# Attaching Custom Elements to an Existing Tree Structure

2017-02-14

Orckestra, Europe
Nygårdsvej 16
DK-2100 Copenhagen
Phone +45 3915 7600

www.orckestra.com

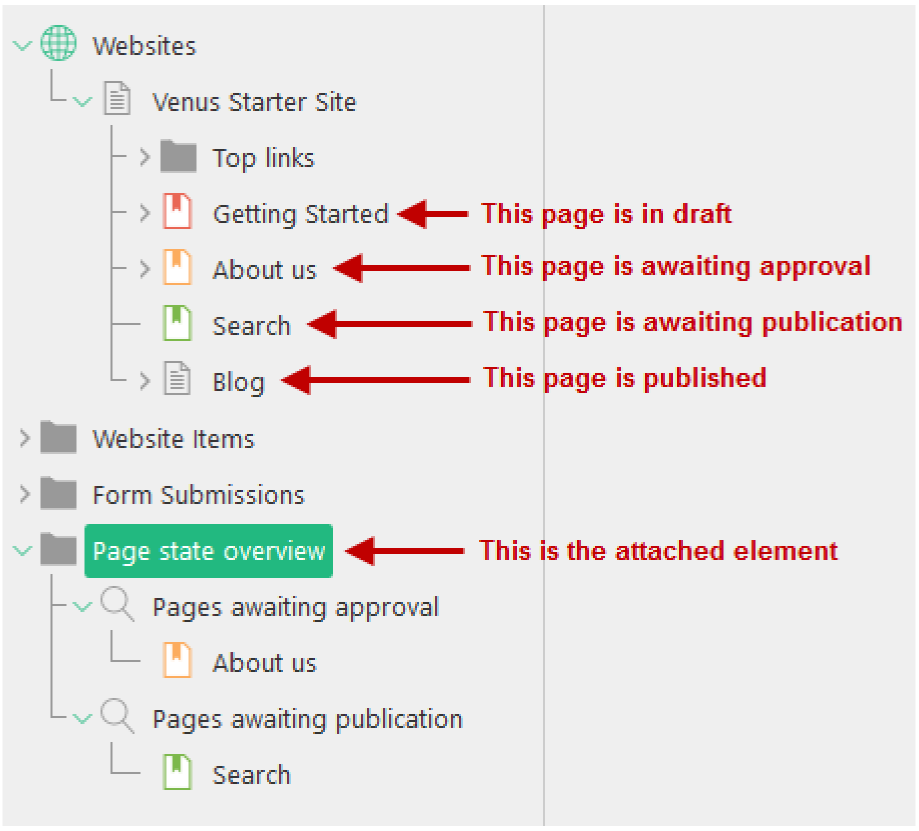
## Contents

## 1 Overview

The goal of this tutorial is to explain how to attach elements to an existing tree, namely, in the Content perspective. This tutorial will show how to add your own elements and how to add elements representing data as well as add workflow actions and add actions that open a custom URL.

**Tree**

Here is how our tree will look like after we complete this tutorial:



## 2 Step 1: Getting Security Hierarchy in Place

First we need to get the security hierarchy in place. Every element in the tree has an entity token and this entity token is used for determining security for the element. When C1 CMS does security check for a given element, it looks for permissions set on the given element and all its ancestor entity tokens. So it is important that there exists a path from our element entity token to the root of the tree. So we start out by creating an entity token class for our root element:

```
[SecurityAncestorProvider(typeof(NoAncestorSecurityAncestorProvider))]
public sealed class MyRootEntityToken : EntityToken
{
  public override string Type
  {
    get { return ""; }
  }

  public override string Source
  {
    get { return ""; }
  }

  public override string Id
  {
    get { return "MyRootEntityToken"; }
  }

  public override string Serialize()
  {
    return DoSerialize();
  }

  public static EntityToken Deserialize(string serializedEntityToken)
  {
    return new MyRootEntityToken();
  }
}
```

The NoAncestorSecurityAncestorProvider value of the SecurityAncestorProvider attribute specifies that elements having the entity token does not have a native parent when determining security. Later in this tutorial we will add a hook from our root element to the Content perspective element, so that it will naturally inherit the security settings from the Content perspective – unless it's overwritten by a user locally on our root element.

## 3 Step 2: Implementing Security for New Root Element

Now we need to fix the security for our new root element. Because of the NoAncestorSecurityAncestorProvider value we need to make a hook from the parent element of our root element to our root element. This takes two steps.

First, we should implement the IAuxiliarySecurityAncestorProvider interface.

```
public sealed class MyRootEntityTokenAuxiliarySecurityAncestorProvider :
  IAuxiliarySecurityAncestorProvider
{
  public Dictionary<EntityToken, IEnumerable<EntityToken>>
    GetParents(IEnumerable<EntityToken> entityTokens)
  {
    Dictionary<EntityToken, IEnumerable<EntityToken>> result =
      new Dictionary<EntityToken, IEnumerable<EntityToken>>();

    foreach (EntityToken entityToken in entityTokens)
    {
      if (entityToken.GetType() == typeof(MyRootEntityToken))
      {
        // Here we specify that the Content perspective element is the
parent of our root element
        result.Add(entityToken, new EntityToken[]
        {
          AttachingPoint.ContentPerspective.EntityToken
        });
      }
    }

    return result;
  }
}
```

When C1 CMS runs its security algorithm, it will call SecurityAncestorProvider to get these hooks. In the code above we can see that we return a hook from our root entity token to the Content perspective's entity token.

Next, we need to register our new MyRootEntityTokenAuxiliarySecurityAncestorProvider class. This can be easily done with an ApplicationStartup handler like this:

```
[ApplicationStartup]
public static class MyApplicationStartup
{
  public static void OnBeforeInitialize()
  {
  }

  public static void OnInitialized()
  {
AuxiliarySecurityAncestorFacade.AddAuxiliaryAncestorProvider<MyRootEntityTo
ken>(new MyRootEntityTokenAuxiliarySecurityAncestorProvider());
  }
}
```

The ApplicationStartup attribute will make C1 CMS call two methods - OnBeforeInitialize and OnInitialized - on its startup. We want to register our new AuxiliarySecurityAncestorProvider after C1 CMS has been initialized. So we put the registering code inside the OnInitialized method.

## 4 Step 3: Making Attaching Element Provider

Now we are ready to make our attaching element provider that will show our new root element.

```
[ConfigurationElementType(typeof(NonConfigurableElementAttachingProvider))]
public sealed class MyElementAttachingProvider : IElementAttachingProvider
{
  public ElementProviderContext Context
  {
    get;
    set;
  }

  public bool HaveCustomChildElements(EntityToken parentEntityToken,
                      Dictionary<string, string> piggybag)
  {
    if (ElementAttachingPointFacade.IsAttachingPoint(parentEntityToken,
      AttachingPoint.ContentPerspective) == false) return false;

    return true;
  }

  public ElementAttachingProviderResult GetAlternateElementList(
     EntityToken parentEntityToken,
     Dictionary<string, string> piggybag)
  {
    if (ElementAttachingPointFacade.IsAttachingPoint(parentEntityToken,
      AttachingPoint.ContentPerspective) == false) return null;

    ElementAttachingProviderResult result = new
ElementAttachingProviderResult()
    {
      Elements = GetRootElements(piggybag),
      Position = ElementAttachingProviderPosition.Bottom,
      PositionPriority = 0
    };

    return result;
  }

  private IEnumerable<Element> GetRootElements(Dictionary<string, string>
piggybag)
  {
    yield return new Element(this.Context.CreateElementHandle(new
MyRootEntityToken()))
    {
      VisualData = new ElementVisualizedData
      {
        Label = "Page state overview",
        ToolTip = "Page state overview",
        HasChildren = true,
        Icon = CommonElementIcons.Folder
      }
    };
  }

  public IEnumerable<Element> GetChildren(EntityToken parentEntityToken,
                       Dictionary<string, string> piggybag)
  {
     throw new NotImplementedException("Will be later in the tutorial");
  }
}
```

First, the ConfigurationElementType attribute is used to specify if this plug-in has its own configuration in the Composite.config file or – as in our case - has no configuration at all. You'll read more on the configuration later.

**ElementProviderContext**

This is set by C1 CMS and is used for creating ElementHandle's, which - combined with an entity token - make the identity of the element. For now you only need to save this in the setter and use it later on when creating new elements.

**HaveCustomChildElements**

This method is used to tell C1 CMS that an existing element in the tree - the one you want your element(s) to be attached under has children (your elements).

**GetAlternateElementList**

This is the actual method that attaches your elements to the existing tree. This is done through the ElementAttachingProviderResult class where you can specify if your elements should be put at the top or at the bottom and a priority that is used if more than one ElementAttachingProvider is attaching elements to the same one as you do. This method calls the GetRootElements described below.

**GetRootElements**

This is where the actual magic happens. The most important thing here is this line:

```
return new Element(this.Context.CreateElementHandle(new
MyRootEntityToken()))
```

Here you can see that the Context is used for creating the element handle and we use our root entity token MyRootEntityToken.

**GetChildren**

At this point we don't have any children to our root element, so this is left empty.

## 5 Step 4: Adding New ElementAttachingProvider to Composite.config

Now we should add our new ElementAttachingProvider to Composite.config and see the result.

Here is a section of the Composite.config file.

```
<Composite.C1Console.Elements.Plugins.ElementAttachingProviderConfiguration
>
  <ElementAttachingProviderPlugins>
  <!--  Existing providers -->
  <add name="MyElementAttachingProvider"
     type="AttachingCustomElementsSample.MyElementAttachingProvider,
AttachingCustomElementsSample" />
  </ElementAttachingProviderPlugins>
</Composite.C1Console.Elements.Plugins.ElementAttachingProviderConfiguratio
n>
```

We have added our new ElementAttachingProvider with this line:

```
<add name="MyElementAttachingProvider"
      type="AttachingCustomElementsSample.MyElementAttachingProvider,
AttachingCustomElementsSample" />
```

And when C1 CMS is restarted, our fine new root element should be shown in the Content perspective. As mentioned earlier, our ElementAttachingProvider doesn’t have any custom configuration. If it had, that would have been added as child elements to the <add> element in the listing above. Note that because our root element reports that it has children and our GetChildren method throws a NotImplementedException, then trying to open the root element in the tree will result in an error.

## 6 Step 5 - Adding Children to Root Element

When adding children to our root element, like with our root element, we need to have security in place. This time we will use native parents instead of the hooking method. We start out by creating a new entity token type:

```
[SecurityAncestorProvider(typeof(MySecurityAncestorProvider))]
public sealed class MyEntityToken : EntityToken
{
  public MyEntityToken(string publicationStatus)
  {
    this.PublicationStatus = publicationStatus;
  }

  public string PublicationStatus
  {
    get;
    set;
  }

  public override string Type
  {
    get { return ""; }
  }

  public override string Source
  {
    get { return ""; }
  }

  public override string Id
  {
    get { return this.PublicationStatus; }
  }

  public override string Serialize()
  {
    return DoSerialize();
  }

  public static EntityToken Deserialize(string serializedEntityToken)
  {
    string type, source, id;

    DoDeserialize(serializedEntityToken, out type, out source, out id);

    return new MyEntityToken(id);
  }
}
```

This class looks almost like our MyRootEntityToken class with two differences, though. The first one is that we have changed the NoAncestorSecurityAncestorProvider to our own SecurityAncestorProvider: MySecurityAncestorProvider. The second one is that this entity token holds a value: publicationStatus. We will use this value later on. It is important to note that some extra care in de-serialization has to be taken.

## 7 Step 6 - Implementing MySecurityAncestorProvider Class

Next we need to implement our MySecurityAncestorProvider class like this:

```
public sealed class MySecurityAncestorProvider : ISecurityAncestorProvider
{
  public IEnumerable<EntityToken> GetParents(EntityToken entityToken)
  {
    yield return new MyRootEntityToken();
  }
}
```

Like with our AuxiliarySecurityAncestorProvider implementation, C1 CMS uses MySecurityAncestorProvider when handling security. And in our example here, any instances of MyEntityToken will have a MyRootEntityToken as a native parent.

## 8 Step 7 – Implementing MyElementAttachingProvider Class

The last step of adding some children to our root element is done in the MyElementAttachingProvider class. Here is the new version:

```
[ConfigurationElementType(typeof(NonConfigurableElementAttachingProvider))]
public sealed class MyElementAttachingProvider : IElementAttachingProvider
{
  // ...the code skipped for readability - see Step 3

  public IEnumerable<Element> GetChildren(EntityToken parentEntityToken,
                              Dictionary<string, string> piggybag)
  {
    using (DataConnection connection = new DataConnection())
    {
      if ((parentEntityToken is MyRootEntityToken) == true)
      {
        yield return new Element(this.Context.CreateElementHandle(
          new MyEntityToken("Approval")))
        {
          VisualData = new ElementVisualizedData
          {
            Label = "Pages awaiting approval",
            ToolTip = "Pages awaiting approval",
            HasChildren = connection.Get<IPage>().
                  Where(f => f.PublicationStatus == "awaitingApproval").
                  Any(),
            Icon = CommonElementIcons.Search
          }
        };

        yield return new Element(this.Context.CreateElementHandle(
          new MyEntityToken("Publication")))
        {
          VisualData = new ElementVisualizedData
          {
            Label = "Pages awaiting publication",
            ToolTip = "Pages awaiting publication",
            HasChildren = connection.Get<IPage>().
                  Where(f => f.PublicationStatus == "awaitingPublication").
                  Any(),
            Icon = CommonElementIcons.Search
          }
        };
      }
      else
      {
        throw new NotImplementedException("Will be implemented later in the
tutorial");
      }
    }
  }
}
```

As all the new stuff is inside the GetChildren method, let's take a closer look at it.

Again, like with the root element, we use the Context to create the element handle, but this time it's with our new entity token type. We feed the constructor of this entity token with values of "Approval" and "Publication" respectively. We will use these values later on in the tutorial. Another thing to note here is that we dynamically set the HasChildren property depending on whether any pages are awaiting approval or publication respectively.

Now we have to add children to our root element with security and all in place. Next, we should add children to our children.

## 9 Step 8 - Adding Elements That Represent Data

This time we will add elements that represent data (pages), and in C1 CMS all data items have their own entity token (DataEntityToken), so we don't need an extra class for that. So the only thing we need to do in order to get security in place is to add hooks from our two child elements to the pages that are going to be children of those two elements. We do that with a new IAuxiliarySecurityAncestorProvider implementation

```
public sealed class MyDataEntityTokenAuxiliarySecurityAncestorProvider :
  IAuxiliarySecurityAncestorProvider
{
  public Dictionary<EntityToken, IEnumerable<EntityToken>> GetParents(
      IEnumerable<EntityToken> entityTokens)
  {
    Dictionary<EntityToken, IEnumerable<EntityToken>> result =
       new Dictionary<EntityToken, IEnumerable<EntityToken>>();

    foreach (EntityToken entityToken in entityTokens)
    {
      DataEntityToken dataEntityToken = entityToken as DataEntityToken;
      if (dataEntityToken.Data == null) continue;

      Type interfaceType = TypeManager.GetType(dataEntityToken.Type);
      if (interfaceType != typeof(IPage)) continue;

      IPage page = dataEntityToken.Data as IPage;

      if (page.PublicationStatus == "awaitingApproval")
      {
        result.Add(entityToken, new MyEntityToken[] { new
MyEntityToken("Approval") });
      }
      else if (page.PublicationStatus == "awaitingPublication")
      {
        result.Add(entityToken, new MyEntityToken[] { new
MyEntityToken("Publication") });
      }
    }

    return result;
  }
}
```

Some notes on this implementation. First, we do some checks to see that it is actually a DataEntityToken for a page that we have. Then we get the page and check the publication status for that given page. If it is awaiting approval we say that it has a hook from our child element with the entity token with the constructed value of "Approval" as mentioned in Step 7. The same goes for pages awaiting publication. If the page is in draft or published we do nothing, i.e. no hook exists. With this code we have told C1 CMS that all the pages in the "awaiting approval" state are a hook from our child element ("Pages awaiting approval") with the constructed value of "Approval". So are the pages awaiting publication.

## 10 Step 9 – Registering MyDataEntityTokenAuxiliary SecurityAncestorProvider

Like with our first IAuxiliarySecurityAncestorProvider, we need to register it. This we can do in our MyApplicationStartup class that now looks like this:

```
[ApplicationStartup]
public static class MyApplicationStartup
{
  public static void OnBeforeInitialize()
  {
  }

  public static void OnInitialized()
  {

AuxiliarySecurityAncestorFacade.AddAuxiliaryAncestorProvider<MyRootEntityTo
ken>(
       new MyRootEntityTokenAuxiliarySecurityAncestorProvider());


AuxiliarySecurityAncestorFacade.AddAuxiliaryAncestorProvider<DataEntityToke
n>(
       new MyDataEntityTokenAuxiliarySecurityAncestorProvider());
  }
}
```

We register our new MyDataEntityTokenAuxiliarySecurityAncestorProvider to be called for all entity tokens of the DataEntityToken type. This means that when C1 CMS has to security for data elements, then our GetParents method will be called, and that's why we test to see if it is a DataEntityToken for a page before proceeding.

## 11 Step 10 – Adding Children to Tree

And now, the fun part: adding the children to the tree. Here is the new version of our ElementAttachingProvider:

```
[ConfigurationElementType(typeof(NonConfigurableElementAttachingProvider))]
public sealed class MyElementAttachingProvider : IElementAttachingProvider
{
    private static ResourceHandle DataAwaitingApproval { get {
     return GetIconHandle("page-awaiting-approval"); } }
  private static ResourceHandle DataAwaitingPublication { get {
    return GetIconHandle("page-awaiting-publication"); } }

  // ...the code skipped for readability - see Steps 3 and 7

  public IEnumerable<Element> GetChildren(EntityToken parentEntityToken,
                          Dictionary<string, string> piggybag)
  {
    using (DataConnection connection = new DataConnection())
    {
      if ((parentEntityToken is MyRootEntityToken) == true)
      {
        // ...the code skipped for readability - see Step 3 and 7
      }
      else
      {
        MyEntityToken myEntityToken = parentEntityToken as MyEntityToken;

        if (myEntityToken == null) throw new NotImplementedException();

        IEnumerable<IPage> pages;
        ResourceHandle icon;
        if (myEntityToken.Id == "Approval")
        {
          pages = connection.Get<IPage>().
            Where(f => f.PublicationStatus == "awaitingApproval");
          icon = DataAwaitingApproval;
        }
        else if (myEntityToken.Id == "Publication")
        {
          pages = connection.Get<IPage>().
            Where(f => f.PublicationStatus == "awaitingPublication");
          icon = DataAwaitingPublication;
        }
        else
        {
          throw new NotImplementedException();
        }

        foreach (IPage page in pages)
        {
          Element element =
           new
Element(this.Context.CreateElementHandle(page.GetDataEntityToken()))
          {
            VisualData = new ElementVisualizedData
            {
              Label = page.Title,
              ToolTip = page.Title,
              HasChildren = false,
              Icon = icon
            }
          };
          yield return element;
        }
      }
    }
  }
  private static ResourceHandle GetIconHandle(string name)
  {
    return new ResourceHandle(BuildInIconProviderName.ProviderName, name);
  }

}
```

As mentioned in Step 5 (and Step 8) we have constructed our entity tokens for our children with some extra information. Now it's time to use this information.

By the construct, we know that the element "Pages awaiting approval" has an entity token with the Id value of "Approval". So when our GetChildren method has to find children for an element with an entity token of the MyEntityToken type, we can use the Id property value of this entity token to see what pages should be children. We also use this value to find the right icon. As listed in the code, if the Id property value is "Approval", we take all the pages in the "Awaiting approval" state – the same goes for "Publication".

Lastly, we create the actual elements. This is again done with the Context, but this time we get the entity token from the page (data item) itself.

Now we are done with the tree, which was one of the goals of this tutorial. Try to add some pages and change their publication status back and forth and note that the tree automatically refreshes. C1 CMS does this by default and uses the security relations to determine how to refresh the tree. If your elements don't show up, refresh them manually.

## 12 Step 11 – Adding URL Action to Pages under Child Elements

To add a URL action to the pages under our child elements, the first thing we need is to make an action token implementation that is going to represent our custom URL action.

```
[ActionExecutor(typeof(MyUrlActionExecutor))]
public sealed class MyUrlActionToken : ActionToken
{
  private static PermissionType[] _permissionTypes = new PermissionType[] {
PermissionType.Administrate };

  public override IEnumerable<PermissionType> PermissionTypes
  {
    get { return _permissionTypes; }
  }

  public override string Serialize()
  {
    return "MyUrlAction";
  }

  public static ActionToken Deserialize(string serializedData)
  {
    return new MyUrlActionToken();
  }
}
```

We use the ActionExecutor attribute to tell C1 CMS who is responsible for executing actions with this action token. So let’s go and implement that.

## 13 Step 12 – Implementing IActionExecutor Interface

The MyUrlActionExecutor implements the IActionExecutor interface.

```
public sealed class MyUrlActionExecutor : IActionExecutor
{
  public FlowToken Execute(
     EntityToken entityToken,
     ActionToken actionToken,
     FlowControllerServicesContainer flowControllerServicesContainer)
  {
    string currentConsoleId = flowControllerServicesContainer.
       GetService<IManagementConsoleMessageService>().CurrentConsoleId;

    DataEntityToken dataEntityToken = entityToken as DataEntityToken;
    IPage page = dataEntityToken.Data as IPage;

    string url = string.Format("/MyUrlAction.aspx?pageId={0}", page.Id);

    ConsoleMessageQueueFacade.Enqueue(new OpenViewMessageQueueItem
    {
       Url = url,
       ViewId = Guid.NewGuid().ToString(),
       ViewType = ViewType.Main,
       Label = "My URL Action"
    }, currentConsoleId);

    return null;
  }
}
```

And it only contains one method: Execute. We know that pages under our children are only going to have this action, so it’s pretty safe to get the page from the passed-on entity token. Then we enqueue a new message on the message queue and tell the client to open the given URL in a new document tab. We pass on the Id of the page to this aspx page in the query string.

## 14 Step 13 – Examples of MyUrlAction.aspx and MyUrlAction.aspx.cs

The following are simple examples of the MyUrlAction.aspx and MyUrlAction.aspx.cs.

One thing to note here is that some extra controls are used to get the C1 CMS looks on the page.

```
<%@ Page Language="C#" AutoEventWireup="true" Inherits="MyUrlAction"
CodeFile="MyUrlAction.aspx.cs" %>

<%@ Register TagPrefix="control" TagName="httpheaders"
Src="~/Composite/controls/HttpHeadersControl.ascx" %>
<%@ Register TagPrefix="control" TagName="scriptloader"
Src="~/Composite/controls/ScriptLoaderControl.ascx" %>
<%@ Register TagPrefix="control" TagName="styleloader"
Src="~/Composite/controls/StyleLoaderControl.ascx" %>

<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://www.w3.org/1999/xhtml"
xmlns:control="http://www.composite.net/ns/uicontrol">
  <control:httpheaders ID="Httpheaders1" runat="server" />
  <head>
    <title>My URL Action</title>
    <control:styleloader ID="Styleloader1" runat="server" />
    <control:scriptloader ID="Scriptloader1" type="sub" runat="server" />
  </head>
  <body>
    <ui:dialogpage label="My URL Action"
image="${skin}/dialogpages/message16.png">
      <ui:scrollbox>
        <asp:PlaceHolder ID="MyPlaceHolder" runat="server" />
      </ui:scrollbox>
    </ui:dialogpage>
  </body>
</html>
```

And now the MyUrlAction.aspx.cs file:

```
using System;
using System.Linq;
using System.Web.UI;
using Composite.Data;
using Composite.Data.Types;

public partial class MyUrlAction : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
      string pageIdString = this.Request.QueryString["pageId"];

      if (pageIdString == null) throw new InvalidOperationException();

      Guid pageId = new Guid(pageIdString);

      using (new DataScope(DataScopeIdentifier.Administrated,
UserSettings.ActiveLocaleCultureInfo))
      {
        IPage page = PageManager.GetPageById(pageId);

        this.MyPlaceHolder.Controls.Add(new LiteralControl("Title: " +
page.Title + "<br />"));
      }
    }
}
```

You can do all sorts of things here. This is only a very simple and naïve example.

## 15 Step 14 - Adding URL Action to Page Elements

Adding the URL action to the page elements is done in our MyElementAttachingProvider and it now looks like this:

```
[ConfigurationElementType(typeof(NonConfigurableElementAttachingProvider))]
public sealed class MyElementAttachingProvider : IElementAttachingProvider
{
    private static ResourceHandle DataAwaitingApproval { get {
     return GetIconHandle("page-awaiting-approval"); } }
  private static ResourceHandle DataAwaitingPublication { get {
     return GetIconHandle("page-awaiting-publication"); } }

  private static readonly ActionGroup ActionGroup =
    new ActionGroup("My tasks", ActionGroupPriority.PrimaryLow);

  // ...the code skipped for readability - see Step 3, 7 and 10

  public IEnumerable<Element> GetChildren(EntityToken parentEntityToken,
                          Dictionary<string, string> piggybag)
  {
    using (DataConnection connection = new DataConnection())
    {
      if ((parentEntityToken is MyRootEntityToken) == true)
      {
         // ...the code skipped for readability - see Step 3, 7 and 10
      }
      else
      {
         // ...the code skipped for readability - see Step 3, 7 and 10

        foreach (IPage page in pages)
        {
          Element element =
           new
Element(this.Context.CreateElementHandle(page.GetDataEntityToken()))
          {
            VisualData = new ElementVisualizedData
            {
              Label = page.Title,
              ToolTip = page.Title,
              HasChildren = false,
              Icon = icon
            }
          };

          element.AddAction(new ElementAction(new ActionHandle(new
MyUrlActionToken()))
          {
            VisualData = new ActionVisualizedData
            {
              Label = "My Url Action",
              ToolTip = "My Url Action",
              Icon = CommonCommandIcons.ShowReport,
              ActionLocation = new ActionLocation
              {
                 ActionType = ActionType.Other,
                 IsInFolder = false,
                 IsInToolbar = true,
                 ActionGroup = ActionGroup
              }
            }
          });

          yield return element;
        }
      }
    }
  }
  private static ResourceHandle GetIconHandle(string name)
  {
    return new ResourceHandle(BuildInIconProviderName.ProviderName, name);
  }

}
```

We use the Element.AddAction method to add our action and uses our MyUrlActionToken to specify that it is our URL action that we want to be executed. When you have done this and refreshes C1 CMS you should be able to see a new action when selecting a page in our attached tree. Note that this action is not on the pages in the original tree. When clicking on the new action a new tab window opens and the title of the page is shown.

# 16 Step 15 - Adding Workflow Action to Page Elements

Our final step, is adding a workflow action to the page elements.

The creation of workflows is beyond the scope of the tutorial, so we use reuse the existing edit page workflow.

```
[ConfigurationElementType(typeof(NonConfigurableElementAttachingProvider))]
public sealed class MyElementAttachingProvider : IElementAttachingProvider
{
  // ...the code skipped for readability - see Step 3, 7, 10, 14

  private static readonly ActionGroup ActionGroup =
    new ActionGroup("My tasks", ActionGroupPriority.PrimaryLow);

  private static ResourceHandle EditPage = GetIconHandle("page-edit-page");

  // ...the code skipped for readability - see Step 3, 7, 10, 14

  public IEnumerable<Element> GetChildren(EntityToken parentEntityToken,
                    Dictionary<string, string> piggybag)
  {
    using (DataConnection connection = new DataConnection())
    {
      if ((parentEntityToken is MyRootEntityToken) == true)
      {
         // ...the code skipped for readability - see Step 3, 7, 10, 14
      }
      else
      {
         // ...the code skipped for readability - see Step 3, 7, 10, 14

        foreach (IPage page in pages)
        {
          Element element =
           new
Element(this.Context.CreateElementHandle(page.GetDataEntityToken()))
          {
            VisualData = new ElementVisualizedData
            {
              Label = page.Title,
              ToolTip = page.Title,
              HasChildren = false,
              Icon = icon
            }
          };

         // ...the code skipped for readability - see Step 3, 7, 10, 14

           element.AddAction(
             new ElementAction(new ActionHandle(new WorkflowActionToken(
             WorkflowFacade.GetWorkflowType(
"Composite.StandardPlugins.Elements.ElementProviders.PageElementProvider.Ed
itPageWorkflow"),
             new PermissionType[] { PermissionType.Administrate })))
          {
            VisualData = new ActionVisualizedData
            {
              Label = "Edit page",
              ToolTip = "Eidt page",
              Icon = EditPage,
              Disabled = false,
              ActionLocation = new ActionLocation
              {
                ActionType = ActionType.Edit,
                IsInFolder = false,
                IsInToolbar = true,
                ActionGroup = ActionGroup
              }
            }
          });

          yield return element;
        }
      }
    }
  }
  // ...the code skipped for readability - see Step 3, 7, 10, 14
}
```

Adding a workflow action is similar to adding our URL action. It’s just done with a different action token type.