done using two filter plugins, which are specified in the Data Administration screens when configuring a View:

1. An `IAdminFilterPlugin` is used in the Profile configuration to automatically filter results for that view.
2. An `IFilterPlugin` is used during the search and allows the user to specify the filtering during the search process.

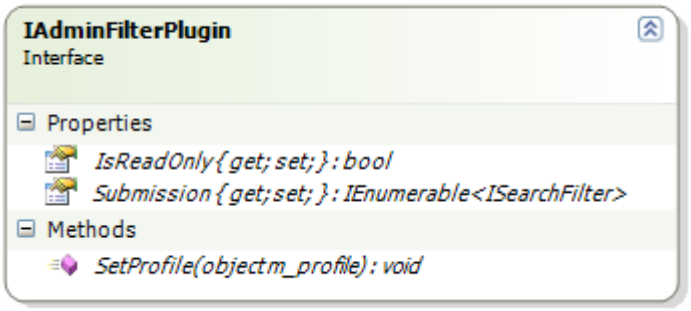
The search filters set up by these Plugins are then used by the Search Plugin (and possibly the Rendering Plugin) to limit the search and display results to the specified criteria.

IAdminFilterPlugin

The `IAdminFilterPlugin` is used in the Profile Management screen to set specific search filters for that profile. This allows multiple profiles to be set with different search filters for the same views.



The `IAdminFilterPlugin` is typically implemented as a user control, using data entry fields to set each `ISearchFilter`.



The `SetProfile` method is used to pass in the Custom Portal Profile object, which can then be used to save the Search Filter values to the Profile via the `Profiles.SetSearchFilter` method. The Search and/or Rendering Plugins can later retrieve these Admin Search filters directly from the Profile.

The `IsReadOnly` property is used to determine when to make the control editable.

The `Submission` property is used to get and set the search filters. The profile's current SearchFilters can be accessed by using the `Profiles.GetSearchFilter()` method and passing the profile that was passed in via the `SetProfile()` method.

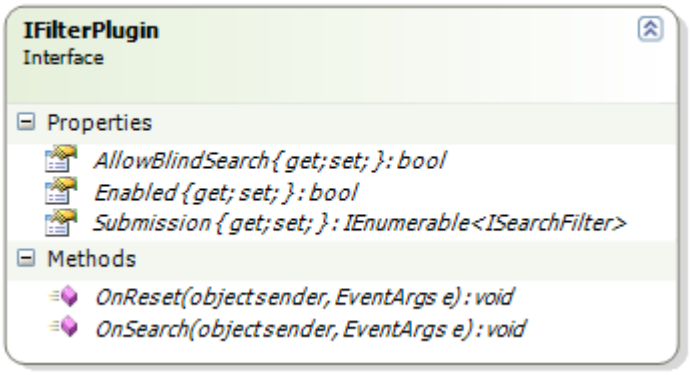
An example usage could be to set the allowable specification types for the view's search results. You could create a user control that lets the admin user select from multiple spec types, and create a SpecTypesFilter class that implements ISearchFilter. The Key would simply be some text description to refer to the filter, such as "SPEC_TYPES_FILTER", and the Operands would be set with the values of the spec type(s) selected. When the Search Plugin later executes the Search, it could evaluate each result to determine if the spec type returned matches the entries in the SpecTypesFilter. (Depending on the

search implementation, the search filters might be used as inputs to the search process, or used to evaluate the results and apply the filtering to the results, or both. The usage of the search filter may depend on the filter type itself.)

IFilterPlugin

Filter Plugins function much like Admin Filter Plugins. Filter Plugins are used in the Search criteria section, and contain a list of SearchFilters that allow the user to enter values for the search process.

The `IFilterPlugin` is typically implemented as a user control, using data entry fields to set each `ISearchFilter`. The Search Filters entered are passed in as parameters (along with the AdminSearchFilters) to the `SearchPlugin's Search`() method.



The `Submission` property is used to get and set the search filters. The search filters are retrieved from the form's data entry fields, and are then passed into the Search Plugin.

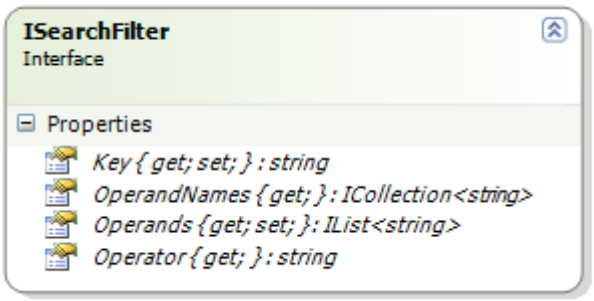
The `Enabled` property is used to determine when to make the control editable. Because the Filter Plugins are used in the search process, this is usually set to true.

The `AllowBlindSearch` can be used to indicate if the search should be allowed without any search criteria.

The `OnReset` and `OnSearch` methods are present to perform any needed event handling, such as clearing the search filters.

ISearchFilter

Search filters are passed to the Search operation of the Search Plugin. Each search filter contains a Key, which identifies the type of search criteria, and a list of Operands, which are the values to search by.



A simple example of using a SearchFilter would be to filter the results based on specification type. Setting the Key="SPECTYPE" and the Operands to a list of desired spec types (ex: ingredient, trade). The SearchPlugin would then use the filters and filter out specs that were not of the correct type.

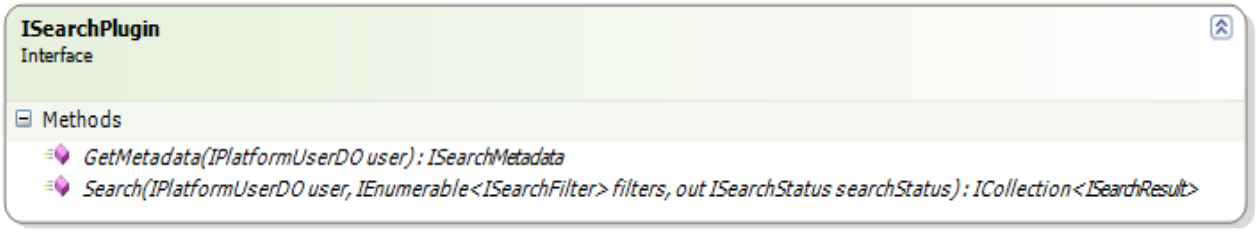
Note that the `OperandNames` and `Operator` properties are currently unused.

Search Plugin Structure

Portal pages display views using Search Plugins. Each View has an `ISearchPlugin` that specifies what search criteria to display, performs a search, and returns the result data.

ISearchPlugin

Criteria and result column metadata is returned by the `GetMetadata()` method. The `Search()` method handles the actual data.



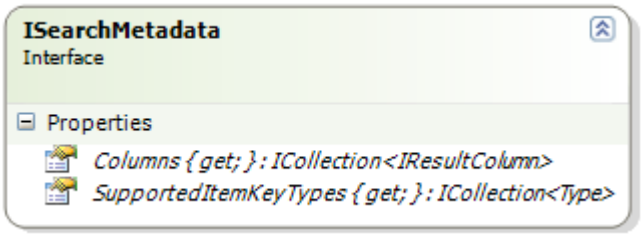
Portal pages query Search Plugins for the information to display. They need to know what type of search Result to offer the user. The pages also need to know what columns to add to the result grid.

The `GetMetadata()` method takes the current user as an input parameter and returns criteria and result column metadata as an `ISearchMetaData`.

The `Search()` method takes the current user as an input parameter, a list of `ISearchFilter` objects to filter the results, and passes an `ISearchStatus` object which will be populated with overall result of the operation and a list of error/warning messages. The result of the Search call is a collection of `ISearchResult` objects.

ISearchMetaData

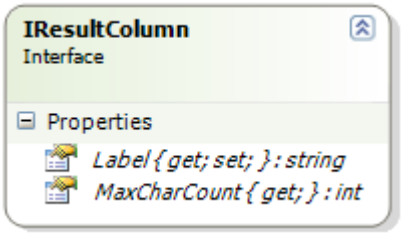
The `ISearchMetaData` interface provides access to the search result columns and the supported item key types.



The `Columns` property lists the result columns returned by the search as an `IResultColumn`.

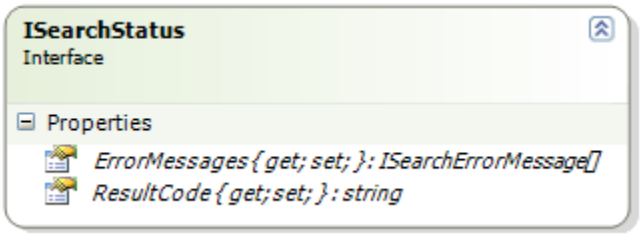
IResultColumn

Each column (`IResultColumn`) has a display label for the column header, and a `MaxCharCount` property that sets the max column width.



ISearchStatus

A SearchStatus object is passed to the SearchPlugin's Search method, and is populated with an overall result of the search, and a list of error messages. The result can be evaluated to determine how to handle the error messages. The error messages can be as display information to the user.

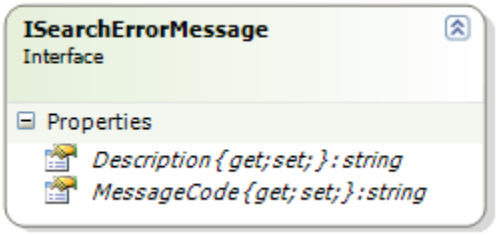


The Web Services API operations all return a result code and error messages indicating any issues encountered when calling the web services. For instance, if the input passed in to the web service is in an invalid format, an error message will be returned. This can be used in the UI to inform the user about of the error.

The error messages are returned as an array of `ISearchErrorMessage` objects.

ISearchErrorMessage

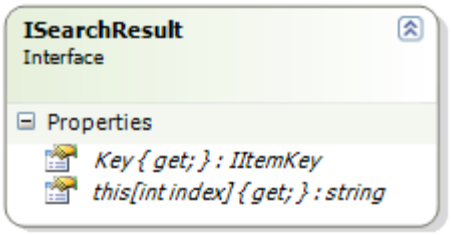
Each ISearchErrorMessage object contains a description and a message code.



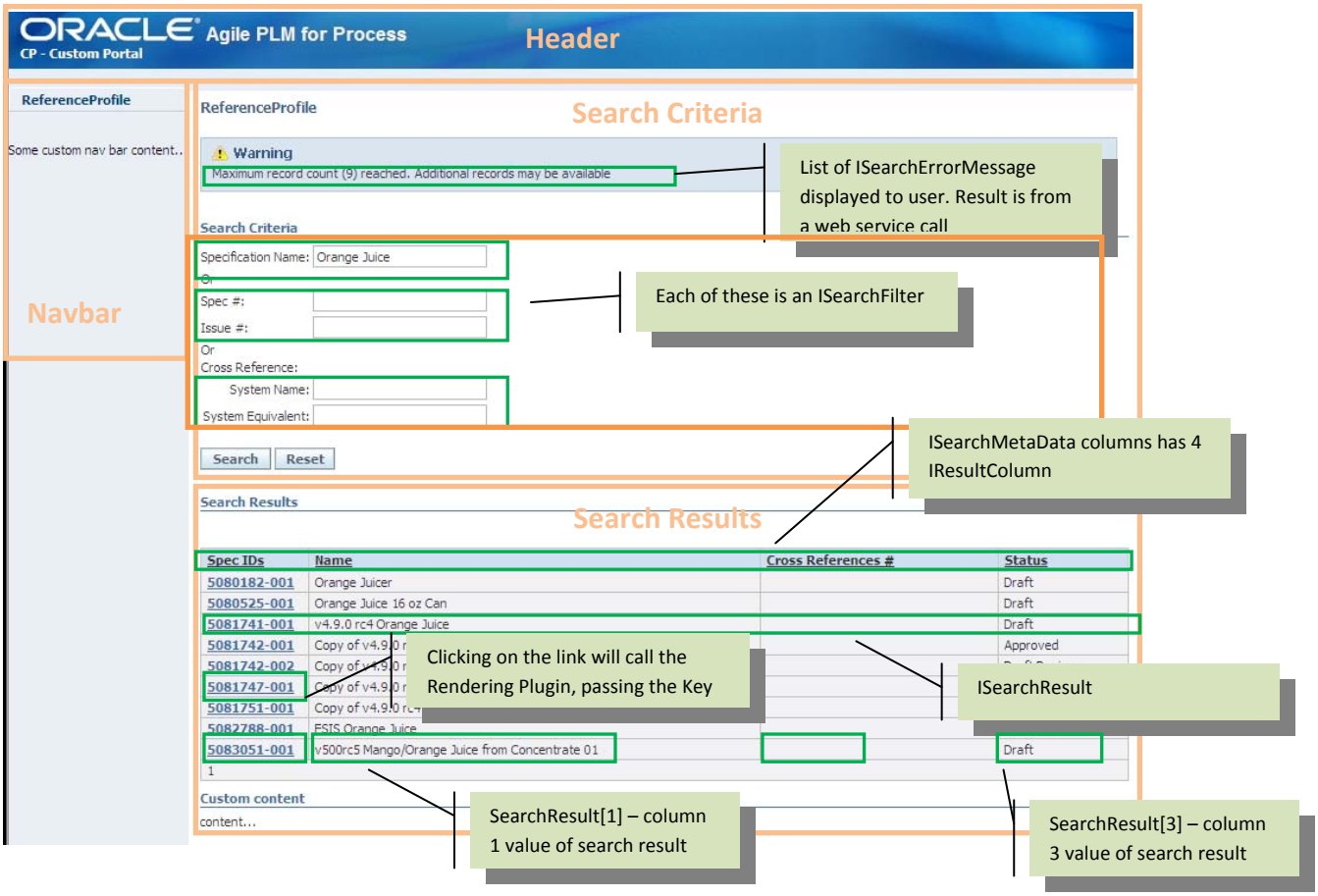
ISearchResult

The Search Plugin returns a collection of `ISearchResult` objects. The SearchResult contains a Key property, which is passed to the Rendering Plugin. This can represent a Spec Number, cross reference, etc, which the Rendering Plugin will then use to retrieve additional information for display purposes.

The indexer returns the value of the column to display, where the index represents the column index.



The `IItemKey` interface is a flag interface, with no implementation requirements.



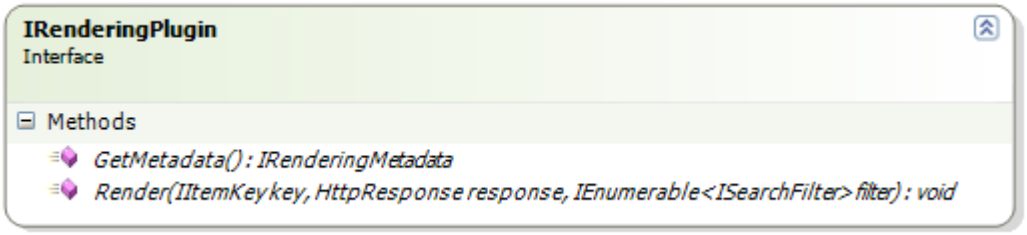
Rendering Plugin Structure

Rendering data involves displaying the desired output results once the search results are displayed and a result item is selected. The Rendering Plugin configured in the Data Admin screen for a View is called to display the result, either in the same window or in a pop-up.

> For instance, the RenderingPlugin could receive a specification number, call one or more web services to retrieve certain specification information, such as the list of Allergens, create an XML version of the data, and send it to Oracle BI Publisher for PDF rendering and printing. Alternatively, the renderer could use the data from the web services to create an HTML representation of the data.

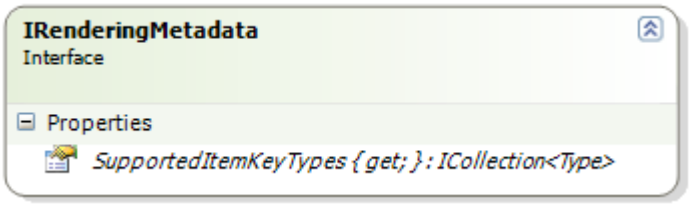
IRenderingPlugin

The `IRenderingPlugin` interface contains a Render method, which is responsible for display the final result item, and a `GetMetadata()` method which returns the support items for that rendered.



The `Render()` method takes the key passed in from the Search Results selection, the Http Response, and the Search filters. It is responsible for rendering the final output of the View to a stream. The `HttpResponse` object must have the `ContentType` property set to the desired MIME Type, and the file extension for the format it is rendering.

IRenderingMetaData



Detailed examples of using the various interfaces described above will be demonstrated in the Reference Implementation section below.

CustomPortal Control

The CustomPortal.ascx class (user control) is the controller for the entire search process. It delegates to the Header, Navbar, Search Panel, and Results Panel controls to render the page. It reads Portal configuration details from the database for the named Portal, loads the Filter Plugin control and the Search and Rendering Plugins.

Any ASPX file may serve as a portal page by including a CustomerPortal control, and passing it the portal name, as such:

```
<form id="form1" runat="server">
        <wc:Portal Portal="Reference Portal" runat="server" >
                <CustomHeaderContent>
                        <wc:ApplicationHeader runat="server" ID="ctlApplicationHeader" />
                </CustomHeaderContent>
                <CustomNavbarContent>
                        Some custom nav bar content ...
                </CustomNavbarContent>
                <CustomContent>
                        Custom content ...
                </CustomContent>
        </wc:Portal>
</form>
```

Plugin Data Access Approaches

Implementing the Search, Filtering, and Rendering capabilities will require a way to access the Agile PLM4P data. This may be achieved use existing web services, direct database queries, using PLM4P internal data objects and services, or some combination of all three. Please review the Accessing Data section above for an overview of these approaches.

In this section, we will discuss these approaches in more technical detail.

Using Web Services

The Agile PLM for Process Web Services API provides over 40 web services that can be used for various data access needs. Incorporating web service calls into the search and rendering process is fairly simple, but requires some analysis of how to orchestrate the various web service calls.

The web service calls may be used in multiple steps of the search process, namely as part of the search, and then once a result item is selected, other web services may be called as part of the rendering process.

Visual Studio .NET version 8 provides simple ways to create web service references. Using the Add Service Reference feature will create the service reference, and also configure the service endpoint information in the web.config or app.config of the project.

Calling Multiple Web Services

The results of many of the existing web service calls can (in part) be used as inputs to other web service calls. Chaining the web service calls in this manner allows for the easy retrieval of various data sets to compose a desired output.

> For example, using the `GetSpecForCriteria` web service (to search for Approved Ingredient Specs that have been modified in the last month, for instance) will return a list of specification numbers (and a list of cross references). The spec numbers can then be used as input parameters when calling the `GetSpecSummary` web service to retrieve the spec names for those specifications.

Note that in Visual Studio .NET, creating a web service client for each service contract will set up a unique namespace for each service. The common data contracts in these services therefore are not equivalent objects. To better enable cross client coding, one could modify the classes by adding a shared interface and program for that interface.

> For instance, the `tMessage` class is returned in a list as part of every web service call. It contains an error message code and description for any warnings/errors that occurred in the web service calls. Updating each `tMessage` class to add the `ISearchErrorMessage` interface will allow the use and display of these errors in the Search UI.

Web Service Parameters as Search Criteria

Many of the web service calls that retrieve specification data use input criteria that impact which specifications are returned. For instance, the AllowApprovedSpecOnly flag, passed into many calls that retrieve specs, dictates whether the specification(s) returned must be in an Approved workflow status or not. If this behavior should be user selected during the search process, then this flag could be a Search Filter that users can select, and its value would then be passed into the web service call.

Web Service User Authentication

The data returned from each web service call may depend on the calling user's security privileges: Specifications that the user does not have Read permission, or Business Unit visibility to, will not be returned; Object Level Security rules are enforced for the calling user for Custom Sections and Extended Attributes; etc.

There are two ways that the PLM4P Web Services authenticate users for web service calls:

1. Using one declared user for *all* the web service calls. This requires the username be entered in the environmentvariables.config when setting up the Web Services. The web service client will therefore not pass in any user information.
2. Passing in the username and password for each and every web service call. This requires the web service client to create and pass in a UsernameToken.

Details of the configuration of the Web Service API can be found in the *Agile Product Lifecycle for Management for Process Web Services Guide*.

The approach used by the web service clients must adhere to the way authentication is configured for the Web Services API.

Using Direct Database Queries

Required data that is not available via the current Web Services may be retrieved using custom database queries. Read access to the database and knowledge of the database structure will be required. The Database Schema documentation available in the Extensibility Pack provides a thorough view of the many database tables and their relationships to other tables. It also provides example SQL code to assist query development.

The following is an example of a direct database query:

```
public static IList<INutrientProfile> QueryForTradeSpecNutrientProfiles(string specNum, string
issueNum)
{
        List<INutrientProfile> nutrientProfiles = new List<INutrientProfile>();
        IXDataManager manager = AppPlatformHelper.DataManager;

        string sql =
                @"SELECT distinct bi.pkid, bi.* FROM gsmNutrientProfile bi
                        inner join gsmSpecNutrientProfileJoin nj on nj.fkNutrientID = bi.pkid
                        inner join gsmBaseTradeSpec ts on ts.pkid = nj.fkSpecID
                        inner join specsummary ss on ss.SpecID = ts.pkid
                        where ss.specNum = '" +
                specNum + "' and ss.IssueNum = '" + issueNum + "'";
```

```
        using (IDataReader r = manager.newQuery().execute(sql))
        {
                while (r.Read())
                {
                        INutrientProfile nutrientProfile = (INutrientProfile)
manager.objectFromID(r.GetString(0), r);
                        nutrientProfiles.Add(nutrientProfile);
                }
        }

        return nutrientProfiles;
}
```

However, using direct database queries bypasses all security logic for the user, unless that logic is then implemented in the query.

See the Reference Implementation code for detailed examples of using direct database queries.

Configuring Branding

User interface branding changes can be made by modifying the default CSS stylesheet TestCustomPortalControl.css, or by creating a new stylesheet and changing the reference to it from each portal page.

The CSS classes include:

cssxCustomerPortalHeader: assigned to the div element enclosing the header. The header contains an `h1` for the title.
cssxCustomerPortalNavBar: assigned to the navbar element, which is an `ul` element. Each view is an `li`.
cssxCustomerPortalContent: assigned to the div element enclosing the search criteria and search results. The view name is an `h2`, the section titles are `h3` elements. Other HTML elements include links, buttons, selects, and text controls.
cssxBaseGrid: assigned to the grid containing the search results.

Reference Implementation

DISCLAIMER:

The reference implementation provided in the Extensibility Pack, and described herein, is for demonstration purposes only and is not for use in production systems.

Overview

The existing reference implementation is called MockCustomPortalPlugins and located in the ReferenceImplementations folder of the Extensibility Pack.

This reference implementation accomplishes the scenario described the Business Requirement section below.

Business Requirement

The reference implementation described herein attempts to implement the following scenario:

1. Allow users to search for Trade specifications by selecting one of the following criteria:
    - Spec name, **or**
    - Spec number, issue number, **or**
    - Cross reference
2. Allow admin users to configure whether the specs returned should be restricted to Approved status only.
3. Search results should include:
    - SpecID (spec number-issue number)
    - Spec Name
    - Cross References
    - Workflow status
4. Selecting a Trade spec should display a PDF in a new window. The following data should be returned in the PDF:
    - Spec Summary information
    - Brand Information
    - The Bill Of Materials information for the related ingredient spec (and formulation context)
    - The trade spec's nutrient profile's Custom Section data for a specific custom section

See the Code Walkthrough section for implementation details.

Contents

The reference implementation contains the following components:

A portal page that accesses a portal named "Reference Portal 2" that should be configured in the PLM4P Data Admin screens for Custom Portal, as described in the Configuring Portal Pages and Views section.
The page may be accessed as follows:
`http://<servername>/customPortal/MockCustomPortalPlugins/portals/SamplePortal.aspx`

Plugins to fulfill the Business Requirements
- A Search Plugin,
- An Admin Filter Plugin
- A Filter Plugin
- A Rendering Plugin
  - Various user controls (.ascx files) used to render the data

Web Service References – web service clients that connect to the Web Service API
- The web service endpoints are configured in the web.config, and **must be modified. Please see the Installation section below**

Installation/Set Up

Be sure to have installed the CustomPortal web application and the Web Services API prior to installing the reference implementation. See the Installation section above from details.

To set up the MockCustomPortalPlugins reference implementation, you must do the following:

1. Install the MockCustomPortalPlugins library – see section 6.2.1
2. Configure the Portal and a View in the Agile PLM4P Data Admin screens – see section 6.2.2
3. Configure a new Custom Portal Profile for your user – see section 6.2.3
4. Add a custom section with 3 columns that will be used in the example – see section 6.2.4
5. Update the MockCustomPortalPlugins\web.config file to specify the web service endpoints for the web services used in the reference implementation.

Installation

The reference implementation code will also need to be installed as follows:

1. Follow the instruction in the Custom Portal Web Application Setup section.
2. Place your custom web application or library (in this case, MockCustomPortalPlugins) in the <PLM4P directory>\Web directory. (other locations may be applicable).
3. Copy the .dll of your custom library/web application (in this case, MockCustomPortalPlugins .dll) into the CustomPortal\bin directory.
4. Create a new virtual directory within the CustomPortal virtual directory that points to your custom web application (in this case, name it MockCustomPortalPlugins).
5. Restart IIS.

Data Setup

You must set up the following configuration in the Data Admin screens:

1. In the Custom Portal screen, add a new Portal as follows:
   Name=”Reference Portal 2”
   Cache Timeout =any value, ex: 30 (minutes).
   Status=Active
   Save your changes.

2. In the Custom Portal Views screen, add a new View as follows:

Name=”Reference View 2”
Search Plugin = ”`Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSearchPlugin,MockCustomPortalPlugins`”
Rendering Plugin = “`Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSummaryRenderingPluginFactory,MockCustomPortalPlugins`”
Filter Controls = “`/MockCustomPortalPlugins/FilterPlugins/SpecSearchFilterControl.ascx;/MockCustomPortalPlugins/AdminPlugins/DefaultAdminFilterPlugin.ascx`”
Note that the filter controls value includes two (2) Filter controls separated by a semicolon. The first value is the filter control used during the search, and the second value is the AdminFilter control used in the Profile Management screen.
Open New Window = checked
Status=Active

Save your changes

Custom Portal Views

	Name	Search Plugin	Rendering Plugin	Filter Control	Open New Window	Status
	Reference View	Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSearchPlugin,MockCustomPortalPlugins	Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSummaryRenderingPluginFactory,MockCustomPortalPlugins	/DefaultSearchPlugin/DefaultSearchFilter.ascx;/MockCustomPortalPlugins/AdminPlugins/DefaultAdminFilterPlugin.ascx	☑	Active
	Reference View 2	Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSearchPlugin,MockCustomPortalPlugins	Class:MockCustomPortalPlugins.ProdikaApiPlugins.SpecSummaryRenderingPluginFactory,MockCustomPortalPlugins	/MockCustomPortalPlugins/FilterPlugins/SpecSearchFilterControl.ascx;/MockCustomPortalPlugins/AdminPlugins/DefaultAdminFilterPlugin.ascx	☑	Active

1

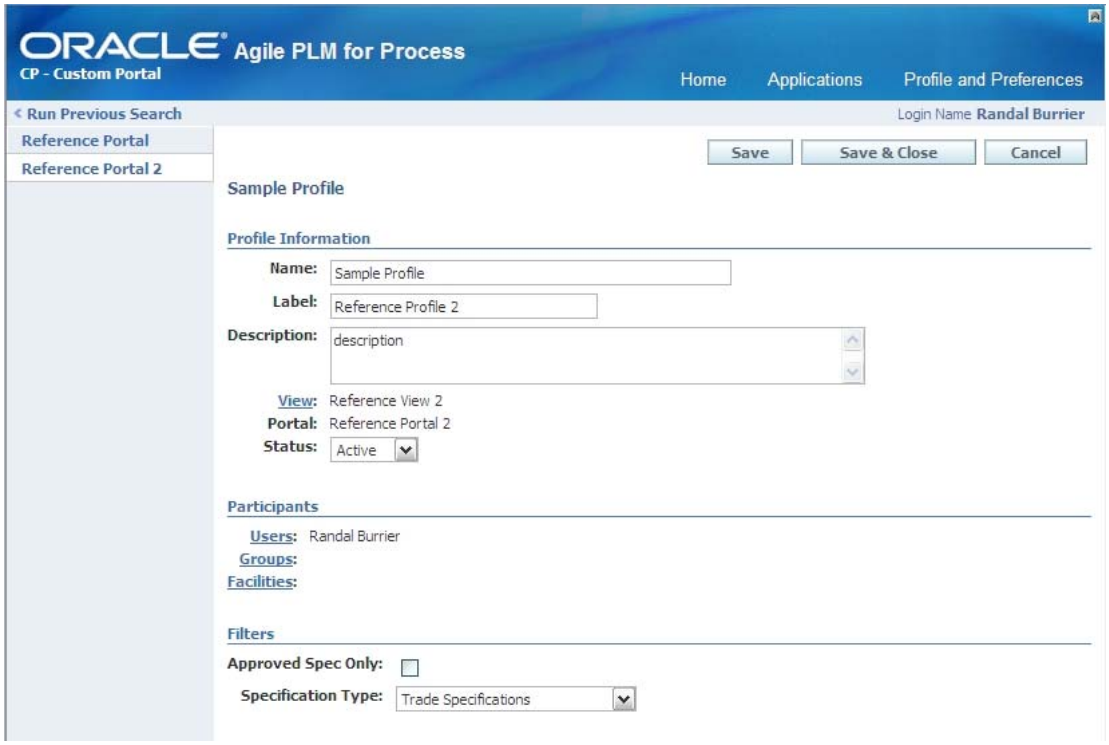
3. Perform a cache flush event, or restart IIS.

Custom Portal Profile Setup

You will need to set up a Portal Profile for your new Portal View

1. Go to the Custom Portal Administration page: http://<servername>/customPortal/Admin/PortalAdmin.aspx
2. Select your new portal (Reference Portal 2)
3. Add a new profile, using the New Profile button.
4. Set up the profile information by selecting the View named Reference View 2. This will load the Filters section to allow you to configure the Admin Filters. **This will only work if you have set up your Custom Portal and the MockCustomPortalPlugins correctly**.
    a. Select a valid user that should have access to this custom portal

b. Be sure to select trade specification for this scenario.



5. Save & Close the profile.
6. You should now be able to log in to the Custom Portal as the configured user and try the implementation: http://<servername>/customPortal/MockCustomPortalPlugins/portals/SamplePortal.aspx

Custom Section Setup

The reference implementation loads a specific custom section for a trade spec's nutrient profile. In this implementation, we simply create a new custom section that contains three columns, not including the row name column.

We use the CustomSectionNumber of 1000381, so you will either need to update your custom Section number through a database script to this number, or change the `SpecSummaryRenderingPlugin.cs` class, by modifying the `GetCustomSectionsForNutrientProfile` method and altering the custom section number there. You will need to rebuild/recompile the MockCustomPortalPlugins project if you do this.

Make sure this custom section is available for Nutrient Profiles, and **add this custom section to the nutrient profile of the trade specification(s) you are testing**.

Web Services Setup

The reference implementation calls several web services contained in three separate service contracts. These service contracts are accessible via three service endpoints, which are specified in the MockCustomPortalPlugins\web.config file.

The endpoint server name values should be modified to the server name of the server hosting the Web Services API. Various other settings (such as the timeouts, etc.) are pre-configured, but can be changed as needed or removed to use the default WCF values.

Verify Results

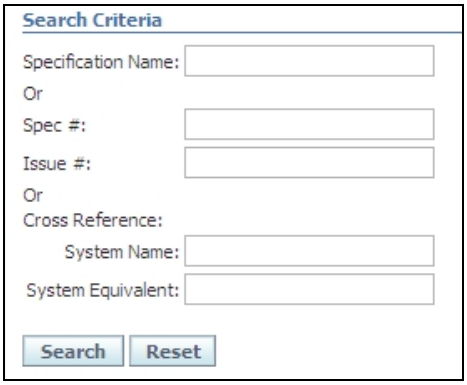
At this point, you should be able to use the reference implementation to search for trade specs and view their results. Search for the trade specification that contains a nutrient profile with the custom section you created, using one of the search criteria. Select a search result entry and verify that a PDF is rendered with the appropriate data.

Code Walkthrough

To fulfill the above Business Requirement requirements, we will take the following approaches:

Creating the FilterPlugin

To allow the search as indicated in the Business Requirement's requirement 1 above, we must create a FilterPlugin that contains three search filters (`ISearchFilter`), one for each criterion. The FilterPlugin will be a new user control that will allow users to enter the search values.



SpecSearchFilterControl.ascx:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="SpecSearchFilterControl.ascx.cs"
Inherits="MockCustomPortalPlugins.FilterPlugins.SpecSearchFilterControl" %>
<table>
    <tr>
        <td><asp:Label ID="lblSpecName" runat="server" />:</td>
        <td><asp:TextBox id="txtSpecName" TextMode="SingleLine" Rows="1" style=" width:150px;
vertical-align:top; overflow:hidden; " runat="server" /> </td>
        <td></td>
    </tr>
    <tr><td colspan = "3"> Or</td></tr>
    <tr>
        <td><asp:Label ID="lblSpecNumber" runat="server" />:</td>
        <td><asp:TextBox id="txtSpecNumber" TextMode="SingleLine" Rows="1" MaxLength="8" style="
width:150px; vertical-align:top; overflow:hidden; " runat="server" /> </td>
        <td></td>
```

```
    </tr>
    <tr>
        <td><asp:Label ID="lblSpecIssueNumber" runat="server" />:</td>
        <td><asp:TextBox id="txtSpecIssueNumber" TextMode="SingleLine" Rows="1" MaxLength="3"
style=" width:150px; vertical-align:top; overflow:hidden; " runat="server" /> </td>
        <td></td>
    </tr>
    //… code removed for brevity
```

The code-behind uses the name of the input fields as the key of the `ISearchFilter` classes, and the value(s) of the inputs as the Operand(s). For example, for the Spec#-Issue# search, the spec# value is the `ISearchFilter`'s Operand[0] and the issue# value is the Operand[1].

SpecSearchFilterControl.ascx.cs:

```
public partial class SpecSearchFilterControl : System.Web.UI.UserControl, IFilterPlugin
{
 //… code removed for brevity
        public IEnumerable<ISearchFilter> Submission
        {
                get { return GetSubmittedCriteria(); }
                set { }
        }

        private IEnumerable<ISearchFilter> GetSubmittedCriteria()
        {
                List<ISearchFilter> filters = new List<ISearchFilter>();
                ISearchFilter nameFilter = GetSpecRelatedfilter(lblSpecName.Text,
txtSpecName.Text, null);
                if (nameFilter != null)
                        filters.Add(nameFilter);
                ISearchFilter specNumFilter = GetSpecRelatedfilter(lblSpecNumber.Text,
txtSpecNumber.Text, txtSpecIssueNumber.Text);
                if (specNumFilter != null)
                        filters.Add(specNumFilter);
                ISearchFilter crossRefFilter = GetSpecRelatedfilter(lblCrossReference.Text,
txtSystemID.Text, txtEquivalentValue.Text);
                if (crossRefFilter != null)
                        filters.Add(crossRefFilter);
                if (filters.Count > 1)
                        throw new ArgumentException("Only one of the three criteria (Spec Name,
Spec #, or Cross Reference) allowed to be submitted.");
                foreach (ISearchFilter filter in filters)
                {
                        yield return filter;
                }
        }

        private ISearchFilter GetSpecRelatedfilter(string key, string Oprand1, string Oprand2)
        {
                if (StringHelper.IsStringEmpty(Oprand1) && StringHelper.IsStringEmpty(Oprand1))
                        return null;
                SearchFilter c = new SearchFilter();
                c.Key = key;
                List<string> oprands = new List<string>();
                oprands.Add(Oprand1);
                if (!StringHelper.IsStringEmpty(Oprand2))
                        oprands.Add(Oprand2);
                c.Operands = oprands;
                return c;
        }
//… code removed for brevity
```

Note that the control enforces a limit of searching on one criteria at a time, but this could be changed with some changes to how the search process works in Step 3.

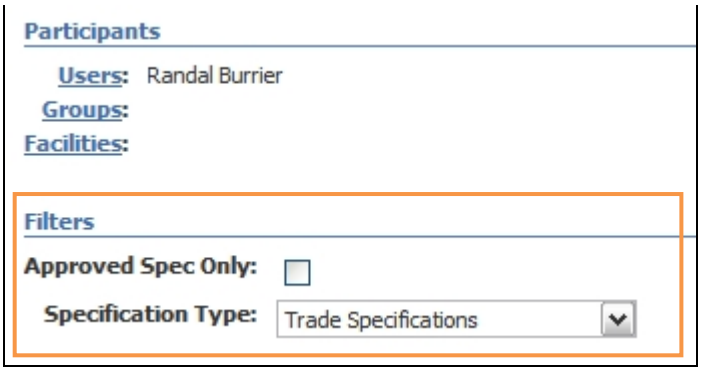
Creating the AdminFilterPlugin

To allow the admin search filtering in the Business Requirement's requirement 2 above, we will create an AdminFilterPlugin that contains two search filters – one that sets the Spec Type and the second that sets the Approved Only flag. [Note: we are making the spec type search filter editable even though our Business Requirement restricts the type to Trade Specs, so that we could reuse this Plugin for some other View.]

These search filters (combined with the Filter Plugin's user entered search filters) will be used as input criteria to two web services that will be used for the search:

1. The GetSpecForCriteria web service can search for specs using criteria, such as spec type, name, and cross reference, among others. However, it does not contain a searchable criterion for the spec number, since the spec number is the actual result. We therefore will use the GetSpecSummary web service.
2. The GetSpecSummary web service returns spec summary information for a given list of spec numbers. We will use this for searching by spec numbers.

The AdminFilterPlugin will also be a user control that will allow admin users to select from a list of spec types and a checkbox for the approved only value. It is accessible from the Profile management screen.



DefaultAdminFilterPlugin.ascx:

```
<%@ Control Language="C#" AutoEventWireup="true"
        CodeBehind="DefaultAdminFilterPlugin.ascx.cs"
        Inherits="MockCustomPortalPlugins.AdminPlugins.DefaultAdminFilterPlugin" %>
<%@ Register TagPrefix="xwc" Namespace="Xeno.Web.UI.Controls" Assembly="XenoWebControls" %>
<table class="cssxBaseList">
        <tr>
                <th><%= Translations["lblApprovedSepc"]%>:</th>
                <td>
                <xwc:BoundCheckBox id="chkApprovedSpec"
                        BoundProperty="AllowOnlyApprovedSpec"
                        runat="server"
                        checkedImageURL="../images/checked.gif"
                        uncheckedImageURL="../images/unchecked.gif"
                        IsReadOnly='<%# IsReadOnly %>'  />
                </td>
```

```
        <tr>
                <th><%= Translations["lblSpecType"]%>:</th>
                <td>
                        <xwc:BoundDropDownList
                                ID="ddSpecTypes"
                                BoundProperty="SpecType"
                                BoundItemsProperty="SpecTypes"
                                BoundItemsValueProperty="Key"
                                BoundItemsTextProperty="Value"
                                DefaultValue=""
                                IsReadOnly='<%# IsReadOnly %>'
                                RunAt="server" />
                </td>
        </tr>
</table>
```

The code-behind retrieves any saved values for the filters from the current Profile, and must save any changes back to the profile.

The SpecType SearchFilter uses the SearchStrategyFactory.SPEC_TYPE value for the Key and the 4-digit object type (the key of the specTypes drop down) as the Operand[0] value.

The AllowApprovedSpecOnly SearchFilter uses the SearchStrategyFactory.APPROVED_SPEC_ONLY value for the Key and the checked status of the checkbox for the Operand[0] value.

DefaultAdminFilterPlugin.ascx.cs:

```
//… code removed for brevity
/// <summary>
/// push the updated filter data to the profile
/// </summary>
private void PushData()
{
        Profiles.SetSearchFilter(m_profile, GetFilters());
}

/// <summary>
/// get the current filters
/// </summary>
/// <returns></returns>
private IEnumerable<ISearchFilter> GetFilters()
{
        IList<ISearchFilter> list = new List<ISearchFilter>();
        if (SpecTypeFilter != null)
                list.Add(SpecTypeFilter);
        if (AllowApprovedSpecOnlyFilter != null)
                list.Add(AllowApprovedSpecOnlyFilter);
        foreach (ISearchFilter filter in list)
        {
                yield return filter;
        }
}

public ISearchFilter AllowApprovedSpecOnlyFilter
{
        get
        {
                if (m_allowApprovedSpecOnlyFilter ==null)
                        m_allowApprovedSpecOnlyFilter =
GetProfileFilter(SearchStrategyFactory.APPROVED_SPEC_ONLY);
                return m_allowApprovedSpecOnlyFilter;
        }
```

```
        set { m_allowApprovedSpecOnlyFilter = value; }
}

public ISearchFilter SpecTypeFilter
{
        get
        {
                if (m_specTypeFilter == null)
                        m_specTypeFilter = GetProfileFilter(SearchStrategyFactory.SPEC_TYPE);
                return m_specTypeFilter;
        }
        set { m_specTypeFilter = value; }
}

private ISearchFilter GetProfileFilter(string type)
{
        foreach (ISearchFilter filter in Submission)
        {
                if (filter != null && filter.Key.Equals(type))
                        return filter;
        }
        return null;
}

public IEnumerable<ISearchFilter> Submission
{
        get
        {
                if (m_submission == null)
                {
                        m_submission = Profiles.GetSearchFilter(m_profile);
                }
                return m_submission;
        }
        set { m_submission = value; }
}

public bool AllowOnlyApprovedSpec
{
        get
        {
                if (AllowApprovedSpecOnlyFilter != null)
                        return bool.Parse(AllowApprovedSpecOnlyFilter.Operands[0]);
                else
                        return false;
        }
        set
        {
                if (AllowApprovedSpecOnlyFilter != null)
                        AllowApprovedSpecOnlyFilter.Operands = GetOperands(value.ToString());
                else
                {
                        AllowApprovedSpecOnlyFilter = new SearchFilter();
                        AllowApprovedSpecOnlyFilter.Key =
SearchStrategyFactory.APPROVED_SPEC_ONLY;
                        AllowApprovedSpecOnlyFilter.Operands = GetOperands(value.ToString());
                }
                PushData();
        }
}
//… code removed for brevity
public void SetProfile(object profile)
{
        m_profile = (ICPProfile) profile;
}
…
```

The `SetProfile` method is called by the framework when loading the Profile Management screen. It is used to pass the profile to the admin filter control, so that the search filters can be retrieved and saved.

The `Submission` property's `get` accessor retrieves the search filters from the profile.

Searching and Displaying the Results

The following Web Service clients/references are set up in the reference implementation project, and are available using the WebServiceTools class:

GeneralSpecServices (used for GetSpecSummary and GetSpecNumbersForCriteria)
CustomDataServices (used for the GetSpecCustomSections)
BillOfMaterialsServices (used for GetOutputBOM)

Important: These web service references must be altered slightly for this reference implementation: `tResponseHeader` classes must implement the `IResponseHeader` interface and the `tMessage` classes must implement the `ISearchErrorMessage` interface.

Searching

We will use the `GetSpecNumbersForCriteria` and the `GetSpecSummary` web service calls to perform the search. The results of `GetSpecNumbersForCriteria` web service call will be used to retrieve the spec name, cross references, and status to fulfill the Business Requirement's requirement 3 above from the `GetSpecSummary` web service.

The `SpecSearchPlugin` delegates the Search process to a Search Strategy class, which uses the SearchStrategyFactory to create a Search Strategy based on the ISearchFilter list being passed in. For example, if the search filter used is the spec number, it will create a `GeneralSpecSearchStrategy`, while if the search filter is the spec name, it will create a `SpecTypeSearchStrategy`.

The `GeneralSpecSearchStrategy's` search process uses the web service transfer objects for the input, and converts the results into individual `SpecSummaryResult` objects used for display.

```
internal class GeneralSpecSearchStrategy : ISearchStrategy
    {

… code removed for brevity …

protected ICollection<ISearchResult>
SearchSpecSummaryByCriteria(MockCustomPortalPlugins.GeneralSpecServices.tSpecIdentifierCriterion[]
list, out ISearchStatus searchResult)
        {
            //setup input params for web service call
            MockCustomPortalPlugins.GeneralSpecServices.tSpecInputCriteria c = new
MockCustomPortalPlugins.GeneralSpecServices.tSpecInputCriteria();
            c.AllowOnlyApprovedSpec = _allowOnlyApprovedSpec;
            c.specIdentifierCriterion = list;

            //setup ouput params for web service call
            MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummaryWrapper[] summaries =
new MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummaryWrapper[] { };
            MockCustomPortalPlugins.GeneralSpecServices.tResponseHeader response =
WebServiceTools.GetgeneralSpecServicesClient().GetSpecSummary(c, out summaries);
```

```
            if (summaries == null)
            {
                summaries = new
MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummaryWrapper[0];
            }

            searchResult = DataTools.GetSearchStatus(response);

            return
Array.ConvertAll<MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummaryWrapper,
SpecSummaryResult>(
                summaries,
                delegate(MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummaryWrapper
input)
                {
                    SpecSummaryResult output = new SpecSummaryResult();
                    if(!StringHelper.IsStringEmpty(SpecType)&& input.SpecSummary.SpecType !=
int.Parse(SpecType))
                    {
                       return null;
                    }
                    output.SummaryWrapper = input;
                    return output;
                }
                );
        }
```

The web service's `tResponseHeader` contains the overall result of the operation and a list of messages. The `DataTools.GetSearchStatus()` method maps this response to the `ISearchStatus` output parameter `searchResult`. Note that **this method will only work if you have modified the web service client**'s `tResponseHeader` to implement the `IResponseHeader` interface and the `tMessage` to implement the `ISearchErrorMessage` interface.

The return code converts the tSpecificationSummaryWrapper array to a SpecSummaryResult array by passing in a delegate method to the Array.ConvertAll() method.

Calls to the SpecTypeSearchStrategy class perform a spec search using the spec name. This class extends the GeneralSearchStrategy class, uses the GetSpecNumbersForCriteria, then passes the result to the GeneralSearchStrategy.

Displaying the Results

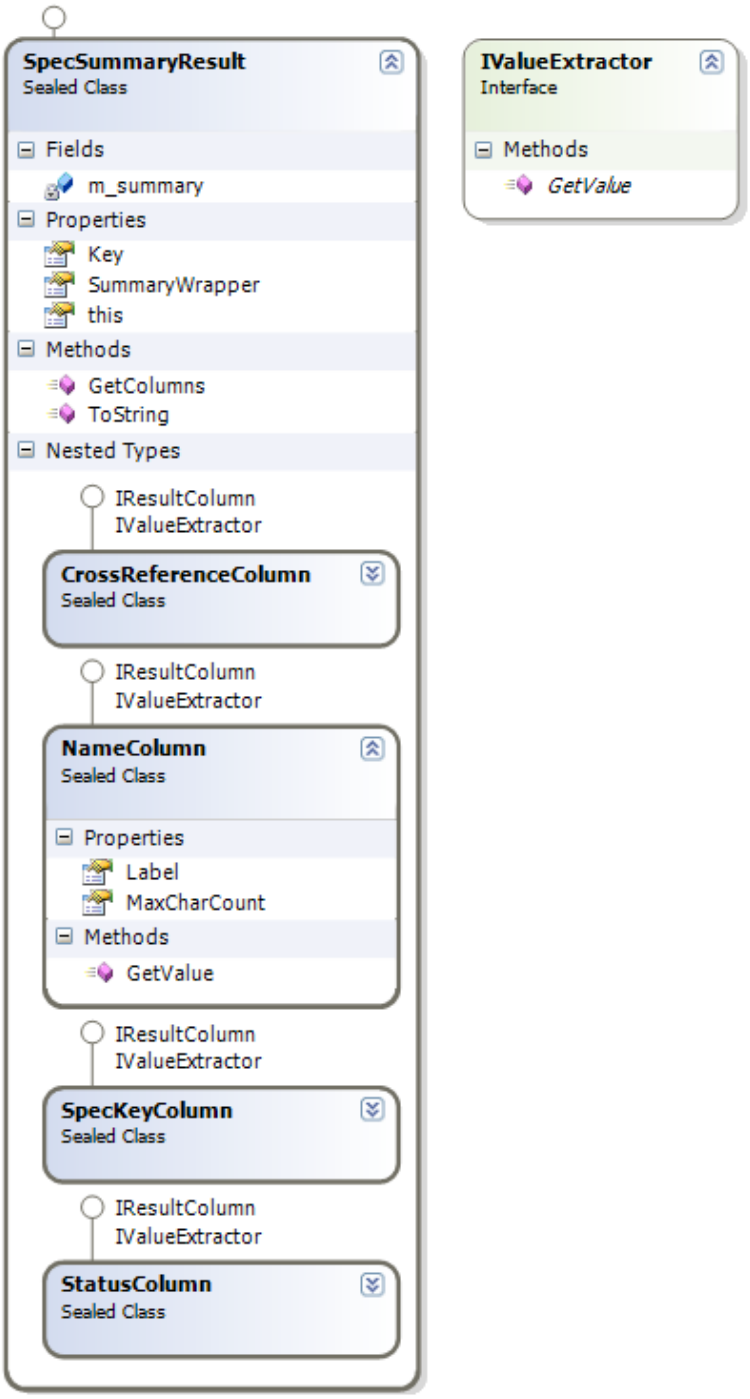
Our `SearchPlugin` defines the columns that will be used in the results through the GetMetadata method.

```
public sealed class  SpecSearchPlugin : IFactory<ISearchPlugin>,  ISearchPlugin
{

        /// ... code removed for brevity ...

        public ISearchMetadata GetMetadata(IPlatformUserDO user)
        {
                return new MyMetadata();
        }

        /// <summary>
        /// SearchResult's Meta data: column and key type
        /// </summary>
```

```
        private sealed class MyMetadata : ISearchMetadata
        {

                public ICollection<IResultColumn> Columns
                {
                        get { return SpecSummaryResult.GetColumns(); }
                }

                public ICollection<Type> SupportedItemKeyTypes
                {
                        get { return new Type[] { typeof( SpecSummaryResult ) }; }
                }
        }

}
```

The `SpecSummaryResult` class will provide access to the result columns and the values for each row. It implements the `ISearchResult` interface, which provides a Key property (which will pass the ItemKey to the RenderingPlugin) and an indexer (which provides access to the column values).

The `SummaryWrapper` property contains the actual results of the `GetSpecSummary` web service call. Passing the `SpecSummaryResult` object (as the `IItemKey`) to the `RenderingPlugin` allows the `RenderingPlugin` to access that data directly for rendering or for use in other web service calls.



`SpecSummaryResult` also contains internal classes for each column: each implements the `IResultColumn` interface, providing the Label property for the column name, and the `MaxCharCount` for the length. Additionally, each internal `ResultColumn` class implements a `GetValue()` method from a new interface called `IValueExtractor` that provides the value to display for each result.

Rendering the Result Data

Rendering the final output data indicated in the Business Requirement's requirement 4 involves several steps. To retrieve the data required, we will be calling a combination of web services and direct database queries.

The SpecSummaryRenderingPluginFactory class is used to render the required data to XSL-FO and then renders the FO into PDF.

The Render method takes an IItemKey as the object key, which is passed in as a SpecSummaryResult from the search results as discussed above. The Render also takes the HttpResponse and the list of search filters, which can then be used to filter out any other data retrieved.

```
public class SpecSummaryRenderingPluginFactory : IFactory<IRenderingPlugin>, IRenderingPlugin,
IRenderingMetadata
{
        ///...code removed for brevity...
        public void Render(IItemKey key, HttpResponse response, IEnumerable<ISearchFilter>
filters)
        {
                SpecSummaryResult summary = (SpecSummaryResult) key;
                MemoryStream foBuffer = new MemoryStream( );
                RenderFO( foBuffer, summary.SummaryWrapper.SpecSummary, filters );

                ConvertFoToPdf( foBuffer, response.OutputStream );
                response.ContentType = MimeType;
        }

        public ICollection<Type> SupportedItemKeyTypes
        {
                get { return new Type[] { typeof(SpecSummaryResult) }; }
        }
...
```

The RenderFO method creates a new Container instance, which is a subclass of Page.

```
        private static void RenderFO(Stream buffer,
MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummary summary,
IEnumerable<ISearchFilter> filters)
        {
                HtmlTextWriter writer = new HtmlTextWriter( new StreamWriter( buffer ) );
                StringBuilder sb = new StringBuilder();
                Page container = new Container( summary, filters);
                container.DataBind();
                container.RenderControl( writer );
                writer.Flush();
        }
```

It is the Container class that then makes several other data access calls to retrieve all of the required data, adds the SummaryFO.ascx user control, and sets its properties with the resulting data.

```
private sealed class Container : Page
{
        public Container(MockCustomPortalPlugins.GeneralSpecServices.tSpecificationSummary
summary, IEnumerable<ISearchFilter> filters)
        {
                bool allowOnlyApprovedSpec = GetAllowOnlyApprovedSpecFlag(filters);
```

```
                //set Spec summary info
                Control summaryFOControl = LoadControl(
"~/MockCustomPortalPlugins/ProdikaApiPlugins/SummaryFO.ascx" );
                ReflectionHelper.SetPropObject( summaryFOControl, "Summary", summary );

                ICollection<ISearchResult> billOfMaterialsResults =
GetBillOfMaterialsResults(summary, allowOnlyApprovedSpec);
                if (billOfMaterialsResults != null)
                        ReflectionHelper.SetPropObject(summaryFOControl, "BillOfMaterials",
billOfMaterialsResults);

                //find the brand information for trade spec
                IList<IBrandInformation>  brands =
DataTools.QueryForTradeSpecBrands(summary.specificationIdentifier.SpecificationNumber.SpecNumber,
summary.specificationIdentifier.SpecificationNumber.IssueNumber);
                if (brands != null && brands.Count > 0)
                        ReflectionHelper.SetPropObject(summaryFOControl, "BrandInfo", brands[0]);

                ICollection<ISearchResult> customSections =
GetCustomSectionsForNutrientProfile(summary, allowOnlyApprovedSpec);
                if (customSections != null && customSections.Count > 0)
                        ReflectionHelper.SetPropObject(summaryFOControl, "CustomSections",
customSections);

                Controls.Add( summaryFOControl );
        }
```

The GetBillOfMaterialsResults method finds the Trade Spec's ingredient specification and the formulation that created it using a direct database query, and then calls the GetOutputBOM web service to get the Bill of Materials information. Note that the database query could actually be replaced by one of the operations in the SpecRelationshipServices web services.

The Trade Spec's brand information is retrieved via a direct database query.

Then, a specific custom section is retrieved for the Trade spec's nutrient profile.

> This reference implementation uses the custom section with customsectionnumber 1000381, and the results that are later rendered are expecting that information, including the specific columns. See the Custom Section Setup section for details.

The nutrient profile is retrieved through a direct database query (it, too, could also be replaced by one of the operations in the SpecRelationshipServices web services). The GetSpecCustomSection web service is then called for that nutrient profile and custom section number.

Note that error handling from the web services is not handled here, but ideally, any errors that result from the web services should be bubbled up to the user. For web service error handling examples, see the Search Plugin and how it returns messages to the user.

The SummaryFO.ascx control includes four other user controls that contain FO tags to render the required data as FO.

Once the FO rendering is complete, a PDF can be generated from the FO results. In this reference implementation, we use an internal PDF renderer, called Apoc, to generate the PDF.

```
private static void ConvertFoToPdf( Stream foInput, Stream pdfOutput )
{
        foInput.Seek( 0, SeekOrigin.Begin );
        ApocDriver driver = ApocDriver.Make();
        driver.Options = new PdfRendererOptions();
        driver.Render( foInput, pdfOutput );
}
```

However, other tools, such as Oracle's BI Publisher, could easily be used to render the PDF instead.

At this point, the PDF should be rendering to the user, and the Business Requirement is fulfilled.

Further coding details can be found in the MockCustomPortalPlugins project.