The Component Library

Version 0.91

© 2014 CrownPeak Technology, Inc. All rights reserved. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission from CrownPeak Technology.

Document History

Author/Editor	Date	Reason for Change	Version
Fahd Shaaban	8/28/2014	Draft	0.90
Fahd Shaaban	10/23/2014	Class Versioning	0.91

Table of Contents

The Component Library **1**

Document History **2**

The Component Library Overview **4**

Benefits of the Component Library 5

Business Process guides the configuration of the Component Library **6**

Considerations 6

Specifications 6

Components 6

Components Methods 7

Templates 9

Playbook Notes 11

The Component Library Overview

The CrownPeak Component Library provides a process to define re-usable template components that provide an approved and repeatable "pattern" with the corresponding logic necessary to provide a complete Authoring Experience (content capture and content validation) within the CMS.

The CrownPeak CMS provides a number of object types, including Templates, Models, and Assets.

Templates, which are made up of individual files, provide the ability to configure an *authoring experience* allowing authors/editors to enter content, adhering to specific validation rules, and merging the captured content within an approved presentation (HTML/CSS/JS) to produce ***pages***.

These ***pages*** are called *assets* in the CMS, and each has a corresponding Asset ID.

The CMS also provides the ability to embed the rendered output of any asset within another, as well as the ability to expose the content of any asset to be queried by, and used, within another. These *content provider* assets are called ***widgets***. Widgets are normal assets, but the way they're used as described above, is what makes them widgets.

Re-using content in the CMS is achieved with widgets. However, re-using code requires an alternate approach.

The CMS provides template developers the ability to re-use code by encapsulating the repeatable code patterns in Library Methods (or Functions) defined a ***Class***. These custom library methods are available to all templates, and provide a way to centralize the definition of the code and to re-use the code as many times as required.

The Component Library is best suited when the HTML/CSS is structured and re-usable, where components represent Front-End Patterns that can be assembled into Templates and ultimately Pages. CrownPeak Partners and Agencies often provide their own Pattern Libraries. In addition, there are also a number of open source and free Pattern Libraries available for download and use.

The Component Library supports Patterns that utilize other Patterns, which means that Components can be nested within other Components. The following is a typical hierarchical structure to organize your Components:

- Atoms or Elements: Smallest pattern type.
- Molecules or Modules: A collection of elements
- Organisms or Blocks: A collection of modules and elements

- Layouts – A container that provides a way to organize the blocks, modules and elements used on a page

Benefits of the Component Library

The Component Library provides the following benefits:

- The ability to define Template Components and use them to build templates quickly and easily.
- The ability to re-use the components multiple times within a single template and within other templates.
- The ability to limit the review and approval of template changes to isolated components. When a component is approved, the change can be deployed to all affected templates.
- By separating the presentation of each pattern into separate Component Assets, this will allow the look and feel of Templates to be branched, and previewed in separate publishing environments.
- The Component Library provides a workflow and process to define Patterns of approved “markup”, or HTML/CSS/JS, without requiring any knowledge of the CrownPeak C# API.
- The Component Library imposes a consistent process to define a collection of Library methods corresponding to each Pattern.
    - Input Methods: These methods are used to add input controls to capture content, and are called from the Input file of a Template
    - Post Input Methods: These methods are used to validate or manipulate the captured content, and are called from the Post Input file of a Template
    - Output Methods: These methods are used to render the content within an approved presentation, and are called from the Output file of a Template.

Business Process guides the configuration of the Component Library

The Component Library can be configured to support many business processes. Before starting the configuration, it is beneficial to solidify the scope of the configuration. Use this list to determine how the customer is going to use the Component Library in their instance.

- Does the customer want to leverage an HTML/CSS library of patterns?
- Does the customer want to re-use capabilities and site components?
- Does the customer want to streamline the process of making changes to common or shared components?

Considerations

- Changing a component used by multiple templates will impact all templates.
- Changing the associated Component Logic (to accommodate new fields, additional validation rules, etc) has to be done within the methods in the Custom Library. Modifying methods can be accomplished by adding conditional statements to encapsulate the changes at the Class Level and associate the various Class Versions with any criteria, such as a specific publishing state. Versioning of Classes is described in detail below.
- An initial configuration effort is required to setup all the necessary components based on the provided Pattern Library. Templates can then be configured relatively quickly leveraging the configured, tested, and approved components.
- The CMS will support multiple Component Libraries.
- Component Libraries can be defined at any level: Instance, Site Collection, Site, Site Section, etc.
- The Component Library approach only impacts how Templates are configured. There is no impact at all on the management of Assets, Workflow, Publishing configurations, and ACLs.

Specifications

Components

Components are created using a Component Model based on a Component Template that prompts for the markup (html/css/js) of an approved Library Pattern.

The markup of the Pattern is copied from the Pattern Library and pasted into a new Component in the CMS, in the Markup content field.

Unlike Patterns, which focus primarily and solely on the final rendering or output of the content, Components extend the Pattern Definition, and provide the necessary CMS rules for a corresponding Authoring Experience to capture the content, and validate it.

How to build a Component

Each component requires at three corresponding Library Methods

Assuming an initial pattern as follows

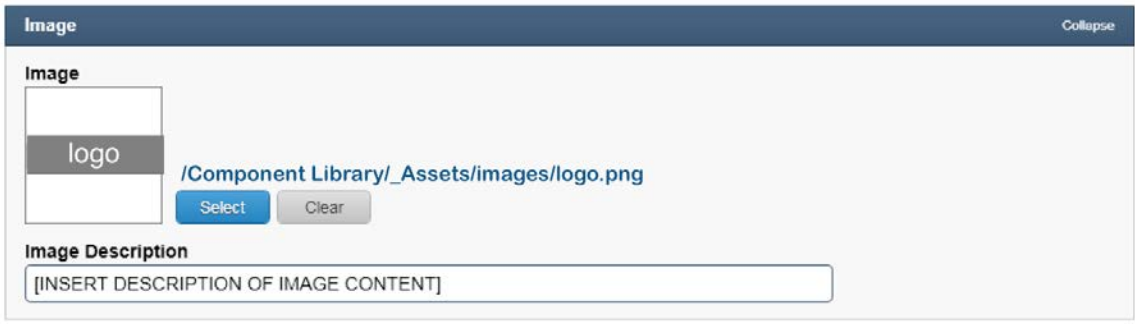
Pattern : `<img class="logo" src="http://www.domain.com/image.png" alt="description">`

The implemented pattern will need to be modified to the following, and implemented as a Component Asset in the CMS

Component : `<img class="logo" src="{src}" alt="{alt}">`

Logo Component Methods:

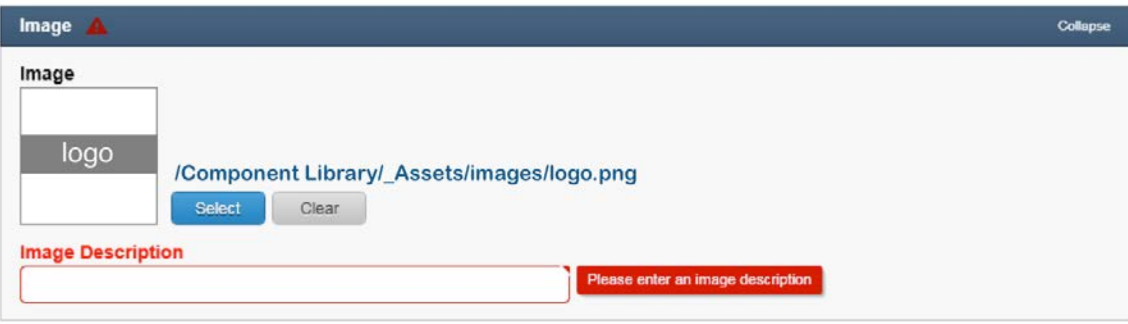
INPUT




```
public static void logo_input(String label,String name)
{
        Input.StartControlPanel(label);
                // input fields, or calls to other input methods go here
                ShowAcquireParams img= new ShowAcquireParams();
                img.ShowBrowse = true;
```

```
        img.ShowUpload = true;
        img.Extensions = Util.MakeList("jpg", "jpeg", "gif", "png");
        Input.ShowAcquireImage(label, name + "_src", img);
        Input.ShowTextBox(label + " Description", name + "_alt");
    Input.EndControlPanel();
}
```

POSTINPUT




```
public static void logo_postinput(PostInputContext context, String name)
{
    // validation rules go here
    if (String.IsNullOrEmpty(context.InputForm[name + "_alt"]))
    {
        context.ValidationErrorFields.Add(name + "_alt", "required");
    }
}
```

OUTPUT

```
public static String logo(Asset asset, String name)
{
    Asset component = Asset.Load(componentsPath()+"/atoms/logo");
    StringBuilder sbContent = new StringBuilder();
```

```
        sbContent.Append(component.Raw["markup"]);
          //variable substitution occurs here
          sbContent.Replace("{src}", asset[name + "_src"]);
          sbContent.Replace("{alt}", asset[name + "_alt"]);
          return sbContent.ToString();
}
```

Templates

The input template is constructed by calling the INPUT method of all the components (atoms, molecules, organisms) that are required for a specific Page Type.

The postinput template is constructed by calling the POSTINPUT method of all the components (atoms, molecules, organisms).

The output template is constructed by calling the OUTPUT method of each component by adding the output of each component to the Column the component belongs to based on selected Layout.

Class Versioning

For each library file that is needed to be versioned a Base Abstract Class needs to be created. This Base Abstract Class will define all the methods available as virtual methods. If a new version needs to be created; A new class can be created that inherits from this Base Class. If this class is empty, it will use all the methods defined in the Base Class. Developers can than create override methods in the new Versioned Class, and those methods with be used.

The switchboard technique is used in the Base Class to return the appropriate version of a Class based on any criteria, such as workflow state.

In the template files, an initial call can be added to the ReturnClass method (passing in the asset or a

criteria, such as the workflow state) and it will return the Versioned Class associated in the switchboard. The major benefit in this is since the method returns the Abstract Class' contract and the Versioned Class, the code in the input and output templates does not have to be changed or updated to test these method changes.

The Switchboard

```
public abstract class ComponentLibrary
{
    public static ComponentLibrary ReturnClass(Asset asset)
    {
        String version = getVersion(asset);
        try
        {
            switch (version)
            {
                case "1":
                    return new CrownPeak.CMSAPI.CustomLibrary.ComponentLibrary_V1();
                case "2":
                    return new CrownPeak.CMSAPI.CustomLibrary.ComponentLibrary_V2();
                default:
                    return new CrownPeak.CMSAPI.CustomLibrary.ComponentLibrary_V1();
            }
        }
        catch (Exception ex)
        {
            return new CrownPeak.CMSAPI.CustomLibrary.ComponentLibrary_V1();
        }
    }

    public static string getVersion(Asset asset)
    {
        Asset clConfig = Asset.Load(componentsPath(asset) + "library config");
        switch (asset.WorkflowStatus.Name.ToString())
        {
    case "Live":
                return clConfig.Raw["live_version"];
            break;
    case "Stage":
                return clConfig.Raw["stage_version"];
```

```
                    break;
          case "Draft":
                              return clConfig.Raw["draft_version"];
                    break;
          default:
                              return "1";
                    break;
                    }
          }
}
```

Notes and Additional Information

More information can be found in

https://connect.crownpeak.com/products/product_features/marketplace_connectors_and_dashboard_widgets

Playbook Notes

https://connect.crownpeak.com