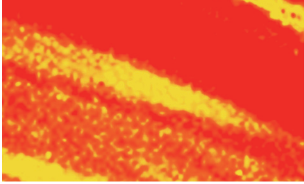


xPoint

Extensibility guide

Table of Contents

Table of Contents 2
Introduction 4
Definitions 4
Extensibility 5
Platform Extensibility framework 6
Platform attribute programming 6
Platform CMS editors 7
Content types 12
Content containment model 13
Custom design support 14
Platform API 14
Page templates 15
Design templates 16
Administrables 17
Using administrables 17
API 18
CCL integration 18
CMS presentation 19
Solutions API 19
Creating a solution 20
Registration 20
Search API 22
Search structures 22
Resource optimization 25
[Resource manipulation (File themes)] 25
Security 25
Identity 25
[Principal] 26
Overview 27
Resources 27

Deep system extensibility. 29
Introduction 29
How are objects organized in the platform 29
Entities 30
Types that support IObjectBind 31
Types that support IFieldBind 31
Component extensibility. 32
Introduction 32
Component system 32
CCL core functionalities 37
Packages and Licensing 38
Packages. 38
Introduction 38
Resource packages 38
Add-ons 38
ScriptServer Services 38
Zipping files 39
Uploading a package 39
Licensing in ScriptServer 40
Introduction 40
Workflow for licensing components 40
Development licensing 40
Licensing restrictions 40
Appendix 1 – Integration core 42
Terms 42
Architecture 42
Client model 43
WCF server 43
Endpoint 44
Actions 44
xPoint integration core 44
Communication manager 44

Introduction

This document includes the main extensibility points for the ScriptServer platform. It describes the programming interfaces and the coordination of the abstracted functionality from the platform view. They will be incrementally introduced from the smaller element to the respective containers.

This document includes:
- Content type extensibility
- Page templates
- Designer templates
- IObjectBind / IFieldBind interface hierarchy and coordination
- Components and CCL
- Packages and licensing
- Examples and usage

Definitions

The following definitions will be used across the document.

- *Attribute*
  An attribute is the actual definition of data blocks that are been saved to the database. The platform analyzes the data types and takes decisions for the persistence of the data. The search indexing is been done in this level.

- *Content type*
  A content type represents a platform Field paired with a UserControl for the presentation and logic of the content. Content types are derived from FieldUserControl base class. Content types are using attributes to expose their structure.

- *Page feature*
  Page feature is a content-type that is allowed only 1 per page. The pages features implement mostly functionalities, rather than providing content (e.g. page trashcan).

- *Editor*
  An editor extension gives the ability to the programmer to set how the attribute content is been interpreted while is been edited. Editors are user controls that are been derived from EditorUserControl base class.

- *Page template*
  A page template represents the actual page that will be rendered for the client, each time a virtual pages uses it as a template. The base class that all templates must use is TemplatePage, where is derived from System.Web.Page to utilize the aspx page model. A page template can have attributes / streams / relations to other pages that define their structure. A page can be populated by content types.

- *Design template*
  The design template is technically a master page template. It can be attached to pages to change design, and at the same time can have data that are been persisted from the database and be shared from all pages that uses the same instance.

- *User template*
  Not affiliated by any programming element, this is a combined result from page templates plus a design template (used as page and master page).

- *Component*
  Components are out-of-process instances that are been used in the CCL. Usually they will be utilized as closed source elements (dll). Components allow the platform to be extended with core functionalities without each component knows each other existence.

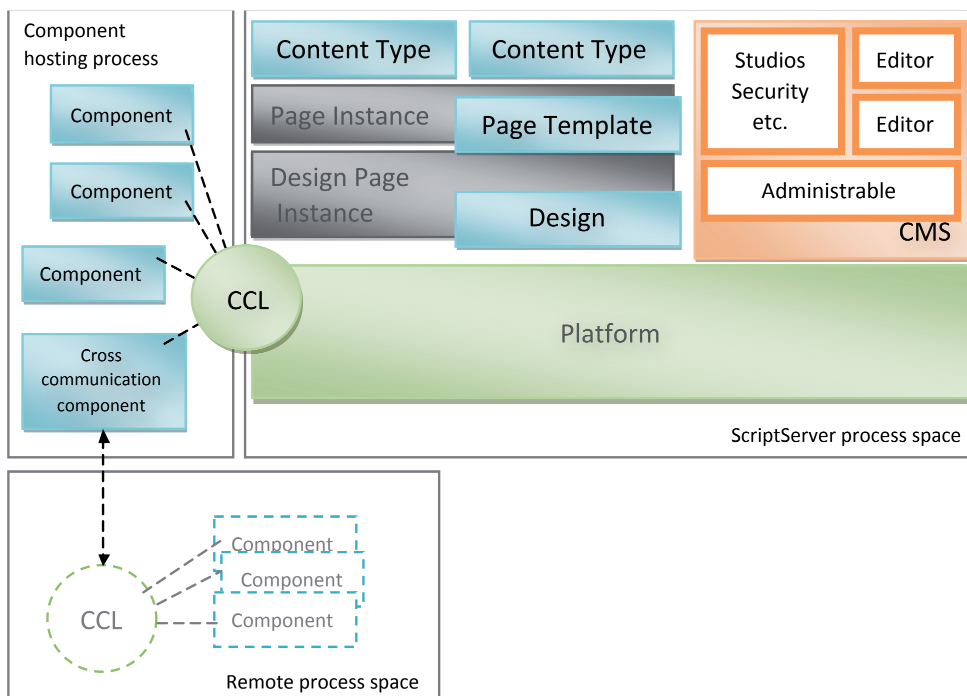
- *CCL*
  The Component Communication Layer (CCL), the responsible part for cross-component communication and discovery.

- *Administrative control*
  It is a programmatic element that allows the developer to enhance the administrative interface of the platform without having to re-develop common elements as rich-text editors functionalities.

- *Xml installation script*
  Xml file specifically structured to accompany packages / add-ons that instructs post installation configuration.

Extensibility

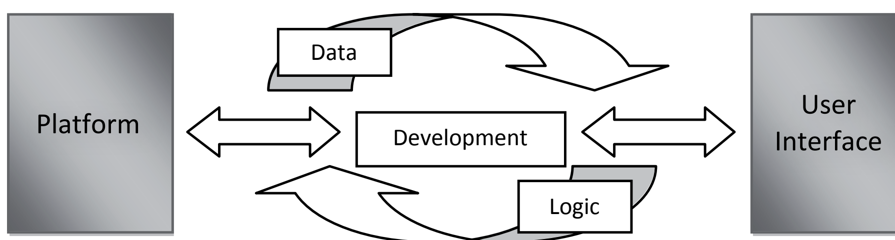
In this section the extensibility of the platform is analyzed from various levels. The following figure shows in details how the levels are been correlated to each other.



The extensible entry points for programming are been marked with blue. The areas marked with orange are the extensible points for CMS developers, and the green areas are for platform development, where both of them are not covered by this guide. Areas marked with grey / black are been handled by the platform.

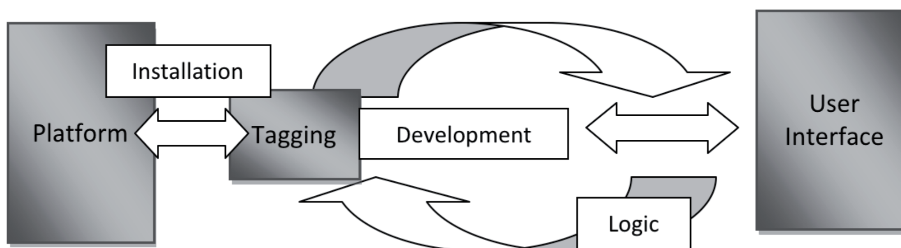
Platform Extensibility framework

In a common development scenario that a developer had to implement a certain functionality he was focused to orchestrate the logic both to the user interface (or to the next programming layer in generally) and to the platform. This process can be viewed in the following figure:



Platform Attribute programming

With the extensibility part of the platform we are making sure that the platform part is been handled automatically, so the developer can focus in the extension of the platform. This is shown in the following figure:



The classes that are taking part to the extensibility of the platform are in "ScriptServer.Extensibility.Attributes" namespace. They are been presented in the following list:

- *AttributePropertyAttribute*
  Defines that the property tagged is an attribute for the internal representation. This allows templates and content types to have different persistence properties dependent of their type, that at the same time they are persisted in database.
  Fields can have more specialized attributes, which can be meta-attributes, attributes that are not participating in search indexing, and localized, attributes that have multiple values per language.

- *RelationPropertyAttribute*
  Defines that property is a relation to a template. The property type must be a special type, an object that implements IObjectBind interface, the main extensibility entypoint for the base platform. All platform templates except fields are been connected to the platform using this interface, such as page templates and design templates. You can use lists of the selected type, or a single instance.
  Relations are not valid to field properties.

- *StreamPropertyAttribute*
  Defines that the property is been represented in the platform as a stream. This gives the advantage on using System.IO.Stream derived property types that will be successfully be saved to the database. If the object is not derived from Stream, then a serialization occurs on the instance.

Using the above markers the platform makes multiple decisions on how to use an object, also according to the object's type.

Additionally the following class-level attributes exist:

- *PhraseTitleAttribute*
  Allows the developer to add a title phrase directly on the object level for the CMS to use. This attribute can be used in properties and types.

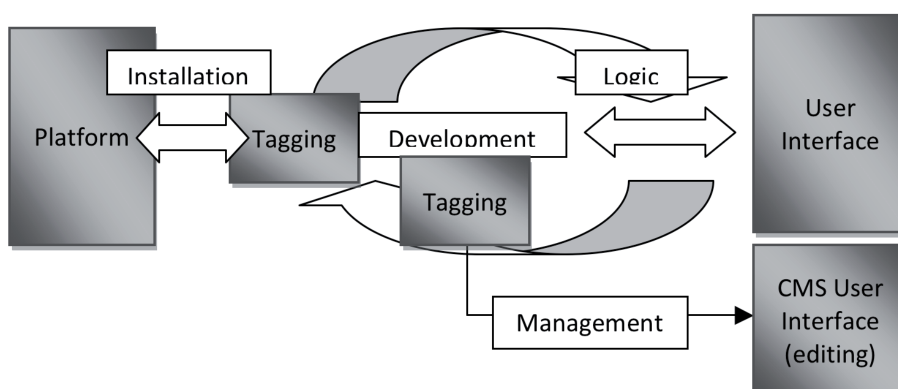
- *PhraseDescriptionAttribute*
  Allows the developer to add a description phrase directly on the object level for the CMS to use. This attribute can be used in properties and types.

- *InstallationActionAttribute*
  Allows the package developer to define post installation actions that will be executed after the package registration. This attribute can be used in the class level only.
  ```
  [InstallAction("<Virtual path to an xml file>")]
  ```

Platform CMS editors

Even if the platform knows how to handle the data types that are been provided by developers, it still missing the most important information for its CMS part: how the information is interpreted by user that edits it? A simple string data type can be html text, javascript code, username, read-only comment etc.

The above considerations are been solved by the usage of *editors.* This is an extension of the attribute model so the programmer in addition with the platform attributes tagging he adds the relevant information of the editor that handles that type.



Using Editors
Editors are been included in the ScriptServer.Web.UI.Editors.Attributes namespace. It contains the following attributes classes:

- *EditorAttribute*
  Specifies that the property been tagged can be edited from the web environment. When this is defined the programmer must also define the installed editor user control that will render the UI that will handle the editing function. For example the following line of code defines a usage of an editor that called Slider (Slider.ascx) and gives the initialization data for it:
  ```
  [Editor("Slider", 49, 1, 1)]
  ```

- *ExplicitEditorsAttribute*
  Specifies multiple editors that must be checked in the specific order, for complex editing generation. For example imaging the following property:
  ```
  public List<string> Links
  ```
  This property includes a list of information, and this means that we need a list of editors to edit the required strings. So we need to add the following line:
  ```
  [ExplicitEditors(typeof(IList), "Text.Rich")]
  ```
  The tagged line says to the editor core to search first any list capable editor, and after that, use the rich text editor to edit the inner information.
  Also types can be added as arguments. The type will be expanded to the namespace and type name string and an editor will be searched that is named in that way. For example the IList interface will be expanded to "System.Collections.IList".

The editor model does not necessarily need to be used on platform-supported classes. It can be used for any object, and the web core is capable to parse and edit it.

An example could be a list of links, controlling how many the will be rendered:

```
List<LinkData> links = new List<LinkData>();

  [PhraseTitle("…", "…")]
    [PhraseDescription("…", "…")]
    [ExplicitEditors(typeof(IList), "System.Object")]
    [AttributeProperty(IsMeta=true, IsLocalized=true)]
    public List<LinkData> Links
    {
        get { return links; }
        set { links = value; }
    }
int numberOfLinksShown = 10;

    [PhraseTitle("…", "…")]
    [PhraseDescription("…", "…")]
    [Editor("Slider", 49, 1, 1)]
    [AttributeProperty(IsMeta = true)]
    public int NumberOfLinksShown
    {
        get { return numberOfLinksShown; }
        set { numberOfLinksShown = value; }
    }
```

The link data are been handled by another structure that is also marked with editor attributes, and the editor "System.Object" that is a complex object editor, can present that to the editor user.

Q&As

Where I may find the CMS editors?

The installed CMS editors are included at ~/Files/Controls/Editors/. Because editors are an extensible part of the platform, it can be placed anywhere. Many other packages are adding editors in the system.

What are the provided functionalities of the editors?

1. Selection / RadioOptions : Using an enumeration, you can provide a single choice for the user, or a yes/no using a Boolean.
2. Checkbox: Yes/No using a Boolean.
3. DateTime: A date and time picker
4. Slider: Uses a slider to pick a number, int or float.
5. Number.Simple: Uses a textbox for number input, int or float
6. Text.Simple: Single line text with no format, as string
7. Text.Rich: Rich text editor as a string
8. Text.Rich.Preview: Rich text editor, showing a preview before editing, as a string
9. Text.Multiline: Multiline simple text
10. Text.Code: Code editor for html, css, javascript or any other language, as a string.
11. File.Pick: Picks a virtual uri of a file as string
12. Image.Pick: Picks a virtual uri of an image as string, includes preview
13. Page.Pick: Picks a page as a virtual uri, as a string
14. Pages.Multiple.Pick:: Picks multiple pages, and allows for ordering of the selection, as a List<string>

Developing an editor

The creation of an editor is needed when the customer must perform certain tasks to order and manage some content.

There can be several types of editor editing a specific type. For example the "System.String" type can be anything, like a Javascript resource or html text. When an editor is implemented is directing the user to take the correct actions, and produce a clean representation of the data that can safely be used from the system, and your logic. Additionally editors are protecting the user of taking invalid decisions, like selecting a page in the system that does not exist and creating a broken link.

Editors are derived from the class `ScriptServer.Web.UI.Editors.Model.EditorUserControl` and should be implemented as UserControl. The derived class must implement the following two members:

1. `public abstract void LoadValue(PropertyInfo property, object instance, Type valueType, object value);`
   Loads the value that is set to a property that is marked using the `Editor` or `ExplicitEditors` attributes. *This method is not guaranteed to be loaded after or before the "OnLoad" event.* Usually the application creates dynamically all the editors and place them in the control tree on "OnLoad" or "OnInit" event (depending on the implementation). In the platform CMS the "OnLoad" event is used.
2. `public abstract object SaveValue(PropertyInfo property, object instance, Type valueType);`
   Returns the value to save (persist in the system) that has been set by the editor. It is called after the editors are in the control tree.

Naming an editor

An important note has to be made for the editor naming. The application only uses the first part before the ".ascx" to identify the editor that will use. This means that in the installation time (so you need to install the editor to test it) the application maps the names with the ascx placement so it can discover it later. *You should not use the same names for the editors. The installation will overwrite the previous registration and will keep the last one.*

The CMS editors (that you can also see as examples) exist in "~/Files/Controls/Editors". There is no restriction in the placement of an editor, you may place it any where. Even if the placement is changed afterwards, the application will refresh the registrations when an update happens.

Editor API

The editors API provide the means to use editors in the automatic way as it is been used in the platform. The class that orchestrates editors is `ScriptServer.Web.UI.Editors.Model.Editors` and is static.

The advantage of the Editors model is that you can use them in any scenario, even in a class that is not a ScriptServer platform part. For example if we have the following implementation:

```
public class Comment
{
    [PhraseTitle("Example.Comment.Title", "Title for the comment")]
    [Editor("Text.Simple")]
    public string Title { ... }

    [PhraseTitle("Example.Comment.HtmlComment", "The comment")]
    [Editor("Text.Rich")]
    public string HtmlComment { ... }

    [PhraseTitle("Example.Comment.ShowMyEmail", "Show your email?")]
    [Editor("Checkbox")]
    public bool ShowMyEmail { ... }
}
```

(Some lines have removed for clarity)

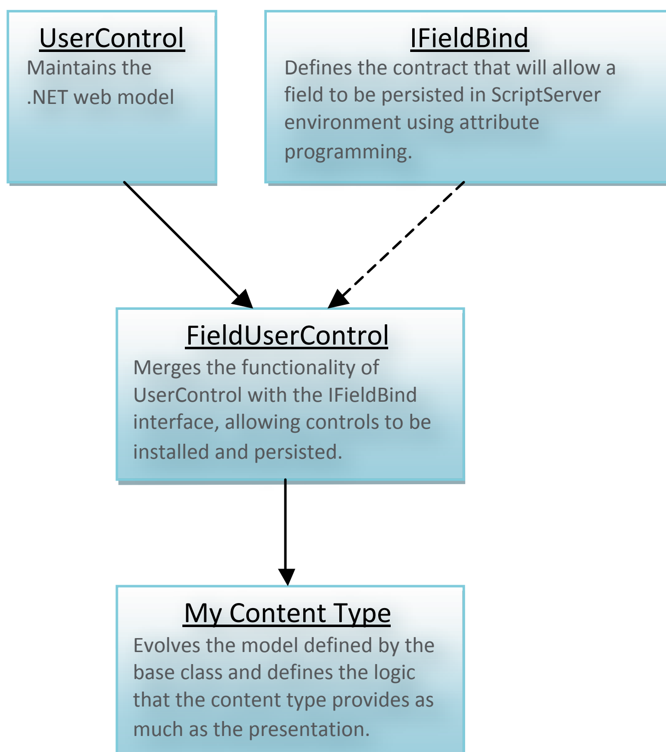
We can still use the Editors class to manipulate the content of an instance of the Comment class. The orchestration is been done by the following members:

- `List<EditorUserControl> GetAllEditors(object editedObject)`
  This method will retrieve all the editor user controls that they are been defined from the attributes in the properties of the object. Then you need to place them in your control tree to activate them.
- `void SaveEditors(List<EditorUserControl> editors, object editedObject)`
  This method will get the values of all the editors, and place them again in the instance of the editedObject, in the correct properties.

The Editors class is been used to remove all the programming tasks for lookup and loading for installed editor user controls.

Content Types

A content type is a definition of a FieldUserControl. The system uses the class definition to read the structural data that will be mapped to the schema in the platform. There is also the part that contains the programmer's logic according to what the content type does. While installing a type the platform is reading the structure of the object that is been registered, and from that moment and on any objects that are created can be persisted in the database. The API uses the following hierarchy to achieve this:



Content types are been separated in the following categories:

- The repeatable content types that are been managed by the users
- The unique content types that exists only one per page (page features)

The above behavior is been controlled by using a single attribute in the top level resource (that may be a page template, design template or a content type) named "SystemResourceAttribute". Depending on how the attribute is defined on the class, it gives a different meaning to the platform CMS. The following rules apply:

- `[SystemResource(IsUnique=true)]`
  This effect is applied only in FieldUserControl type. This defines that the FieldUserControl is a page feature, which means that is unique per page, and the users cannot manipulate it directly, but only through specific user interface.
- `[SystemResource]`
  The above defines that the class marked is not visible on CMS selections from normal users, e.g. when the user selects to create a page based on a template.

Additionally the following attributes can apply to a content type control:

- `[Category("ScriptServer.Categories.Example.Category", "<Title of the example category>")]`
  The category defines the tab that the content type will be provided to the user. Additionally each container can have a category allowance to allow/filter the available content using the above attribute (See FieldHost).
- `[Image("~/ExamplePath/image.png", "<Image type>", ImagePathType.VirtualPath)]`
  Defines an image that can be used to accompany the text, to help user visualize the content part. It can be added several times. The CMS supports "MenuImage" and "BigImage" types.

Additionally the PhraseTitleAttribute and PhraseDescriptionAttribute must be applied to give overall review to the user what this content-type is about.

The FieldUserControl extends all the platform elements with the model referred in the previous section. In addition provides some more functional elements for the derived content types:

- Provides a content instance containment model
- Provides a connection to the performance API, for minimizing and compressing resources
- Provides the connection with the platform API

Content Containment model

The field model defines a data variable that allows a content type instance to have a container. This allows another content type to act as a container to others, and maintain an even more complex structure.

Each FIeldUserControl includes a property named `ContainerFieldUniqueId` that allows the user control to persist its container value. This of course varies according to the implementation. The CMS uses server control Ids to map to specific hosts.

This ability has been implemented as a host in a content type called "FieldHost". This is the control that allows the user to drag and drop and rearrange the contents, while the page is in edit mode.

The same ability is been used by the trashcan control to keep track of the deleted content, and if desirable, to move the content to another container.

The container ability will be abstracted in the future release, and will be described by an interface to keep design capabilities known at other elements.

FieldHost
FieldHost is the user control that must be used in a TemplatePage derived page to specify a possible placement for other content types. The registration is of the form:

```
<Fields:FieldHost runat="server" ID="<Id of the container>"
Category="<Content categories allowed>" />
```

The categories that are implemented for the default CMS are always starting with "ScriptServer.Categories.CMS". The default CMS includes the default design elements like "ScriptServer.Categories.CMS.DefaultDesign". You may also select comma-separated categories for the content selection.

CMS Category
When a category is been set, then the available tabs that are been rendered in the application are been filtered using the the above category. Only the category chosen and the subcategories are allowed to be selected in this container from the user. This allows fine tuning in the editing part of the application, like specialized editors, email or page editors, allowing safely to select parts that match the content of the editable area.

Sharing Content
When a FieldHost control is been placed in a MasterPageTemplate derived class then the content is been placed on the design and not the page itself. This allows the application to share the content from the design, to all the pages that are been linked with this design.

The CMS knows this difference and renders the two different areas using different colors, to be recognized by the user.

Custom design support
When a FieldUserControl needs to allow to be designed then it should be linked with the "Save Design" edit mode button. Programmatically this is been done using the IDesignable interface. When the save design button is been pressed then all of the designable elements will be prompted by their implementations to save their state.

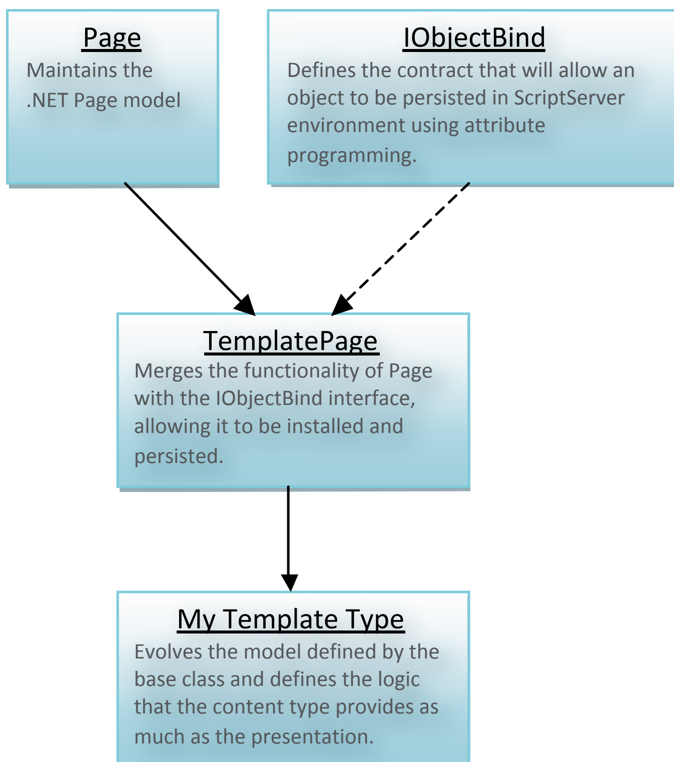
Platform API
The platform API that connects the object model with the object and field hierarchy is been implemented using the IFieldBind interface. The interface is been explained later in this document.

It provides the following implementation:
- CRUD content-type instance manipulation
- Strongly typed parent retrieval (IObjectBind - TemplatePage)

The interface with the attribute tagging support allows the platform to manipulate any derived type.

Page Templates

A page template is a definition of a TemplatePage derived type. Like the content type model the system uses the class definition to read the structural data that will be mapped to the schema in the platform. Template API uses the similar hierarchy architecture to achieve this:



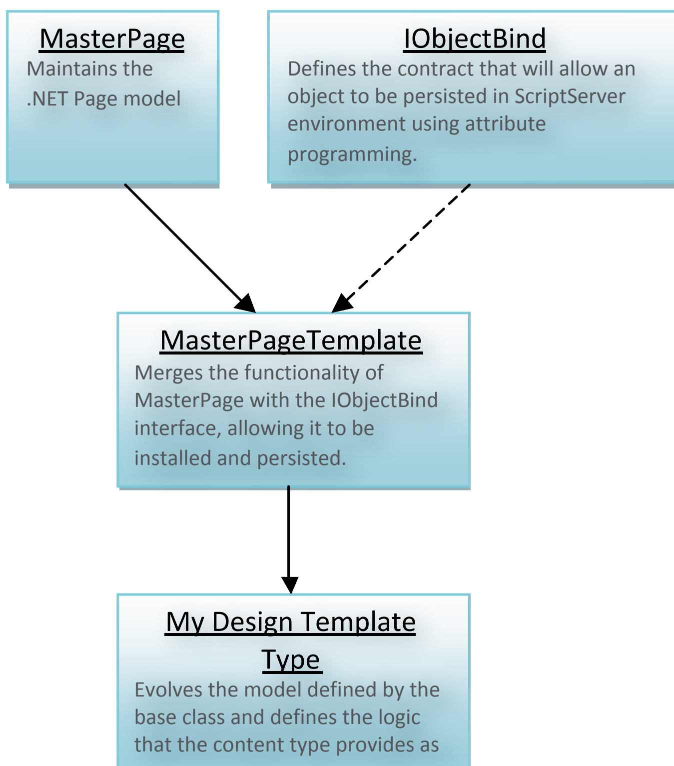
The base class extends all the platform elements with the model referred in the previous sections (the attribute based programming elements). The templates also allow the categorization of the instances by path, so each instance can be requested directly by it, or navigated using the object model (Parent and Children). In addition it allows direct access to the content types that are attached on it.

Design Templates

A design template is a template for design, which is in .NET world a MasterPage. Master pages can have content and functionalities that are been shared across a group of pages. This implementation gives the user the ability to have the same design pattern, but with different content or abilities.

All design templates that can be used from the platform are been derived from MasterP-ageTemplate. The class implements the IObjectBind contract, like the TemplatePage, so it is capable to be found by path and can host content instances.

The class hierarchy can be viewed in the following figure.



Administrables

Administrables are user controls that allow constructing and editing global application settings. They are been created in user controls form, and they are using a similar attribute model to allow defining settings.

Because of using the attribute model like all other platform parts, administrables needs to be licensed and installed to be used by the system.

Using Administrables

The base class that an administrable is deriving from is
`ScriptServer.Web.UI.Editors.Model.AdministrationUserControl` class.

The attributes are important to define for administrable usage:

- `[Administrable]`
  Any property that defines an administrative setting must have the above attribute. The property name with the administrable user control category will act as the full administrative value path. You can define editors on each property that you would like the system to assist user editing.
- `[Category("<Dot separated values>")]`
  The administrable class must have this attribute defining a categorization for the user control, and the administrative values.
- `[PhraseTitle("<Title Id>", "<Title>")]`
  `[Image("<~/V.Path>", "BigImage", ImagePathType.VirtualPath)]`
  `[Image("<~/V.Path>", "MenuImage", ImagePathType.VirtualPath)]`
  The administrative class must also have defined some presentation content, so the user can view what this administration control is about, when viewing the Settings Manager.

In the following example the behavior of the above elements are explained.

Example

The following administration user control is been defined in the CMS:

```
[Category("ScriptServer.Site.Caching")]
[PhraseTitle(…)]
[Image(…)]
[Image(…)]
public partial class Files_Templates_Settings_Caching :
AdministrationUserControl
{
    bool noCache = false;

     [Administrable]
     [Editor("RadioOptions")]
     [PhraseTitle(…)]
     [PhraseDescription(…)]
    public bool NoCache
    {
        get { return noCache; }
        set { noCache = value; }
    }
}
```

The above definition mean the following for the system:

- The category of this administrable is "ScriptServer.Site.Caching"
- The property because of the above has full name "ScriptServer.Site.Caching.NoCache" (you can use this from the API)
- The property will be edited using the editor "RadioOptions"
- The virtual path of the setting is /ScriptServer/Site/Caching/NoCache (you can use this on the API)
- The CCL path is "admin://ScriptServer/Site/Caching/NoCache" for setting and retrieving the specific value.

API

The Administrable API is defined in the static class named
`"ScriptServer.Web.UI.Editors.Model.Administration".`

- `GetAllAdministrativeCategories`
  Gets all the available categories that have been installed for this application.
- `GetAllAdministrativeParts`
  Returns a list of instances of all the installed administrative parts.
- `GetAdministrativePart`
  Returns an instance of a specific administrative part.
- `GetAdministrativeValue`
  Gets the value that has been set for a specific administration user control. Category and name may be used to retrieve the value, as well as the path's virtual setting.
- `SetAdministrativeValue`
  Same as above, but now sets the value that is been provided.
- `SetAdministrativeStringValue`
  CCL implementation of the administrative settings. When you try to set the value in the above example using "admin://ScriptServer/Site/Caching/NoCache?set=True" this method is invoked.

CCL integration

The API has defines 2 CCL entrypoints:

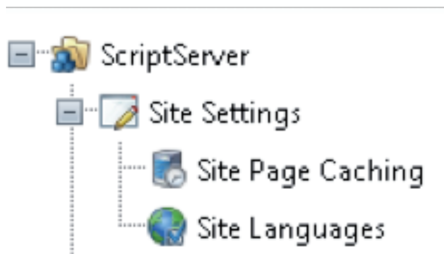
- Getting administrator settings
  ```
  ScriptServer.CCL.Components.Resolve(typeof(object),"admin://Scr
  iptServer/Site/LoginUrl")
  ```
- Setting administrator settings
  ```
  ScriptServer.CCL.Components.Resolve(typeof(string),"admin://Scr
  iptServer/Site/FirstPageUrl=/start")
  ```

The advantage of CCL entrypoints is that the application does not need to have direct dependency with the web platform (ScriptServer.Web.dll). It is been used mainly to support the core with callbacks to the outer layers. If you need to manage the administrative settings from the UI, it will have better performance to choose the API.

CMS Presentation

Each Administration user control that is installed in the system will be shown in the "Settings Manager". This will give the user a unified way to make changes in the software, plus the developer will not need to specify any administration interface, just the text and images.

In the category name the application will treat all the dot separated values as folders, except the last. The last will be the actual node for the user to view the control. In the above example for the caching we will see in the system:



Note that because we have also a control in the "ScriptServer.Site", this is merged with the tree.

Solutions API

Solutions are implemented in the system to help users identify or solve specific tasks that will or might arise from the use of the software (e.g. broken links).

Solution is considered to provide fixes to problems that involve uncontrolled user usage. You should use update for software components if there is any error or bug (through ScriptServer Services update).

The solutions are always acting against an error (Exception). All the system exceptions are been checked against a solution, so this is a heavy task.

The process of the solution execution starts by matching the solutions using the API `SolutionCenter.Match` method (the user can do that from the solution center).

Each solution is been checked for the link that it provides; the link might be simple information from a web page, or a link to an implemented solution. In any case a GET request is been constructed using the following variables:

- `ProposedSolutionId: int`
- `ApplicationExceptionId: int`

The first argument is required for a solution to get activated and start fixing. If the second argument is provided then the solution will be executed as a *single fix* (so it can be used through the system to enable solutions on demand as it is in the template manager).

A solution is been defined in the database using the following data:

- `string messageRegex`
  The regular expression that will run against the message of the error
- `string sourceRegex`
  The regular expression that will run against the source of the error

- `string urlRegex`
  The regular expression that will run against the url of the error
- `string stackTraceRegex`
  The regular expression that will run against the stacktrace of the error
- `string description`
  The description that will be visible to the user. The origin of the error must be described as well all the system changes. Make sure that the user understands the risks of solving an error if there are any
- `string link`
  The link of the solution. It can be any link, in any site. The application will append the solution id as well as the error id if it is provided
- `bool isSolution`
  True if the entry provides solution to a problem
- `bool isWarning`
  True if the error is a warning (in severity level) for the user
- `bool isError`
  True if the error is important (in severity level)
- `bool isInformation`
  True if the solution provides information about the problem to the user

[Explain how the application takes in account of the information above]

Creating a solution

A single solution in the system is been implemented by the `SolutionPage` class. It provides the API to solve one problem, or be used in batch like process, to solve multiple ones. Example of solution implementations can be found on "~/Files/Templates/SolutionCenter/Solutions" of your application, where the CMS solutions are stored.

The solution implementation includes the following methods:

- `SingleSolutionExecution`

The method that will be called against a matched error with a solution in the database. The automated process must use the information to take *safe* decisions to change the system to make it more usable and fast, or to gather data from the user.

- `Finish`
  The method that is called when a solution will be entering in the last process step. Normally the resources are been cleaned and the data are been formatted to the user as html in this method.

Registration

The programmatic way to add solutions in a system can be:

- Add solution through CRUD API
  `ScriptServer.Provisioning.Error.ProposedSolution`
- Add solution through
  `ScriptServer.ExceptionHandler.SolutionCenter.AddEntry`

The solutions can also be added in the database through the xml installation scripts, by calling CCL on the path:

```
[Path("add-solution:@any[|]@any[|]@any[|]@any[|]@any[|]@any[|]@bool
ean[|]@boolean[|]@boolean[|]@boolean")]
        static public string AddEntry(
string messageRegex, string sourceRegex, string urlRegex,
string stackTraceRegex, string description, string link,
bool isSolution, bool isWarning, bool isError, bool isInformation)
```

To define a component all the above elements must be defined.

Search API

Along with the extensibility model, there is a specific part for retrieving the information easily in the platform. The search API exists in the namespace ScriptServer.Search.

The main static class that includes all the search methods is the class "Find". Searching with this class always involves the same concept: The developer is searching in a property or content, and the relevant pages are returned, that include that content.

The search functionality is implemented in the platform so any IObjectBind with IObjectField combination can be searched.

When searching the system automatically determines the security of each object (e.g. pages), and if an object will not be able to instantiated, the search functionalities will skip it.

Search structures

To invoke a search in the platform you need to create a list using the available options and define the available order of the results.

The available options are been described by the classes `ScriptServer.Search.FindOption` and `ScriptServer.Search.OrderingOption`. When you need to define an option you need to instantiate a FindOption class. The available search and order options are been described in the following section.

Search Options

The type of search in the platform is broken in the following categories:

- `PageName`
  Searches using the page name, the page-part that consists the url of the page. It must be provided as a string type.
- `TemplateType`
  Narrows the search by searching a specific template type. The type must be provided as a string value.
- `Live`
  Search is narrowed to the live pages, the pages that the user is allowed to see. Live pages do not include non-published, expired, hidden (restricted) and system pages.
- `PublishedAndNotExpired`
  Search is narrowed on published and not expired pages only.
- `ExpirationOnBetween`
  Search is limited on the expiration of the pages between the provided dates. They must be provided with the option.
- `PublishingOnBetween`
  Search is narrowed by the publishing date of the pages, between two given dates.
- `CreatedDateOnBetween`
  Search is narrowed by the creation date between the provided dates.
- `ModifiedDateOnBetween`
  Search is narrowed by the modified date between the provided dates.
- `ByCreatedBy`
  Search is narrowed by the username of the user that created the pages. It must be provided as a string value.

- `ByModifiedBy`
  Search is narrowed by the username of the user that had last modified the page. It must be provided as a string value.
- `Expired`
  Narrows the search to the pages that are expired or not. It must be provided as a Boolean.
- `Published`
  Narrows the search to the pages that are published, or not published depending the value provided. It must be provided as a Boolean.
- `Deleted`
  Narrows the search to the deleted pages or the not-deleted pages. It must be provided as a Boolean. *The system in a default search automatically sets this value to false.*
- `Restricted`
  Narrows the search to the hidden (restricted) pages, or the non-restricted pages if false is provided. Must be provided as a Boolean.
- `System`
  Narrows the search to the system pages, or the non-system pages if false is provided. Must be provided as a Boolean.
- `AttributeStringValueSearch`
  Search a value by a string in the content.
- `AttributeBoolValueSearch`
  Search a value using a bool variable.
- `AttributeIntegerValueSearch`
  Search a value using an integer variable
- `AttributeDateTimeValueSearch`
  Search a value using a date variable
- `AttributeFloatValueSearch`
  Search a value using a float variable
- `AttributeName`
  Narrow search in a specific attribute name (the property name that is marked by the AttributePropertyAttribute)
- `FieldName`
  Narrows the search using a specific field name. The platform manages the names automatically, and is been used for specific controls and containers.
- `FieldType`
  Narrows the search using a specific field type. The field type must be provided as a string value.
- `AlsoDeletedFields`
  Expands search to fields that are been considered deleted.
- `OnlyDeletedFields`
  Search will only search in the deleted content.
- `ParentPath`
  Narrows the search results only to the pages that have the specified page as a parent. The page path must be provided as string.
- `PageTitle`
  Searches in the application using the page title, in all languages if the LanguageId is not specified. This value must be provided as a string.
- `PageDescription`
  Searches in the application using the page description, in all languages if the LanguageId is not specified. This value must be provided as a string.
- `PageMenuTitle`
  Searches in the application using the page menu title, in all languages if the LanguageId is not specified. This value must be provided as a string.

- `LanguageId`
  Defines that the search will act in a specific language. If the language is specified then also the non-localized values are been searched.
- `AnyText`
  Searches any text value

Note: When searching, if the fields `Deleted, AlsoDeletedFields` or `OnlyDeletedFields` are not defined, the core searches only in the content that is not deleted in any way.

The ordering of the resulting data can be also extensively defined:

- `ByName`
  Orders the result by page name
- `ByPublishDate`
  Orders the results by publish date
- `ByExpireDate`
  Orders the results by expiration date
- `ByCreatedDate`
  Orders the results by creation date
- `ByModifiedDate`
  Orders the results by the date modified
- `ByCreatedBy`
  Orders the results by the username that created the pages
- `ByModifiedBy`
  Orders the results by the username that created the pages
- `ByIntVal`
  Orders the results by the attributes' integer values
- `ByFloatVal`
  Orders the results by the attributes' float values
- `ByDateTimeVal`
  Orders the results by the attributes' date values
- `ByStringVal`
  Orders the results by the attributes' small string values
- `ByTextVal`
  Orders the results by the attributes' text values
- `ByFieldName`
  Orders the results by the field names (do not have any result on the page searching – this is included for future extension)
- `ByAttributeName`
  Orders the results by attribute names (do not have any result on the page searching – this is included for future extension)
- `ByLanguageID`
  Orders the results by the available translation languages (using LCIDs)
- `ByTitle`
  Orders the results by the pages' phrase titles
- `ByDescription`
  Orders the results by the pages' descriptions
- `ByMenuTitle`
  Orders the results by the pages' menu titles

Additionally in the ordering of each field you may define if it will be ascending or descending order.

Resource Optimization

The API defines functions that allow the developer to use javascript and css in a centralized way, which later will be optimized and merged as a single resource response.

The optimizations on the javascript and css resources are the following:
- Merge – All resources are been merged to a single http call
- Minimization – All resources are been stripped from the excess characters
- Compression – The resulting responses are been compressed using gzip or deflate, depending on the browser support
- Caching – The response is cached for next requests
- Relative paths are been fixed (css only)

The resources can be extracted by string or files using the following 4 methods:
- `void RegisterScriptToCompress(string id, string file)`
- `void RegisterScriptTextToCompress(Type type, string id, string text)`
- `void RegisterCSSToCompress(string id, string file)`
- `void RegisterCSSTextToCompress(Type type, string id, string text)`

The optimization is only supported on platform elements:
- TemplatePage
- FieldUserControl
- MasterPageTemplate
- SimpleUserControl
- AdministrationUserControl
- EditorUserControl

[Resource Manipulation (File themes)]

[Future Placeholder]

Security

Security in the platform is implemented by utilizing the IIdentity and IPrincipal interfaces. The main classes that are provided by the security subsystem are `ScriptServerIdentity` and `ScriptServerPrincipal`.

Identity

The identity class provides all the means for user management. The developer can use the API to do stuff like:
- Manage other users
- Change user data
- Impersonate
- Login as NT user
- Login using Open ID
- Track actions

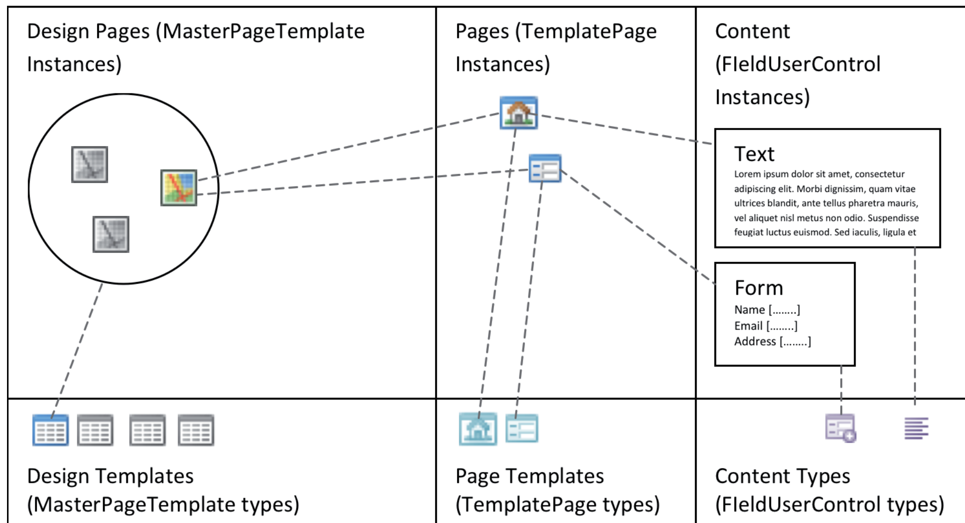
There are some security rules that apply when you manage security. The user must have "`User Management and Security User Right`" or "`Administration User Right`" to be able to change other user's data or manage them. Because many times you need automated tasks to be able to create or manage users in an unknown user context we are using impersonation for this purpose. The programming flow for developing in unknown user context is the following:

- Impersonate the SystemAccount super user (is an internal administrator)
- Do the actions that need elevated privileges
- Stop impersonation.

[Principal]

Overview

The 3 base elements for the web extensibility are the FieldUserControl, TemplatePage and MasterPageTemplate. They are related as been shown in the following figure.



Each page has a unique instance of a master page, so each TemplatePage instance has also one MasterPageTemplate instance attached to it. The connection that pages and master pages have is been defined as [RelationAttribute] in the TemplatePage implementation.

The templates and types that are been developed are been known to the platform as "Resources" (See "`ScriptServer.Web.UI.Resources`" class). The developers provides the functionalities that can be used by the system by the filesystem (*.master for master page templates, *.aspx for page templates and *.ascx for content types).

The page classes that are been provided and installed to the system are been instantiated from the platform web internals, and then execute the request. All the relevant information that is attached in a template (like design page and content types) are been connected at runtime. Specific content types allow the automatic creation of UserControl instances that are taking part on a page (like FieldHost.ascx content type).

Resources

The resource part of the web platform allows the developer to deal with the late-bound types that ASP.NET provides. The problem is when someone adds a file in the filesystem, the ASP.NET core must first compile it and then load it into process memory. The compilation occurs only when the part is been requested in any way. The Resource class allows you to load or install parts of the site that has not been loaded until the current point. If the part is not loaded but installed then the Resource class will find it, load it and return an instance of the resource type.

The same is true for editor controls that are been used across the system. You can use the abilities of the Resource class to manually add or remove editors.

The resource static class contains also methods that will allow installation of packages (zip resources) that contain resource files, and other files. Then it will automatically install all the files that can recognize, and the system will be able to use them immediately.

SystemResourceAttribute Attribute
The resource object in the system that is marked using this attribute is considered a system part, and will not be shown or manipulated from the user interface. This allows safe installation of resources like administration interfaces, that they already use the platform, but the user should not be able to use them, like for example the "/admin" section template.

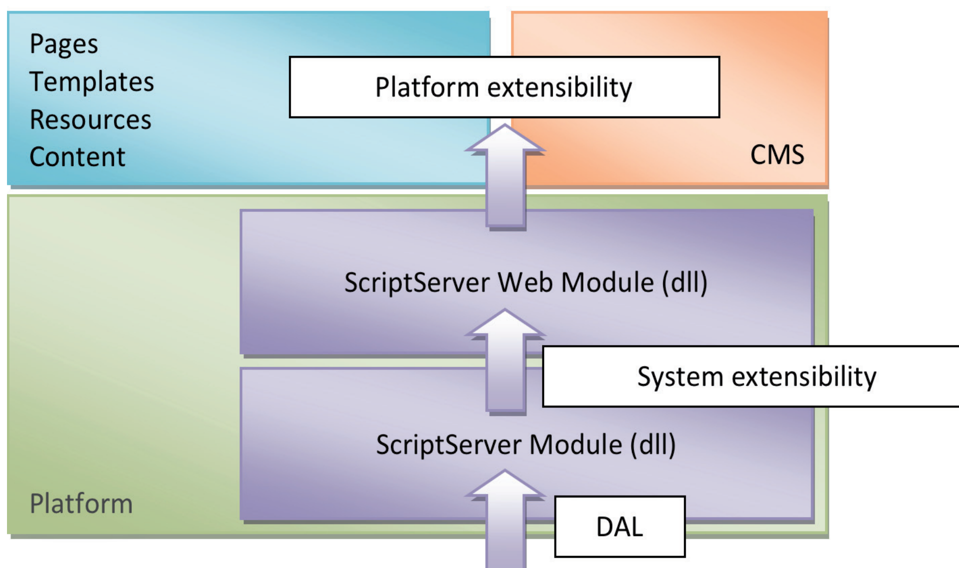
This attribute can be used in TemplatePage, MasterPageTemplate and FieldUserControl.

When a FieldUserControl has the SystemResourceAttribute has also the `IsUnique` flag to true, then the system marks the object as Page Feature. Pages features can be manipulated by the user (from a special UI), so they must have the necessary phrase and image attributes.

Deep System Extensibility

Introduction
Inside the platform actually 2 modules reside. The immediate accessible from the web environment is the "ScriptServer.Web" which provides the object model and features for the web management system.

The other module is "ScriptServer" (ScriptServer.dll) and provides the application with the object model and the data access layer.



How are objects organized in the platform
So how the data access layer (DAL) is look like for the ScriptServer platform? This subsection analyses how the object model in the DAL looks like, as well as their relation.

Platform data entities
The platform is considered to have the following data entities:
- Objects
- Fields
- Attributes

Objects
Objects are the biggest entity in the system. An object can be uniquely defined by its path; a URI path, that acts as a URL in the web application (but this is clearly on who is controlling this information).

Objects contain attributes. Because attributes are used for programming purpose, it can only be defined and used by the template that defines their schema (structure). Attributes cannot be change at runtime except if the component is been reinstalled.

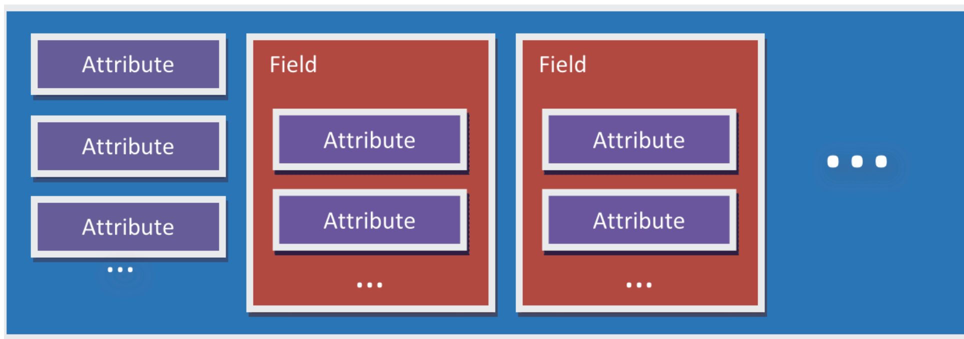
Objects also contain fields. Fields can be dynamically attached / created /deleted to / from objects.

Fields
Fields are considered the content provider for the platform.

Fields can have attributes that can be further marked with the following flags
- localizable
- meta

Localizable field attributes have different value per language, while meta field attributes are not be used when indexing the resources for keywords.

Structure schema
The structure schema can be described by the following figure:



Pages and Objects
The page model is tightly connected to the objects. Whenever we are talking about page in platform, then the same things happen exactly for its uniquely connected object.

Editing a page
When we edit a page, we are editing the object. So we are editing each attribute and field included in the object. In addition on that, editing each field, we are editing its attributes also. For that purpose, there are no unique editors for fields and pages. The editors are been applied in attributes.

Entities
[This module is in pre-release state and is subject to change in future releases]

Entities are the actual contracts and logic that drive the synchronization from the strongly typed object mode (pages / templates) to the database and back. They are very important because they allow the developer to further extend the functionality of the platform in any class and hierarchy, with the advantage that the platform can still save and load the instances without any more information.

The interfaces and classes that are used for that to be accomplished are the following:
- IObjectBind
- IFieldBind
- ObjectsCoordinator
- FieldsCoordinator

IObjectBind
The IObjectBind interface implements a connection to an SSObject instance in the database.
It allows an object instance to be retrieved using its path, name and parent and other patterns.
It allows getting strongly type information for fields and other object structure.

IFieldBind
The IFieldBind, in a common way as the IObjectBind, allows a type to use SSField data entity.
It uniquely connects the specific derived class (e.g. Control) to a specific field object that is connected to an SSObject.

ObjectsCoordinator
Object coordinator offers the services that an IObjectBind implementation needs to be orchestrated from the instance to database and back. The class services are using the IObjectBind to get a map on an instance, and the instances use ObjectsCoordinator to further orchestrate their actions. It is a repetitive procedure that allows the model to extend as far as a developer wants.

In addition to the common tasks of mapping on instances, managing relations and other information types, the ObjectCoordinator class allows the installation of types that are going to be used from the platform core (In the web core this is further abstracted because of the use of late-bound resources).

FieldsCoordinator
The FieldsCoordinator has the same purpose, but servicing only objects that implement IFieldBind interface. It provides the same functionality, for saving and loading object maps, and installing uninstalling to the ScriptServer core (ScriptServer Schema).

Types that support IObjectBind
The types that support the IObjectBind interface are:
- `ScriptServer.Extension.Model.SystemCatalog` in ScriptServer.dll
  Represents immutable catalogs inside the application to keep a folder-like structure to objects (force structure more accurately).
- `ScriptServer.Extension.Model.Template` in ScriptServer.dll
  Represents an installed template in the system. The templates can be related between them for creation allowance, or default template under a folder.
- `ScriptServer.Web.UI.TemplatePage` in ScriptServer.Web.dll
  The base class that all templates derive on *.aspx files.
- `ScriptServer.Web.UI.MasterPageTemplate` in ScriptServer.Web.dll
  The base class *.master page templates (design templates) derive in the system.
- `ScriptServer.Extension.Model.Catalog` in ScriptServer.dll
  Represents a catalog inside the application to keep a folder-like structure to objects. Those catalogs can be deleted.

Types that support IFieldBind
The types that support the IFieldBind interface are:
- `ScriptServer.Web.UI.FieldUserControl` in ScriptServer.Web.dll

The base class that all content types derive.

Component Extensibility

Components are a powerful way to extend the application layer with new features by reducing dependencies.

Introduction

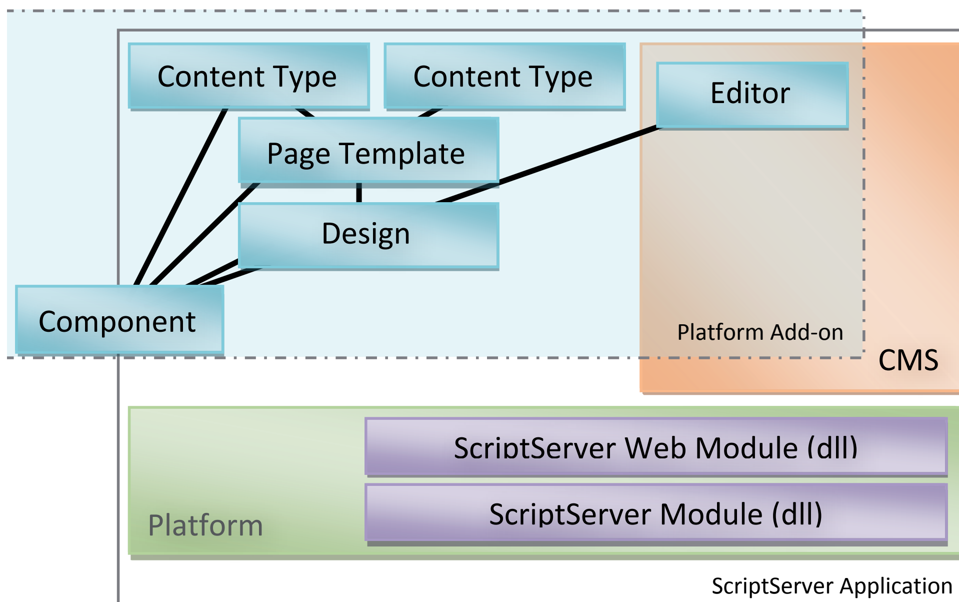
The extensibility sections before that was been referring to a normal object oriented model that can be expanded and be recreated on the runtime using the data in the database. While it allows great expandability, it does however not break the barriers of the "componentization" of the platform.

This chapter is introducing the ScriptServer components, where allows you to interact with unknown modules inside the platform using URI requests in MVC pattern. The component is considered the controller of the pattern.

Component System

The system uses a componentized subsystem to allow dynamic addition of modules without changes in the existing modules.

The following figure shows how the component is participating in the platform and add-on structure.



Definition of a component

The component must be considered as a service-provider object in regards to the object model. When a request exists the component is queried and if supports the current request, then it executes it and responds back to the CCL. Component classes must be provided as dlls or be implemented in the App_Code ASP.NET subfolder.

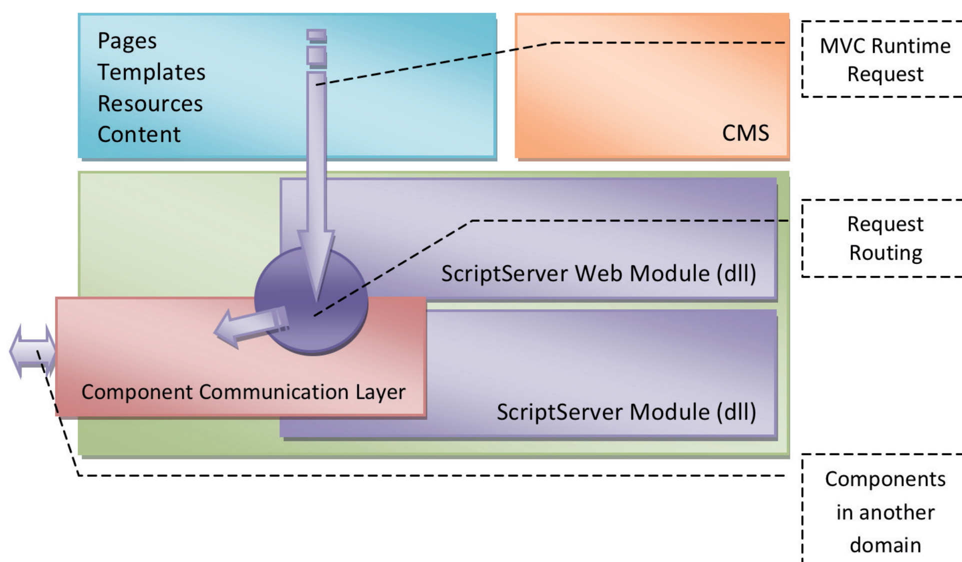
The advantage of using a dll is that the CCL core will load the components in an out-of-process model (with the CCL windows service), which means in different process that will communicate directly with the application.

Because components are service-providers they must be multi-threading safe. They must be designed in a way that will allow concurrent use of resources and/or data.

Programmatically, component is a class that provides a number of methods. Those methods can be discovered using Reflection from CCL and when a requests needs to be resolved, they are queried of their capabilities depending on their URI and return type. The type can be static or not. If it is not static, an instance is created with the default constructor. If the default constructor is not available, then the component is not loaded, and discarded in any future requests. Note that an instance is only created if a successful match is found for the specific type/method pair.

Communication Layer (CCL)
The CCL is responsible for transferring requests and selecting the target method that will actually resolve it.

An example of the component request-response model is as following.



- A request is been created from a source. The source can be UI, component, workflow or any part that has access to the platform core (only the ScriptServer.dll is enough). The request is been done with a URI (e.g. "send-email://an.email@somewhere.com"). Data can also be sent together that might be useful for the component that will service the request.
- The request is been delivered to the CCL (Component Communication Layer) that is held responsible to route the request to the correct destination (could be in different process or protocol and environment).
- The request is been handled manipulating the local data (Model), returning the data of the response. If no matching components found, the requests returns nothing to the requestor.
- If an error happens during this process, the CCL will return it as an object if supported, or throw the exception
- The requestor (View) redirects or reveals information that has to do with the current action.

In this process the request does not know the handler that provides the service. This is been discovered or registered from CCL, which is responsible of discovering and managing the references to those objects.

Discovery and Registration
The discovery of the components is been done through reflection. The data are been cached in the CCL core for further inquires. Programmatically, someone can add / remove registration entries that route / deny the requests to assemblies and/or classes.

In addition the discovery is been done only in the current process space. All public types of all assemblies are been searched for components.

Component classes and functionalities
The component subsystem can be categorized as follows:
- Classes that the component caller uses to call existing functionality
- Component base classes, that are used from component developer
- CCL (Component Communication Layer)

User Classes
To retrieve data from the internals of the CCL the UI developer can call

```
T Resolve<T>(string uri, params object[] data)
```

or

```
object Resolve(Type type, string uri, params object[] data)
```

if the context that is working does not allow generic types (e.g. COM objects).

The most important element is the pair of [return type, uri]. It is used to determine if a registered method supports the request or not. Then the path is matched with a developer provided pattern (as a regular expression) fill out with any of the arguments sent with invocation.

The data parameter allows the developer to add name / value lists of options that might be consumed from the method (e.g. IDictionary or NameValueCollection). This allows sending requests that include form or header data along with a request. For example:

```
CCL.Resolve<bool>("send-email://an.email@somewhere.com",
Request.Form, Request.Headers);
```

Also object properties and/or fields can be consumed by the core using reflection.

In the above example a possible candidate would be:

```
[Path(@"^send-email://([a-z0-9\.]+@[a-z0-9\.]+\.[a-z0-9]{2,3})$", 0)]
public bool GetData(System.String email);
```

(See below for the customization of `PathAttribute` class)
Note that the path expression allows to the core to embed the email to the uri.

In the same way the core allows to add options in any list using the parameter name. An overriding string can be set on the parameter level to change the field that is searched (e.g. `[ArgName(@"Accept-Language")])`.

Component base classes
The component class that is to participate in dynamic invocation from the CCL must have the `[Component]` attribute. Note that if you declare a component on runtime, this does not have to have this attribute. Further masking must be applied on the method level, where the actual invocation will take place.

The attribute that specify that a method is available for invocation is `PathAttribute`.
It accepts one of the following inputs:

- Path template that allows core to specify where every argument is placed
  This usage is the simplest one. You just need to specify every variable that is going to be filled up with uri data, using the predetermined variable definitions. For example "/products/@name/@int/@boolean". This expands to a regular expression that will try to match any requests.
- Regular expression text and regular expression indexes
  This usage allows you total control over the path input. When the attribute is used in that way, then it needs to be specified the matching indexes that will be matched on the respective parameters. For example adding 1,2,0 after the regular expression string, the [1] (second) match will be added to the 1st parameter, the [2] (third) match on the 2nd parameter and the [0] (first) match will added to the 3rd parameter.

When resolving with additional data structures then the core matches the name of name/value pair with the name of the argument, and if the type can be supported, then is successfully converted to the argument value that will participate in the invocation.

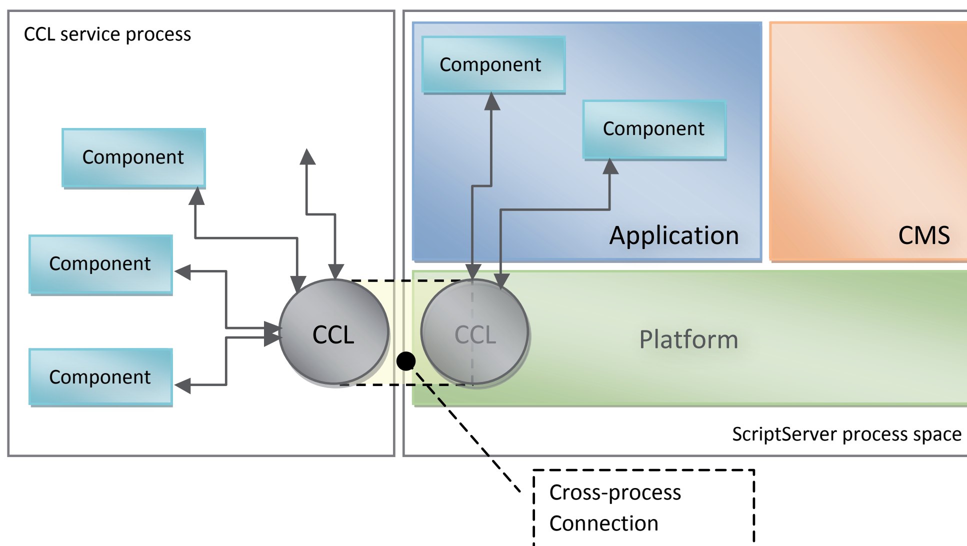
There are occasions where the input dictionary or object property to have different name with the variable we would like to match. Like the above example someone could try to match a Request.Headers variable. The variables in this name/value collection do not have an argument-friendly name. This is where the `ArgNameAttribute` attribute comes to fill that gap. So to accept an "Accept-Language" variable from the collection we need a `[ArgName(@"Accept-Language")]` definition. You can supply overriding argument names at any time using CCL.

CCL API
CCL static class allows you to resolve uris, add or remove at runtime types and methods to be invoked (they do not have to be masked with any attribute). Also it allows you to create ignore rules for uris that must not been served from CCL, or from specific classes.

CCL core implements service interfaces that allow direct access into the system from other technologies. The technologies that are supported are:

- Web Services *[Not supported yet]*
- Remoting *[Not supported yet]*
- WCF Services *[Not supported yet]*
- COM interoperability

CCL architecture is very flexible, and is designed for scaling. It is described in the following figure.



Simplest CCL setup is to work as a proxy to an external windows service that makes the actual communication with components. CCL classes allow the developer to provide more routes for component discovery and invocation.

Components can be discovered anywhere in the CCL process. Because the core elements are in the platform core, the CCL functionalities can be used also from the web application, but the components in the CCL process space won't be able to use them.

Note
I f you use components in the user interface, have in mind that they will be delay-loaded. Only when the UI is been requested the units are compiled and loaded, and therefore available for interaction. This means that the UI components will miss the first application-wide component registration sweep, so they have to be registered manually (programmatically).

Even if the Components are not been discovered when loaded for first time, the CCL API allows programmer to register type/method pairs for request listening with a custom Uri. You can also at runtime provide change the argument naming scheme for each argument that are persisting across calls.

CCL Core functionalities

Included with the CCL core are multiple CCL implementations that allows extensibility and smooth integration with the core. The following categories important:

- Installation
- Search
- Image thumbnail creator

Installation

[Future Placeholder]

- Security
- Solutions

Search

Search is provided using CCL categorized results through a number of providers. When extending the search system you need to implement 2 component methods:

```
[Path(@"search-categories")]
SearchData MyCategory()

[Path(@"search://<unique category here>/@integer/@any")]
[Path(@"search://([0-9]+)/([^\/]*)", new int[] { 0, 1 })]
public List<SearchData> MyActualSearch(int maxCount, string
searchString)
```

Then your search is been integrated directly to the search system, which includes the UI and the admin management.

Image thumbnails

Image thumbnails are not an extensible CCL interface. It provides you with the ability to create thumbnail images by only using links. It follows the pattern:

http://<your site>/<width>x<height>/<real image path>

For example:

http://localhost:4806/Site/100x100/UserFiles/eCommerce/ItemImages/1000-1.gif

The application will automatically cache the image for further use.

Packages and Licensing

Packages

Introduction

The software components that have been created to wrap specific functions must be deployed in the client server with as much less interaction and access of the user / administrator side. The platform provides a deployment scheme that includes both resources and components, automating the installation process by uploading only a single file.

Resource Packages

Resource packages are zip files that contain in a file structure all the relevant code that a user needs to add to his site, to enable developed features. The resources can be also programmatically installed by using the Resource class. Any folder or zip file can be designated as target, and the system successfully adds / installs any known resource and/or components.

Add-ons

Add-ons are resource packages that are accompanied from a description xml file. The xml file includes information about the resource package (e.g. developed by, prerequisites and more). Both of those files can then be sent to ScriptServer to take a license number that components and/or UI can use.

ScriptServer Services

ScriptServer Services is the software that coordinates the applications in every aspect. All the packages and licenses are been served by this application.
ScriptServer Services exists at:

- http://78.111.168.107/

In the future more domains and IPs will be accumulated for the services application.

When you access the ScriptServer Services you may:
- Change your user info
- Change partner info
- Change your customers' info
- Access ScriptServer Marketplace and purchase new modules
- *Upload new packages*

Zipping files
When the package is ready to be uploaded, the files need to be zipped in the application folder. The folders that need to be zipped must be without including the application path.

The folders will be extracted in the application, and each platform component will be installed in the system. Even if the folder structure exists in the application the system will place the files exactly in the same position. For optimization purposes, only the folders that are provided in the package will be indexed for installation.

Uploading a package
[This section might be subject for future changes]

You need the following data to successfully start a uploading your package:
- The zip folder with the application files
- A small package image
- A big package image
- A whitepaper pdf file
- A user guide pdf file
- Package Name (this will be unique for every package)
- Package marketing description
  This description will be a general description for the product. It can be the same across versions.
- Package version description
  Description of changes and other information regarding the current version.
- Version information
- Packages that are prerequisites / dependencies
- Package category in the marketplace
- If your package is private (a private package cannot be sold from the marketplace)
- If the package is updatable from the current version
- Price
- Annual fee

You must have all the above cleared up to upload the software for your package.

Feedback / Checking
When you upload the package, then the system will automatically check the application resources for errors and validity of the content. The services indexing will also test the provided package with all the prerequisites.

Note: When an application part fails to be registered, then it will not work, because the license mechanism cannot validate it.

Licensing in ScriptServer

Introduction

Licensing services are provided by the core to ensure that rights are been preserved across all ScriptServer installations. There two models used depending if a solution is in developing stage or released. When the application is been developed it is considered that it is in "Developer/ Partner licensing model", while when it is released to a customer is considered as "customer licensing model".

Developer/Partner license

This is the model that is used that allows a developer or a support member to add and test functionalities that have not been yet licensed. The license that supports that is called "developer license".

In this mode the system will make, in specific time intervals, callbacks in the ScriptServer web services system that will validate the license and the installation.

Customer license

This model allows the system only to use resources that are defined in the license that is provided to the customer. Any new functionality that is installed in the system must be properly licensed before it is start been used. The licensing system will connect to ScriptServer services each time an add-on / package is installed and download the renewed license that the user had purchased.

Workflow for licensing components

The common workflow that allows the automatic delivery of components is the following:

1. [Developer] The company entity that develops uses their own developer license that will allow them to create and test freely new developed features.
2. [Developer] After development has reached release state, then the company entity uploads the code for validation on ScriptServer organization (through ScriptServer.Services).
3. [ScriptServer] ScriptServer organization accepts and verifies the package. Then the application is been parsed by the automated tools for errors and information.
4. [Customer] When the information has been parsed from the ScriptServer automated systems, then the customer can buy the component from ScriptServer Marketplace and add it to his license.
5. [Application / System] The application then checks the licensing server and verifies that the selected module is been purchased, and updates the software as necessary.

Development Licensing

A development license must be requested when starting customization procedures. Because the customization needs to be a license free procedure, the application must have access to ScriptServer web services site, which will validate the development environment.

Licensing restrictions

The customer's license is restricted. The customer cannot start or manage portions that are not licensed. All the licensing management is been handled automatically from the application and ScriptServer licensing web services using the customer's data and unique information that identifies each application.

If the licensing subsystem fails to reach the servers, there is a risk that the software locks itself after trying too many times. A locked / disabled application involves the denial of the following or more of ScriptServer services:

- Update / Rollback
- Support
- Licensing procedures initiation on the ScriptServer licensing services
- Error reporting

The system is also locked if the license expires.

When the system is considered locked, the administrators cannot login and the elevated privileges of edit / create are been halted for the duration of the lock. Any read operation is normally processed, and system does not block the existing content to be delivered (so the site does not go down).

After a specific interval counted on days the licensing core will close any access to the underlying system, even read, until the license will manually be updated.

Appendix 1 – Integration Core

The integration core is shipped with the ScriptServer tools platform parts. It includes all the necessary components to customize a server using Windows Communication Foundation (WCF) or implement a new client. It includes all the reusable components and structures that are exposed to different technologies (like COM and WCF) for communicating two different end-points.

The main structure that defines also how the integration core works, is been defined as `ScriptServer.LocalERP.Integration.Data.Action`. The actions are the main interaction between the client and the server. It is the actual directive on what the client asks from the server in a simple and effective way.

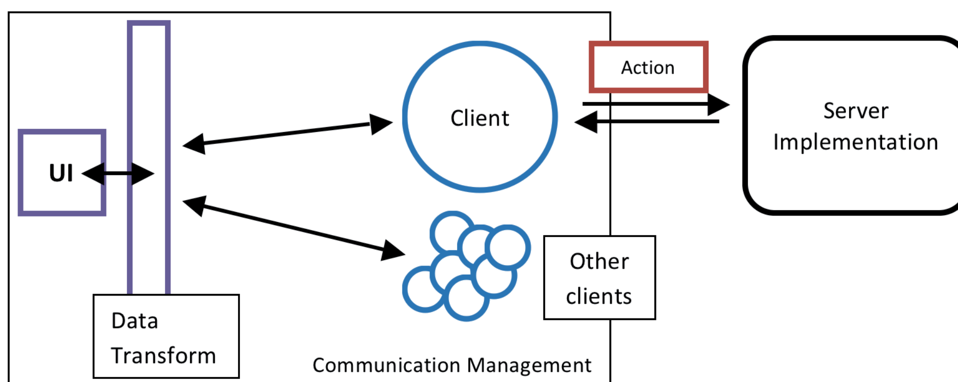
Terms

The integration core can be connected to different backend systems. Each registration of a system is called a service. The application can have any time of services at any time.

The services furthermore have *connections* to the backend system. Each connection represents a different, parallel connection to the backend service (probably connecting to different server instance). The connections implemented by WCF use URIs to connect to the respective server.

Architecture

The information exchange architecture can be seen in the following figure.



The architecture is been designed in a way that:

- Abstracts information from the user interface
- Encodes / Decodes information for different services an connections
- Allows the user to configure what is been transferred and where
- Manages automatically the connections
- Provides an extensible client model

The main target is that the UI will always get a uniform set of data in a Name/Value perspective. This will always ensure that any connection or service can undertake any request, because the communication manager will format the data using the user's definitions.

Usually, if no specific service is been called by the programming interfaces, the communication core will try to send the request to all services. Then the same application layer will merge fields and other information in the same structure that will be provided to the UI, to seamlessly integrate services. Requests can be configured from the administration user interface to point to only one service or to map specific fields to specific services.

Connections on the other side cannot be controlled from the API. Connections supports a load balancing scheme and the application must always provide the same scheme for every unique connection in the same service. The integration core works with this assumption.

Client Model

To transfer the data a client is used that is bound to the protocol and the server that can connect to. There is one stock implementation in the transport core provided. This implementation is based on Windows Communication Foundation (WCF) using web services over http. However the client model is extensible for custom implementations. The LocalERP package for example ships with its own client implementation.

The client model allows the communication manager to manipulate many different connections in many disparate servers. This removes any user interface considerations that would make the software be developed towards a specific backend system. Additionally the core can provide very advanced management features, like load balancing and failover.

You can anytime create and manipulate a client connection using the API if you need to.

Every client is derived from the interface `ScriptServer.LocalERP.Integration.Communication.IClient`. One client instance will be created for each connection, and they may be used on different threads, so the resources must be synchronized properly.

WCF Server

The stock implementation of the server is provided in the integration core. The server has been built on .NET technology, and supports COM. A developer can use the server to build a bridge between xPoint and a custom backend system, and open information flow to the UI.

The server is implemented as the `ScriptServer.LocalERP.Integration.Communication.Wcf.WcfServer` class. It provides two error events. The "OnError" error handler in this level is been fired when the application has a serious unrecoverable error. The "OnMessageError" is been fired when the application has an error on the OnGetMessage handler of an attached endpoint. Then the error is somewhere in the implementation of the logic that the specific action triggers.

Server is actually a closure for multiple URI servers, which are been called *endpoints*. Whenever the developer needs to add a server listener, he must instantiate and adds this on the server endpoint collection.

Endpoint

Every endpoint is a server instance listening in a URI using a protocol. When an endpoint is attached to a server, it can start processing requests in parallel with other endpoints. When a message arrives the "OnGetMessage" event is been activated to start handling the request. The endpoint also defines the required API to send error and notification messages with the application response. Those messages are been then decoded by the communication management on the xPoint.

Actions

The request data that are been transferred across the services are been encoded as Actions `(ScriptServer.LocalERP.Integration.Data.Action class.` The action class keeps all the necessary information that the server system might need o use. Because of the various information that this layer supports the system is able to provide advanced services from the server to client like the following:

- Error and logging information
- Action name, parameters, results; the normal request data
- Message statistics, timeouts information
- Fields selection
- Sorting selection
- Paging
- Availability of features (e.g. supports paging?)
- Requests with no response

xPoint integration core

Communication Manager

The communication manager is the last layer before the user interface for any application that will connect to the backend systems. This is the management layer for opened connections and advanced features that automates the evolution of the system across time.

It abstracts the need to directly call a specific client by using types and action names. The action name specifies what services and fields will be requested depending on the configuration file. The default configuration file that is been used by the system exists on "<path>\Components\eCommerce Platform\Files\LocalERP.Base.Configuration.xml".

Object Map

The object map is the internal view of the communication core. It is been used to store services, types, communication extensions, field transformations and more.

The configuration is an xml file that is been saved in application global settings. It includes the following sections:

- Registrations
- Services
- Entities

Registrations

In this section the base communication types are been registered in the system. The administrator can select then the communication component that must be used to connect to a server implementation. The communication core includes an implementation of WCF web services.

Services

All the relevant information from services is been stored under this xml node.

Entities

In this section all the type mappings are been registered. When a pair of type and actionname is been requested, the core resolves the field names, services and other request details using the elements under this node. This allows requesting customizations on field level for different ERPs. The entities structures are like the following example:

```
<Entity type="ScriptServer.LocalERP.Data.eCommerce.ErpItem">
   <Field erp-name="No." field="No" service="any">
       <Feature name="navision-calculate" />
       <Relation erp-relation-table="Item Variant"
                    erp-relation-field="Item No."
                    local-field="Variants">
             <Field ... />
   <Memory-Caching timeout-seconds="7" />
   <Valid-actions>
       <Action name="GETITEM" />
```

1. The entity node specifies the type that can be returned

2. A field definition is a mapping definition from the backend ERP to the type

3. The 'Feature' node provides additional directions to the backend systems on how to proceed with the field value retrieval. For example the "navision-calculate" forces a calculation to a field before the value can be retrieved. One field may have multiple features. No error is generated if a feature is not supported.

4. Relation node can be declared when a search must be performed after the value evaluation on the field. The backend system, if supports that feature, it searches in another record collection entity with the current value and returns the specified matches as a collection on the requested type.

5. Memory caching suggests the default memory caching of the full evaluated structure object (it may be constructed from multiple requests).

6. Valid actions node makes the specific type request to be triggered only over a specific action. This allows having multiple requests for different actions, to fine tune the application performance.

The communication manager (ScriptServer.LocalERP.Communication.CommunicationManager) is based on the configuration and includes the following sections:

- Sending / Requesting information
- Create / Update /Delete
- Low level connection management
- Configuration customization

Sending / Requesting information

Send methods are resolving the requests by using the type-name that needs to be returned with the action type. Requests are been resolved using the xml configuration. Internally the application checks for the fields that need to be requested, as well as the services that they are requested from. If any field is been set to a specific service then the request will be done only on those services.

The type can be anything; the communication core provides a base implementation of the commonly used classes. Additionally the application allows adding a list of a type like the following:

```
CommunicationManager.Send<ErpItem[]>("GETITEM", "*");
```

This will tell the communication manager to bring more than 1 item. When the application "expects" to get one item by using the statement

```
CommunicationManager.Send<ErpItem>("GETITEM", "*");
```

then the paging mechanism is been automatically triggered to limit the items to 1. This is been transported to the backend systems, where if they support paging, they enable it for the request. The overall bonus of the application is that it has the best performance in all levels because only the required elements are been processed.

Variations

Send method is also provided with the following variations:

- Paging
- Sorting
- Service selection
  - All services
  - Selection of services

The paging and sorting is been done inside the backend system if supported.

Activation

The services that can be activated from the configuration and the available fields can be retrieved by the API:

```
List<string> GetActivatedServices<T>(string actionName)
List<string> GetFieldsForType<T>()
```

The main advantage is that the above methods of the API are not invoking the server and will perform better on the UI.

To retrieve the fields that the application UI will *actually* get resolving the server request use the method:

```
List<string> GetErpFields<T>(string actionName, params object[]
data)
```

The above method will make the request on the server and will retrieve the backend ERP names of the fields that will be retrieved.

Support

Support from the available backed systems on the different features can be retrieved from the method:

```
Action GetMinimumFeatures(string[] services, string[] actionNames)
```

were returns an Action object with the "Support" properties filled up. It returns the minimum features of the requested services and it will return no support if at least one service do not support a feature.

Create / Update / Delete

The integration core allows updating, creating and/or deleting backend entries from the xPoint, when of course the backend connection supports it.

All of the methods have the same target functionality: First a collection of objects is formed or retrieved from the backend system, then is been manipulated and then used for one of the above methods. The order of the data sent is irrelevant. The backend system uses the primary keys of the record to find the correct entry, and changes/creates/deletes the correct one.

The following example changes the address2 field on the backend system:

```
ErpCustomer customer =
CommunicationManager.Send<ErpCustomer>("GETCUSTOMER");
customer.Address2 = "set from the web - 2";
CommunicationManager.Update<ErpCustomer>(
        "GETCUSTOMER",
        customer,
        new ErpField[] { new ErpField("Address2") }
        );
```

The "address2" field will be mapped accordingly on the correct field when sent from the core.

Caching and Timeout

The integration core includes a two-layer based caching. One is on the connection level, and one on the object resolution. This allows the administrators to fine tune the application depending on the backend systems.

Additionally timeout can be applied to resolve bottlenecks easily using timeouts. Then the resource that gave the error can be redirected using failover mechanism to another resource (local storage for example).

ScriptServer – 15 years in business
– dedicated to meet our customers challenges of tomorrow

ScriptServer has 15 years of expertise in developing, selling and supporting web based software in standard packages.

New platform: xPoint.NET – e-commerce, CMS and intranet all in one

Since 2009 ScriptServer has developed a new platform: xPoint. The platform contains all products in one platform and is based on today most advanced it-technology. Packages and components can be purchased in the web based ScriptServer Marketplace.

The ScriptServer organization

Our office in Denmark is the HQ for the ScriptServer activities. We have the responsibility for development, sales, support, production and administration.

ScriptServer Solutions A/S
Herstedvang 7C
2620 Albertslund
Denmark
Phone +45 33 26 31 00
support@scriptserver.com
www.scriptserver.com

ISV/Software Solutions
Custom Development Solutions