ZpqrtBnk
Umbraco

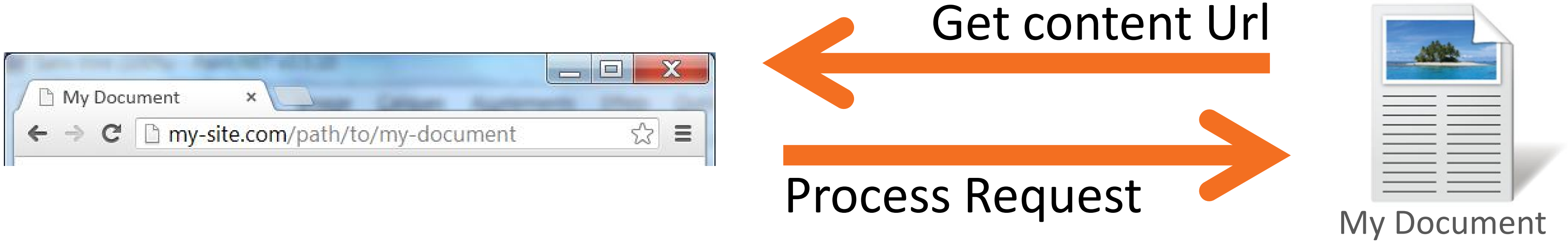
The Umbraco 6 Request Pipeline

Warning: this session may contain code

Stéphane Gay
2013/06/13

sgay@pilotine.com
@zpqrtbnk

“What’s the pipeline?”

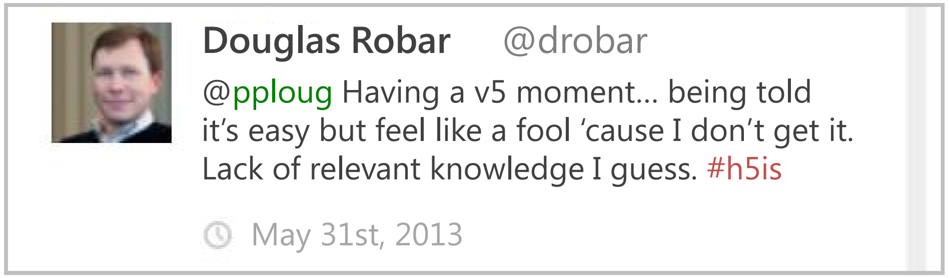


Get content Url

Process Request

> “Why the pileline?”

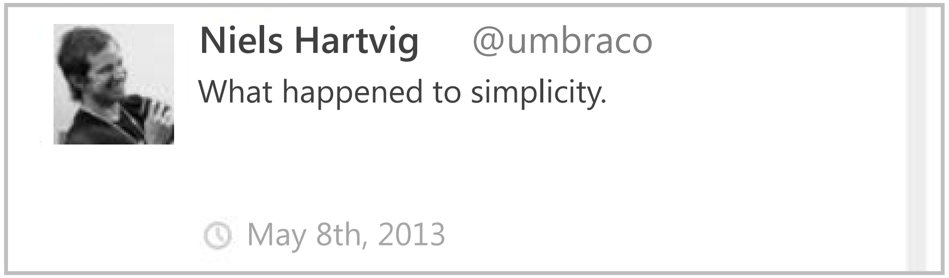
It Is Completely Broken™



Douglas Robar @drobar

@pploug Having a v5 moment… being told it's easy but feel like a fool 'cause I don't get it. Lack of relevant knowledge I guess. #h5is

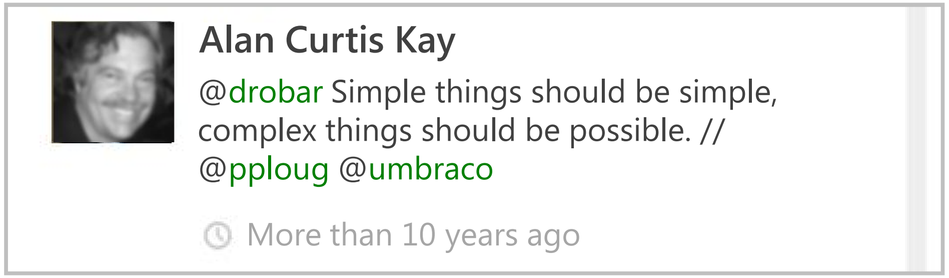
May 31st, 2013



Niels Hartvig @umbraco

What happened to simplicity.

May 8th, 2013



Alan Curtis Kay

@drobar Simple things should be simple, complex things should be possible. // @pploug @umbraco

More than 10 years ago

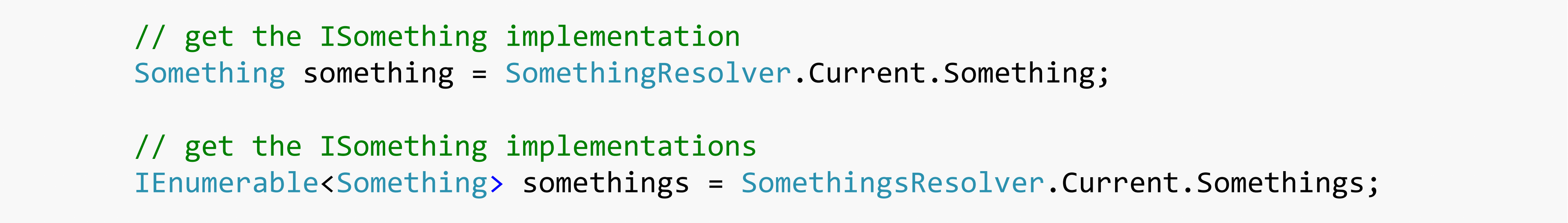
If you find yourself having a "v5 moment", please shout!
Unless it also happens to you all the time in your everyday life.

"Object Resolution"

```
// Provides something
interface ISomething { … }

// Implements ISomething one way
class Something : ISomething { … }

// Implements ISomething another way
class SomethingElse : ISomething { … }
```

Dependency Injection, IoC, containers… banned (v5 killing collateral damage).
Microsoft Extensibility Framework (MEF)… quite nice prototype, hard to explain.

Need something that can be explained with only a couple of slides!

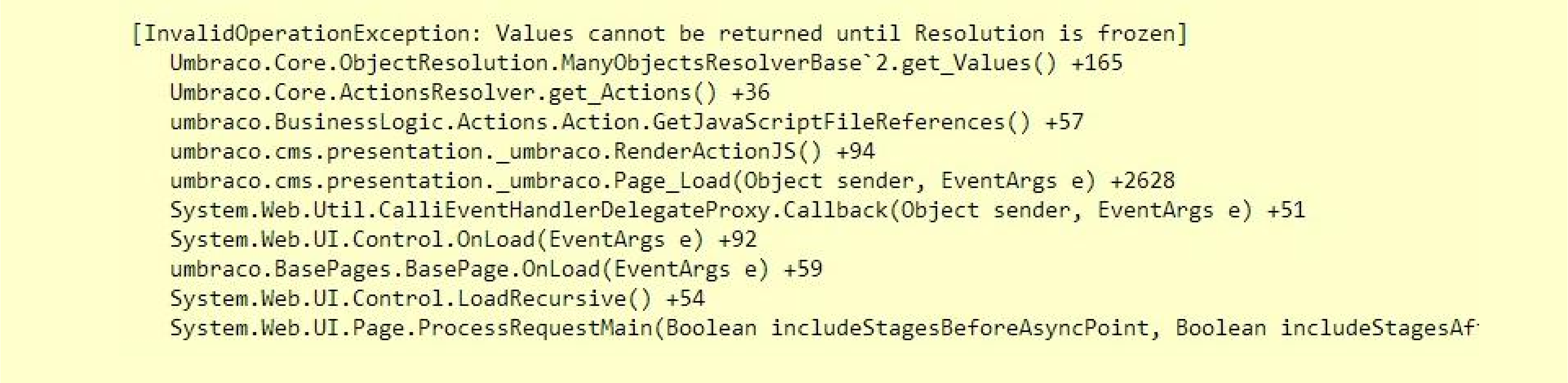


```
// get the ISomething implementation
Something something = SomethingResolver.Current.Something;

// get the ISomething implementations
IEnumerable<Something> somethings = SomethingsResolver.Current.Somethings;
```

Object Resolution,
Uses object resolvers to get the interface implementations.

```
// Initialize SomethingResolver
SomethingResolver.Current = new SomethingResolver(
    new Something());

// Initialize SomethingsResolver explicitely
SomethingsResolver.Current = new SomethingsResolver(
    typeof (Something),
    typeof (SomethingElse));

// Initialize SomethingsResolver by discovering types
SomethingsResolver.Current = new SomethingsResolver(
    PluginManager.Current.ResolveTypes<ISomething>());
```

Resolvers are explicitely initialized when the app starts.
Initialized once, then resolution is *frozen* and nothing can change anymore.
Otherwise, it throws!



```
[InvalidOperationException: Values cannot be returned until Resolution is frozen]
   Umbraco.Core.ObjectResolution.ManyObjectsResolverBase`2.get_Values() +165
   Umbraco.Core.ActionsResolver.get_Actions() +36
   umbraco.BusinessLogic.Actions.Action.GetJavaScriptFileReferences() +57
   umbraco.cms.presentation._umbraco.RenderActionJS() +94
   umbraco.cms.presentation._umbraco.Page_Load(Object sender, EventArgs e) +2628
   System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs e) +51
   System.Web.UI.Control.OnLoad(EventArgs e) +92
   umbraco.BasePages.BasePage.OnLoad(EventArgs e) +59
   System.Web.UI.Control.LoadRecursive() +54
   System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAf
```

Cannot resolve objects before resolution is frozen
Cannot configure resolution once it is frozen
Helps protect against invalid setup

“ Application Event Handler ”

```
public interface IApplicationEventHandler
{
    // ApplicationContext exists
    void OnApplicationInitialized(…);

    // Resolvers are initialized, resolution is not frozen
    void OnApplicationStarting(…);

    // Boot completed, resolution is frozen
    void OnApplicationStarted(…);
}
```

Umbraco will find and run every implementation of that interface.
Actually, there’s a resolver for that, too, albeit a special one.

```
public abstract class ApplicationEventHandler : IApplicationEventHandler
{
    // ApplicationContext exists
    protected virtual void ApplicationInitialized(…) { }

    // Resolvers are initialized, resolution is not frozen
    protected virtual void ApplicationStarting(…) { }

    // Boot completed, resolution is frozen
    protected virtual void ApplicationStarted(…) { }
}
```

Umbraco will find and run every implementation of IApplicationEventHandler
Better to inherit from the abstract class, and override only what’s needed.

```
public class MyApplication : ApplicationEventHandler
{
    protected override void ApplicationStarting(…)
    {
        SomethingResolver.Current.SetSomething(new SomethingBetter());

        SomethingsResolver.Current.RemoveType<Something>();

        SomethingsResolver.Current.AddType<SomethingBetter>();
    }
}
```

Drop that class anywhere in your code,
And you’re done configuring the resolver!

Arbejdsglæde

Belle ooohhhh la la

102.438

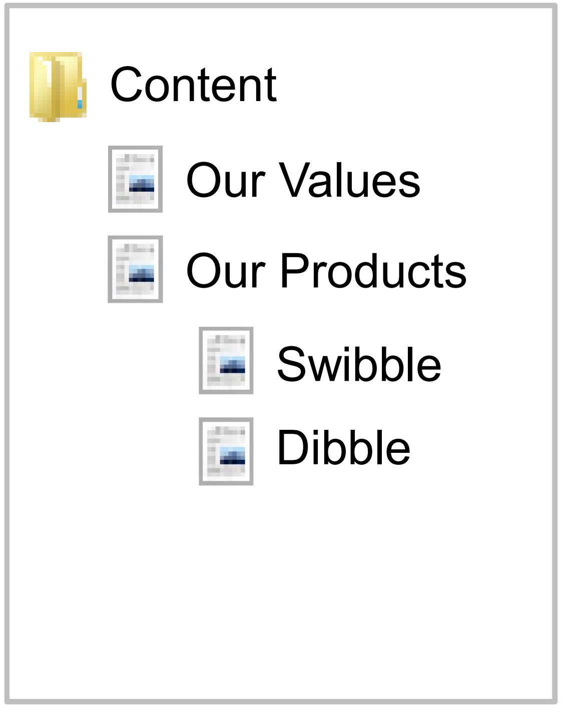
"Published Content"

```
public interface IPublishedContent
{
    int Id { get; }

    …

    IPublishedContent Parent { get; }
    IEnumerable<IPublishedContent> Children { get; }

    …
}
```

Represents a content (or a media) on the front-end.
Can be either published, or draft (for preview).

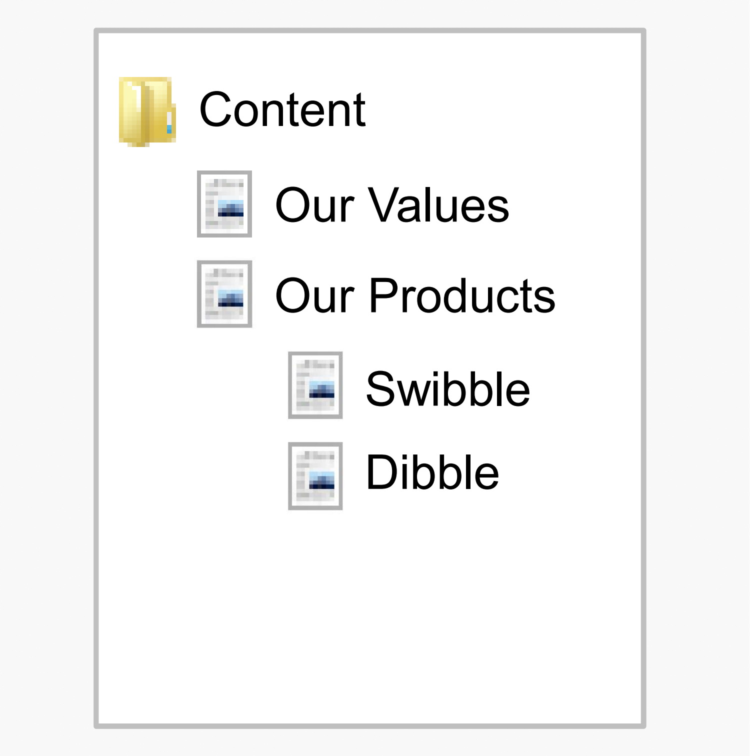
```
<div>
    <a href="@Model.Content.Url">@Model.Content.Name</a>
</div>
```

Available e.g. in MVC views model.
What is the value of Content.Url?



- Content
  - Our Values
  - Our Products
    - Swibble
    - Dibble

Swibble → "/our-products/swibble"
Or, maybe ".../swibble/" or ".../swibble.aspx"?
Absolute or relative? Which hostname? Http or Https?
That's the role of the *outbound* pipeline



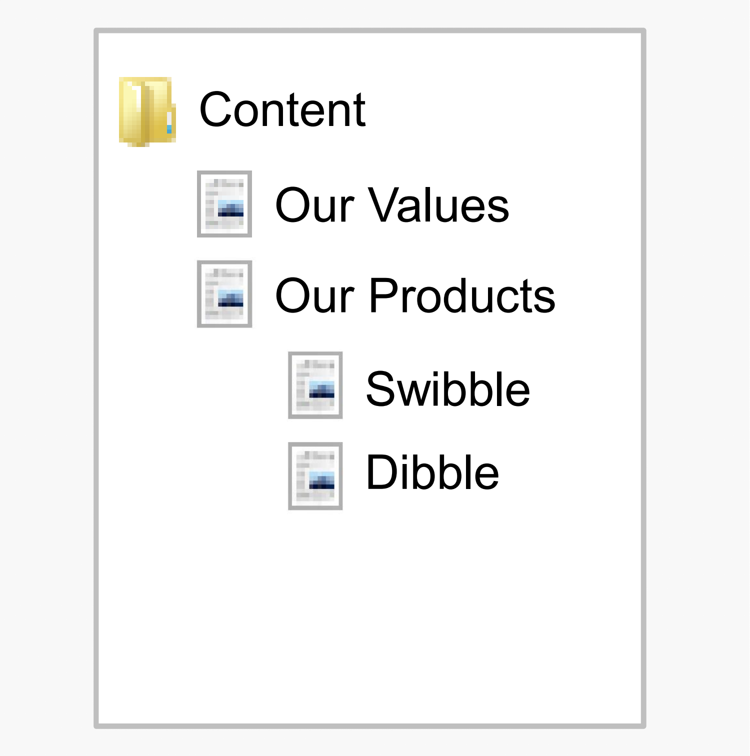
- Content
  - Our Values
  - Our Products
    - Swibble
    - Dibble

Outbound pipeline:

Creates segments
“our-products”, “swibble”…

Creates paths
“/our-products/swibble”

Creates urls
“http://site.com/our-products/swibble.aspx”



- Content
  - Our Values
  - Our Products
    - Swibble
    - Dibble

Each published content has a url segment, a.k.a. “urlName”.

“Our Products” → “our-products”
“字漢字” → ?

“Url Segment Provider”

```
public interface IUrlSegmentProvider
{
    string GetUrlSegment(IContentBase content);
}

// That one is initialized by default
public class DefaultUrlSegmentProvider : IUrlSegmentProvider
{ … }

// But feel free to use your owns
public class UrlSegmentProviderResolver
{ … }
```

Run all providers, stops at the first one that does not return null.
Falls back to default url segment provider if still null.

You are free to return whatever pleases you.

```
public class MyProvider : IUrlSegmentProvider
{
    readonly IUrlSegmentProvider _provider = new DefaultUrlSegmentProvider();

    public string GetUrlSegment(IContentBase content)
    {
        if (content.ContentTypeId != 1234) return null;
        var segment = _provider.GetUrlSegment(content);
        return string.Format("{0}-{1}", content.Id, segment);
    }
}
```

Instead of “swibble”, would return “5678-swibble”.
Becomes the native url segment, so no need for rewriting or anything.

- Default url segment provider considers
  - content.GetPropertyValue<string>(“umbracoUrlName”)
  - content.Name
- Then uses Umbraco string extension ToUrlSegment()
- String extensions are implemented by IShortStringHelper
  - Which can be extended too
  - New helper not enabled by default

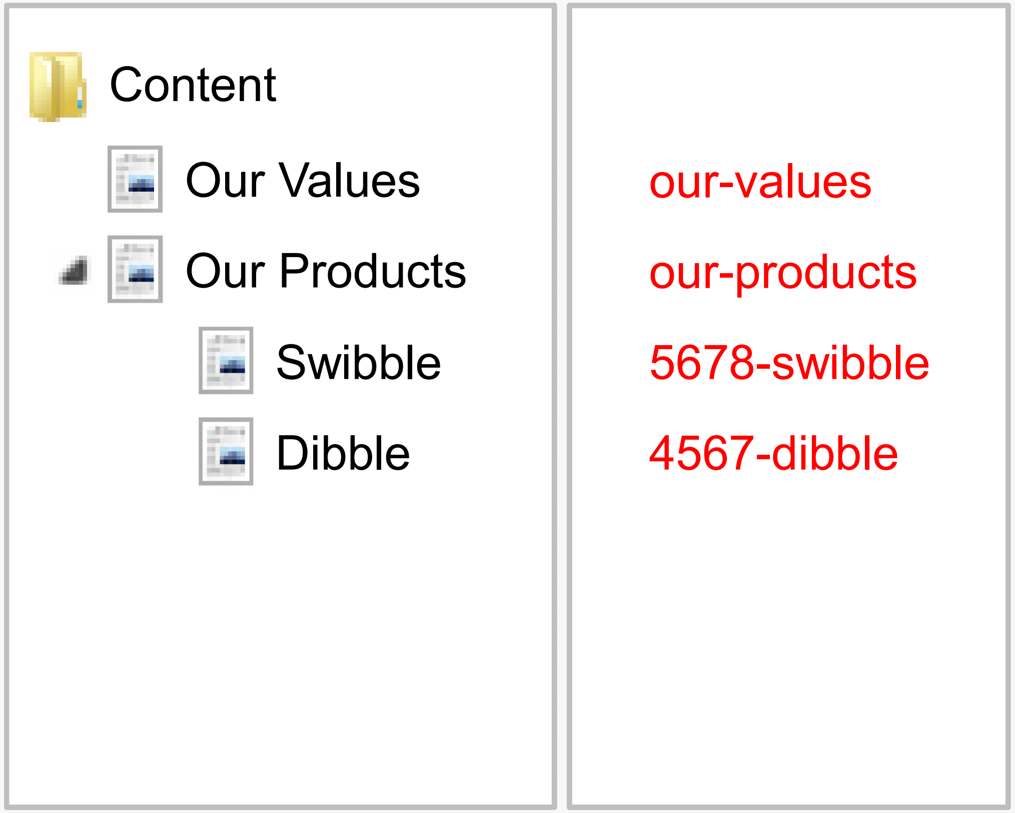
```
public interface IShortStringHelper
{
    …
}

// That one is initialized by default and is 99.99% backward-compatible with 4.x
public class LegacyShortStringHelper : IShortStringHelper
{ … }

// But feel free to use your own
public class ShortStringHelperResolver
{ … }
```

```
// That one is the future default helper
public class DefaultShortStringHelper : IShortStringHelper
{ … }
```

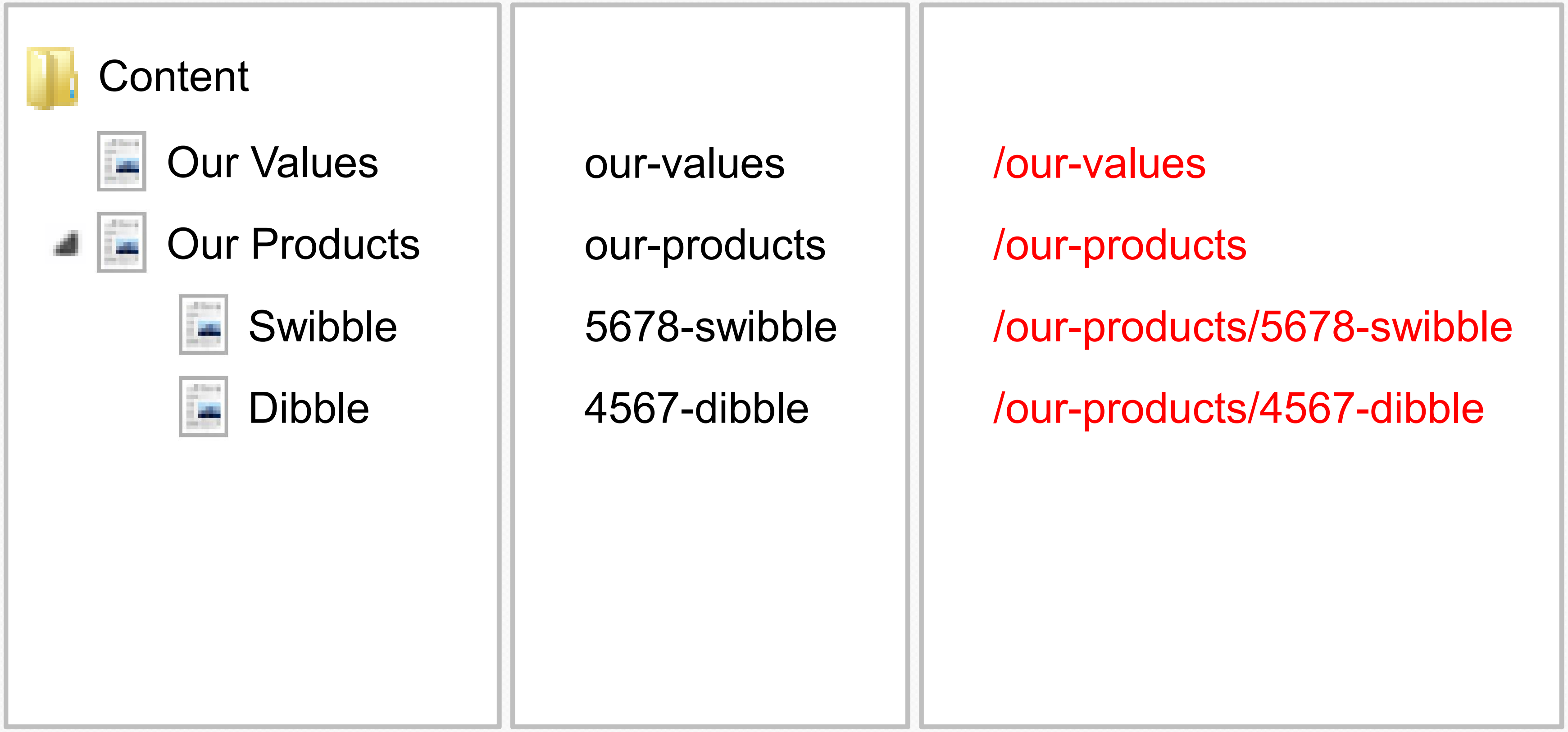
- Improved performance
- Improved utf-8 support
    - e.g. native “déjà” → “deja” substitution

- Many configuration options

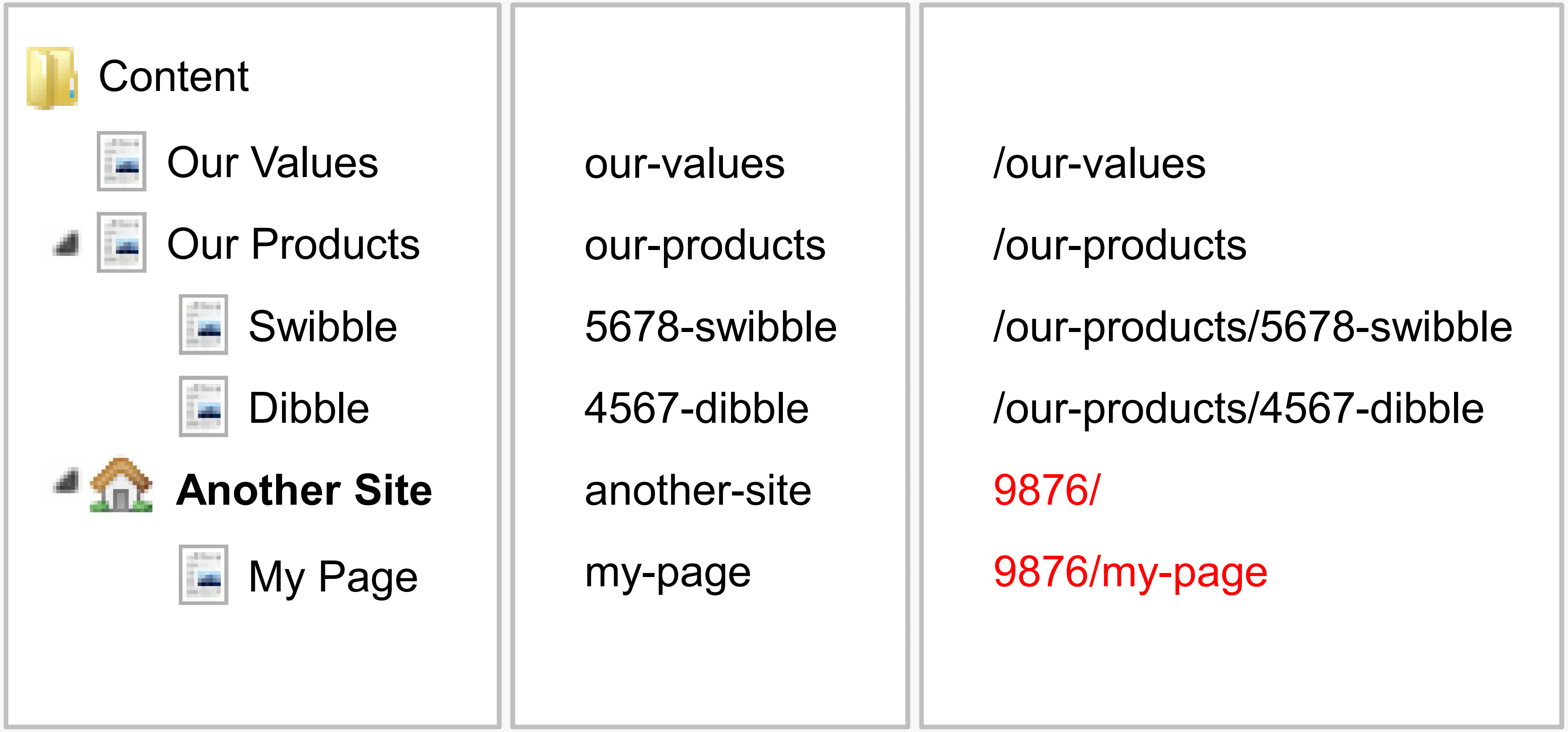
- Prob. 80% backward-compatible with pre-6
- You should try it



Content	
Our Values	our-values
Our Products	our-products
Swibble	5678-swibble
Dibble	4567-dibble



Content		
Our Values	our-values	/our-values
Our Products	our-products	/our-products
Swibble	5678-swibble	/our-products/5678-swibble
Dibble	4567-dibble	/our-products/4567-dibble

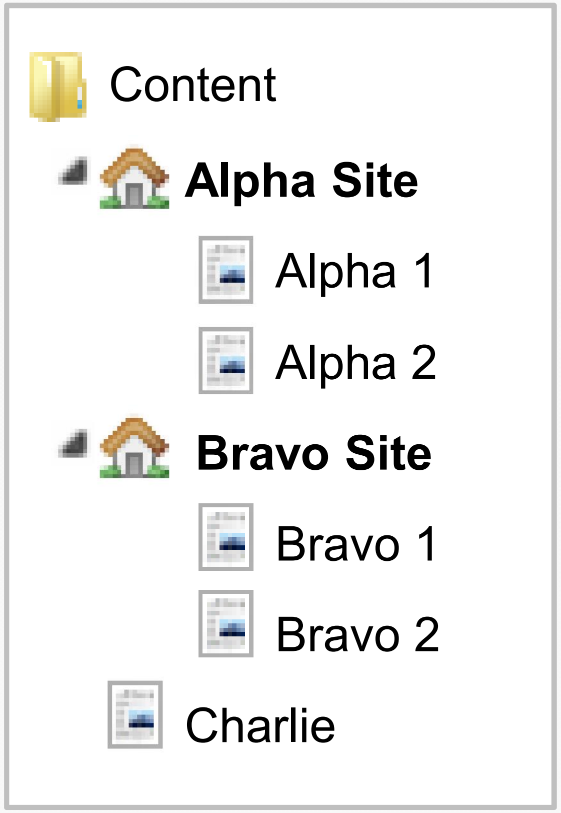


Content		
Our Values	our-values	/our-values
Our Products	our-products	/our-products
Swibble	5678-swibble	/our-products/5678-swibble
Dibble	4567-dibble	/our-products/4567-dibble
Another Site	another-site	9876/
My Page	my-page	9876/my-page

Any content node with a hostname defines a “new root” for paths.
Paths can be cached, what comes next cannot (http vs https, current request…).

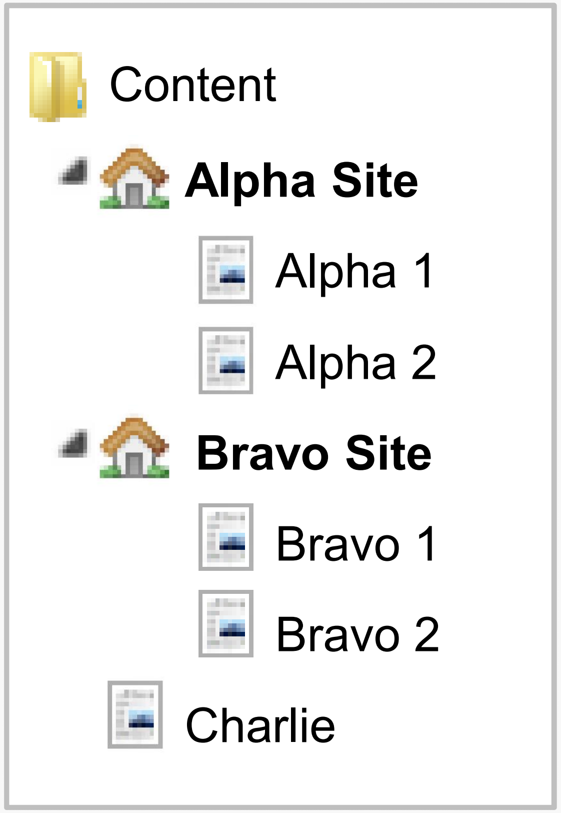
- Also note…
  - Domain without path e.g. “www.site.com”
    → “1234/path/to/page”
  - Domain with path e.g. “www.site.com/dk”
    → “1234/dk/path/to/page”
  - No domain
    → “/path/to/page”
  - Unless HideTopLevelNodeFromPath config is true
    → “/to/page”

"Url Provider"



Content		
Alpha Site	www.alpha.com, staging.alpha.com	1001/
Alpha 1	1001/alpha-1	
Alpha 2	1001/alpha-2	
Bravo Site	www.bravo.com, staging.bravo.com	1002/
Bravo 1	1002/bravo-1	
Bravo 2	1002/bravo-2	
Charlie	/charlie	

Navigating “http://www.alpha.com/alpha-1”…
What’s Alpha 2 url? “/alpha-2”?
What’s Bravo 1 url? “http://www.bravo.com/bravo-1”?



Content		
Alpha Site	www.alpha.com, staging.alpha.com	1001/
Alpha 1		1001/alpha-1
Alpha 2		1001/alpha-2
Bravo Site	www.bravo.com, staging.bravo.com	1002/
Bravo 1		1002/bravo-1
Bravo 2		1002/bravo-2
Charlie		/charlie

What about Charlie?
What-if current is https?
What about staging vs. www?

```
public interface IUrlProvider
{
    string GetUrl(UmbracoContext umbracoContext,
        int id,
        Uri current,
        UrlProviderMode mode);
}

// That one is initialized by default
public class DefaultUrlProvider : IUrlProvider
{ … }

// But feel free to use your owns
public class UrlProviderResolver
{ … }
```

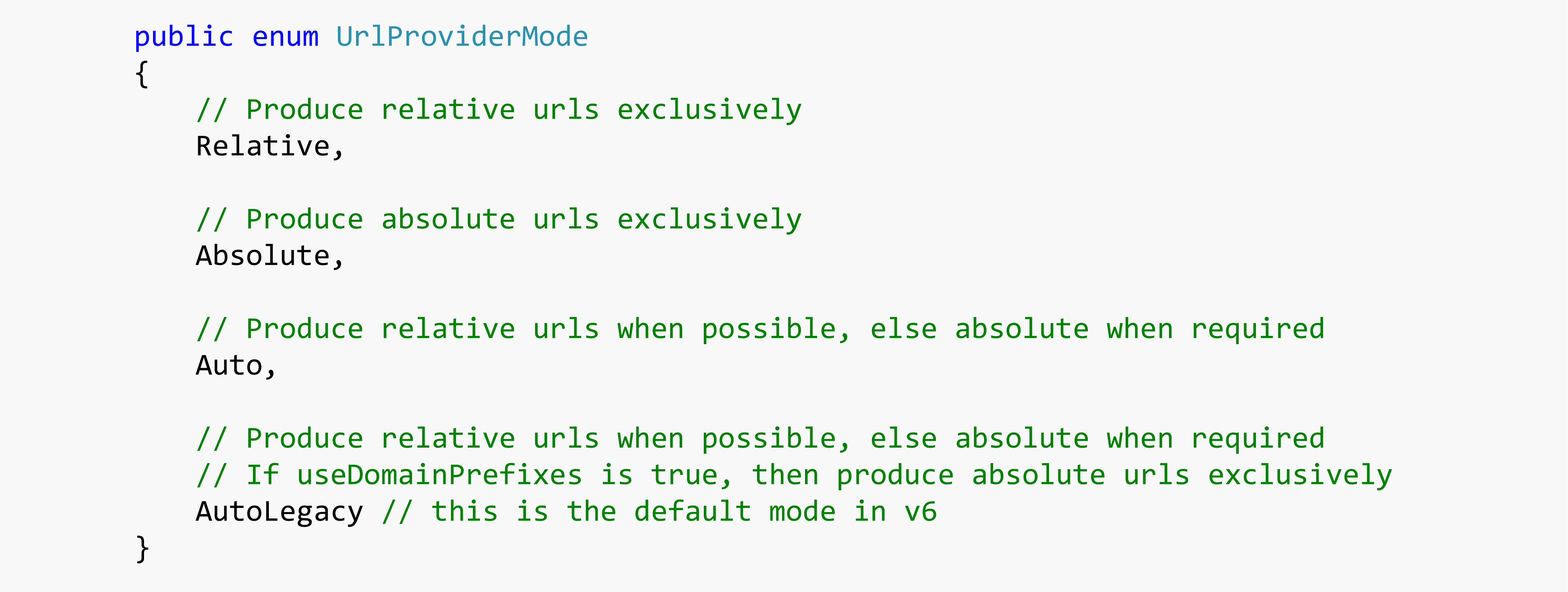
Run all providers, stops at the first one that does not return null.
Falls back to default url provider if still null.

You are free to return whatever pleases you.

- Might want to use some cache
- Has to know how to handle domains, schemes (http vs https), etc.
- Inbound might require rewriting
- Tricky to implement right → better override default

- Used whenever you write
  - @Content.Model.Url
  - @Umbraco.Url(1234)
  - @UmbracoContext.Current.UrlProvider.GetUrl(1234)

- Per-context UrlProvider
  - Provider “mode” determines absolute vs. relative urls
  - Can change the mode of the current provider
  - Default mode can be configured



```
public enum UrlProviderMode
{
    // Produce relative urls exclusively
    Relative,

    // Produce absolute urls exclusively
    Absolute,

    // Produce relative urls when possible, else absolute when required
    Auto,

    // Produce relative urls when possible, else absolute when required
    // If useDomainPrefixes is true, then produce absolute urls exclusively
    AutoLegacy // this is the default mode in v6
}
```

Auto is equivalent to AutoLegacy with useDomainPrefixes set to false

UseDomainPrefixes is ignored in every mode except AutoLegacy

Configure in /umbraco/web.routing/urlProviderMode

- If the current domain matches a root domain of the target content
  - Return a relative url
  - Else must return an absolute url

- If the target content has only one root domain
  - Use that domain to build the absolute url

- If the target content has more that one root domain
  - Figure out which one to use
  - To build the absolute url

- Complete the absolute url with scheme (http vs https)
  - If the domain contains a scheme use it
  - Else use the current request’s scheme

```
public interface ISiteDomainHelper
{
    DomainAndUri MapDomain(Uri current, DomainAndUri[] domainAndUris);
}

// That one is initialized by default
public class SiteDomainHelper : ISiteDomainHelper
{ … }

// But feel free to use your own
public class SiteDomainHelperResolver
{ … }
```

Gets the current Uri and all eligible domains.
Must return one domain.

```
public class MyApplication : ApplicationEventHandler
{
    protected override void ApplicationStarting(…)
    {
        SiteDomainHelper.AddSite("www",
            "www.alpha.com", "www.bravo.com");

        SiteDomainHelper.AddSite("staging",
            "staging.alpha.com", "staging.bravo.com");
    }
}
```

Then it knows it should pick e.g. “www.bravo.com” when current is “www.alpha.com”.
Have a better idea?

```
public class MyApplication : ApplicationEventHandler
{
    protected override void ApplicationStarting(…)
    {
        SiteDomainHelper.AddSite("www",
            "www.alpha.com", "www.bravo.com");

        SiteDomainHelper.AddSite("mobile",
            "mobile.alpha.com", " mobile.bravo.com");

        SiteDomainHelper.AddSite("staging",
            "staging.alpha.com", "staging.bravo.com");

        SiteDomainHelper.BindSites("www", "mobile");
    }
}
```

Back-end on www.alpha.com/umbraco
→ link is “www.bravo.com/bravo-2” ; alternate link is “mobile.bravo.com/bravo-2”
It works. Have a better idea?

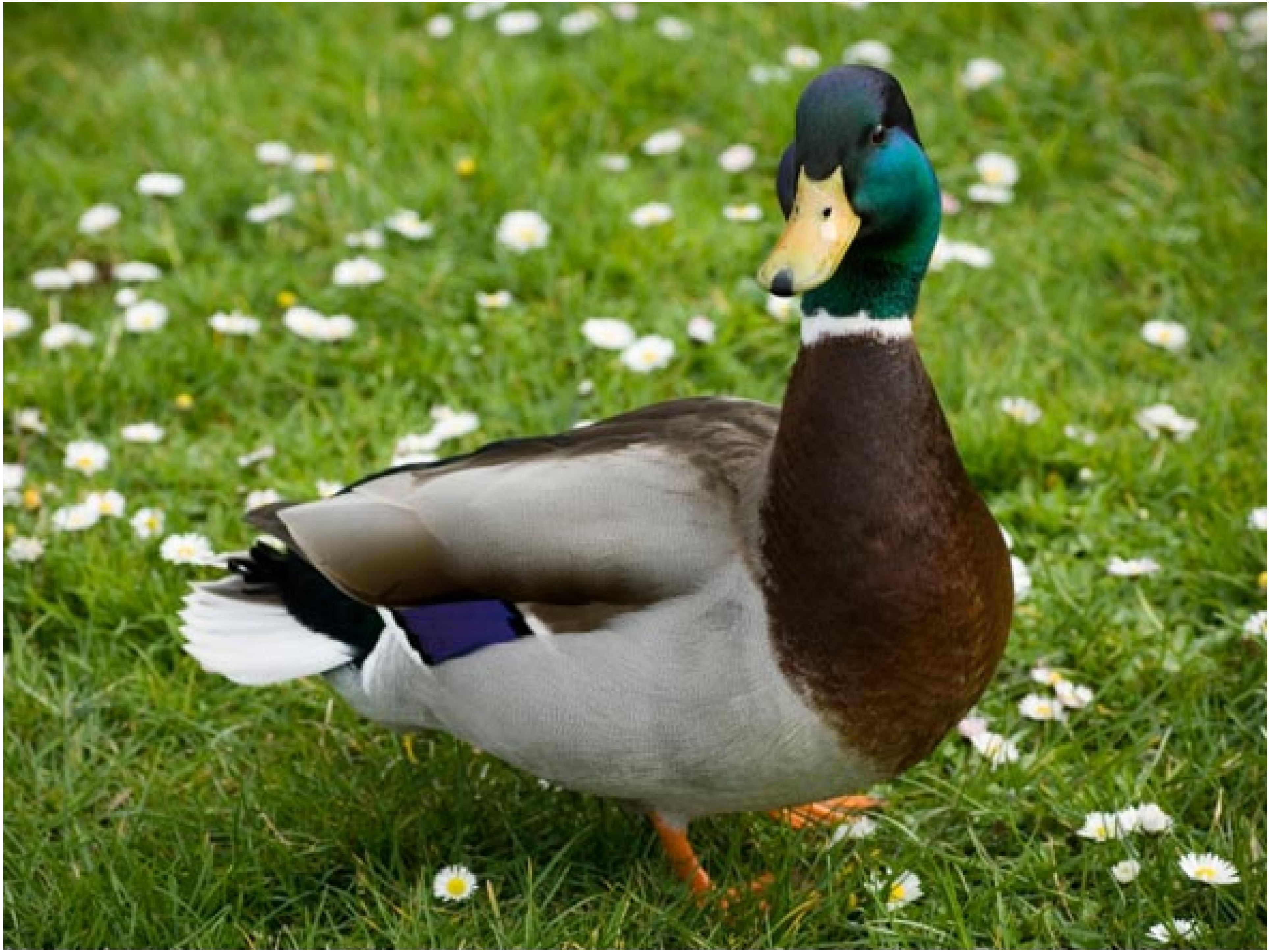
- Last steps
  - useDirectoryUrls? else add .aspx
  - addTrailingSlash? add /
  - add the virtual directory

- Finally, we have a url!

- How’s Umbraco going to handle the request?
- That’s the role of the *inbound* pipeline

- Just a few more things
  - The IUrlProvider also has a GetOtherUrls method
  - For the back-end

  - Experiment with AliasUrlProvider
  - Will show the umbracoUrlAlias url in the back-end



“Published Content Request”

```
public class PublishedContentRequest
{
    public Uri Uri { get; }
    …
}
```

Represents the request that Umbraco must handle.
Contains everything that will be needed to render it.
All this happens when the Umbraco modules thinks it’s a document to render.

```
public bool HasDomain { get; }
public Domain Domain { get; }
public Uri DomainUri { get; }

public CultureInfo Culture { get; }
```

Contains the domain and the culture.
Domain is “site.com”, Uri is “http://site.com”

```
public bool HasPublishedContent { get; }
public IPublishedContent PublishedContent { get; set; }

public bool IsInitialPublishedContent { get; }
public IPublishedContent InitialPublishedContent { get; }

public void SetIsInitialPublishedContent();
public void SetInternalRedirectPublishedContent(IPublishedContent content);
public bool IsInternalRedirectPublishedContent { get; }
```

Contains the content to render.

```
public bool HasTemplate { get; }
public string TemplateAlias { get; }
public RenderingEngine RenderingEngine { get; }

public bool TrySetTemplate(string alias);
public void SetTemplate(ITemplate template);
```

Contains the template to render the content.
And the corresponding rendering engine (WebForms or MVC).

```
public static event EventHandler<EventArgs> Prepared;


PublishedContentRequest.Prepared += (sender, args) =>
{
    var request = sender as PublishedContentRequest;

    // do something…
}
```

Triggers once the published content request is fully ready to be processed.
Up to you to change anything (content, template…).
Will come back to it later.

"Published Content Request Preparation"

```
void ProcessRequest(…) // in UmbracoModule
```

- Ensures Umbraco is ready, and the request is a document request
- Creates a PublishedContentRequest instance
- Runs PublishedContentRequestEngine.PrepareRequest() for that instance
- Handles redirects and status
- Forwards missing content to ugly 404
- Forwards to either WebForms or MVC

```
void ProcessRequest(…)
    ↳ void PrepareRequest() // in PublishedContentRequestEngine
```

- FindDomain()
- Handles redirects
- Sets culture
- FindPublishedContentAndTemplate()
- Sets culture (again, in case it was changed)
- Triggers PublishedContentRequest.Prepared event
- Sets culture (again, in case it was changed)
- Handles redirects and missing content
- Initializes a few internal stuff

```
void ProcessRequest(…)
    ↳ void PrepareRequest() // in PublishedContentRequestEngine
```

- FindDomain()
- Handles redirects
- Sets culture
- FindPublishedContentAndTemplate()
- Sets culture (again, in case it was changed)
- Triggers PublishedContentRequest.Prepared event
- Sets culture (again, in case it was changed)
- Handles redirects and missing content
- Initializes a few internal stuff

```
void ProcessRequest(…)
    ↳ void PrepareRequest() // in PublishedContentRequestEngine
        ↳ bool FindDomain()
```

- Looks for a domain matching the request Uri
- Greedy match: “domain.com/foo” takes over “domain.com”
- Sets published content request’s domain

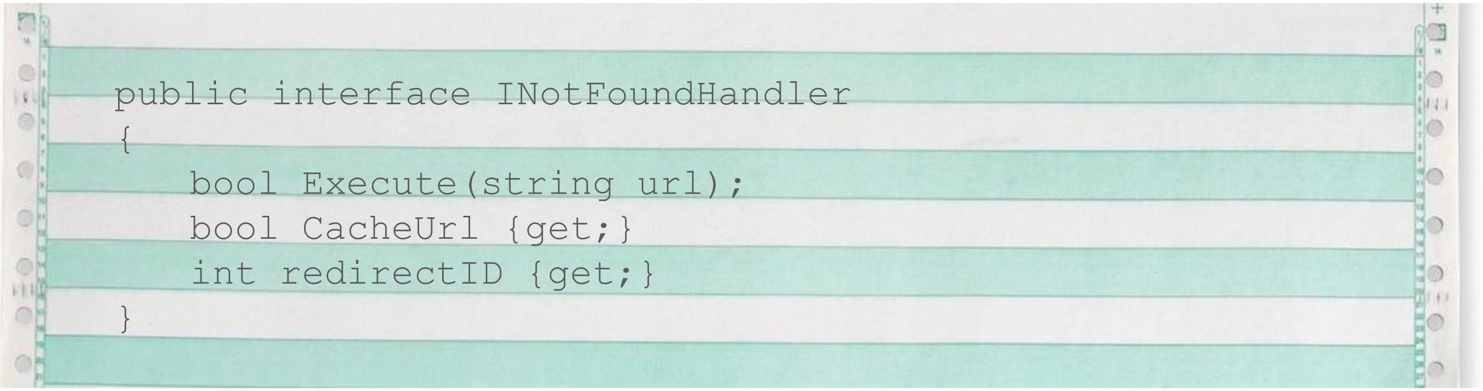
- If a domain was found
  - Sets published content request’s culture accordingly
  - Computes domain Uri based upon the current request (“domain.com” → “http://domain.com” or “https://domain.com”)
- Else
  - Sets published content request’s culture by default (first language, else system)

```
void ProcessRequest(…)
    ↳ void PrepareRequest() // in PublishedContentRequestEngine
```

- FindDomain()
- Handles redirects
- Sets culture
- FindPublishedContentAndTemplate()
- Sets culture (again, in case it was changed)
- Triggers PublishedContentRequest.Prepared event
- Sets culture (again, in case it was changed)
- Handles redirects and missing content
- Initializes a few internal stuff

```
void ProcessRequest(…)
    ↳ void PrepareRequest() // in PublishedContentRequestEngine
        ↳  void FindPublishedContentAndTemplate()
```

- FindPublishedContent ()
- Handles redirects
- HandlePublishedContent()
- FindTemplate()
- FollowExternalRedirect()
- HandleWildcardDomains()



```
public interface INotFoundHandler
{
    bool Execute(string url);
    bool CacheUrl {get;}
    int redirectID {get;}
}
```

```
public interface IContentFinder
{
    bool TryFindContent(PublishedContentRequest contentRequest);
}
```

```
public interface IContentFinder
{
    bool TryFindContent(PublishedContentRequest contentRequest);
}

// Some are registered by default
// But feel free to use your owns
public class ContentFinderResolver
{ … }
```

Umbraco runs all content finders, stops at the first one that returns true.
Finder can set content, template, redirect…

```
public class MyContentFinder : IContentFinder
{
    public bool TryFindContent(PublishedContentRequest request)
    {
        var path = request.Uri.GetAbsolutePathDecoded();

        if (!path.StartsWith("/woot"))
            return false; // not found

        // have we got a node with ID 1234?
        var contentCache = UmbracoContext.Current.ContentCache;
        var content = contentCache.GetById(1234);
        if (content == null) return false; // not found

        // render that node
        request.PublishedContent = content;
        return true;
    }
}
```

```
public class ContentFinderByNiceUrl : IContentFinder
{
    public virtual bool TryFindContent(PublishedContentRequest request)
    {
        string path = request.HasDomain
            // eg. 5678/path/to/node
            ? request.Domain.RootNodeId.ToString() + …
            // eg. /path/to/node
            : request.Uri.GetAbsolutePathDecoded();

        var node = FindContent(request, path);
        return node != null;
    }
}
```

Default finder will look for content under the domain root.
This is an *un-breaking* change.
What if you *really* want a broken Umbraco?

```
public class MyContentFinder : ContentFinderByNiceUrl
{
    public override bool TryFindContent(PublishedContentRequest request)
    {
        if (base.TryFindContent(request)) return true;

        if (!request.HasDomain) return false;

        var path = request.Uri.GetAbsolutePathDecoded();
        var node = FindContent(request, path);
        return node != null;
    }
}
```

Done.

```
public class MyApplication : ApplicationEventHandler
{
    protected override void ApplicationStarting(…)
    {
        // Insert my finder before ContentFinderByNiceUrl
        ContentFinderResolver.Current
            .InsertTypeBefore<ContentFinderByNiceUrl, MyContentFinder>();

        // Remove ContentFinderByNiceUrl
        ContentFinderResolver.Current.RemoveType<ContentFinderByNiceUrl>();
    }
}
```

Warranty Void
If Tampered
UMBRACO VERSION 6.1 PIPELINE

```
public class ContentFinderByNotFoundHandlers : IContentFinder
{
    public bool TryFindContent(PublishedContentRequest request)
    {
        // Runs legacy NotFoundHandler
        // As per 404handlers.config
        //
        // umbraco.SearchForAlias → ContentFinderByUrlAlias
        // umbraco.SearchForProfile → ContentFinderByProfile
        // umbraco.SearchForTemplate → ContentFinderByNiceUrlAndTemplate
        // umbraco.handle404 → ContentFinderByLegacy404


        …
    }
}
```

Enabled by default as the last finder.
Runs every legacy NotFoundHandler as per 404handlers.config.
As of 6.x NotFoundHandler are still supported.

```
public class MyApplication : ApplicationEventHandler
{
    protected override void ApplicationStarting(…)
    {
        ContentFinderResolver.Current
            .InsertType<ContentFinderByNotFoundHandler<MyNotFoundHandler>>();
    }
}
```

Wraps one NotFoundHandler in a content finder.

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
```

- FindPublishedContent ()
- Handles redirects
- HandlePublishedContent()
- FindTemplate()
- FollowExternalRedirect()
- HandleWildcardDomains()

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
        ↳ void HandlePublishedContent()
```

- No content?
  - Run the LastChanceFinder
  - Is an IContentFinder, resolved by ContentLastChanceFinderResolver
  - By default, is null → ugly 404
- Follow internal redirects
  - Take care of infinite loops
- Ensure user has access to published content
  - Else redirect to login or access denied published content
- Loop while there is no content
  - Take care of infinite loops

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
```

- FindPublishedContent ()
- Handles redirects
- HandlePublishedContent()
- FindTemplate()
- FollowExternalRedirect()
- HandleWildcardDomains()

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
        ↳ void FindTemplate()
```

- Use *altTemplate* if
  - Initial content
  - Internal redirect content, and InternalRedirectPreservesTemplate is true

- No alternate template?
  - Use the current template if one has already been selected
  - Else use the template specified for the content, if any

- Alternate template?
  - Use the alternate template, if any
  - Else use what’s already there: a template, else none

- Alternate template is used only if displaying the intended content
  - Except for internal redirects
  - If you enable InternalRedirectPreservesTemplate
  - Which is false by default

- Alternate template replaces whatever template the finder might have set
  - ContentFinderByNiceUrlAndTemplate
  - /path/to/page/template1?altTemplate=template2 → template2

- Alternate template does *not* falls back to the specified template for the content
  - /path/to/page?altTemplate=missing → no template
  - Even if the page has a template

- But preserves whatever template the finder might have set
  - /path/to/page/template1?altTemplate=missing → template1

- Yes, these rules are arbitrary… feedback?

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳  void FindPublishedContentAndTemplate()
```

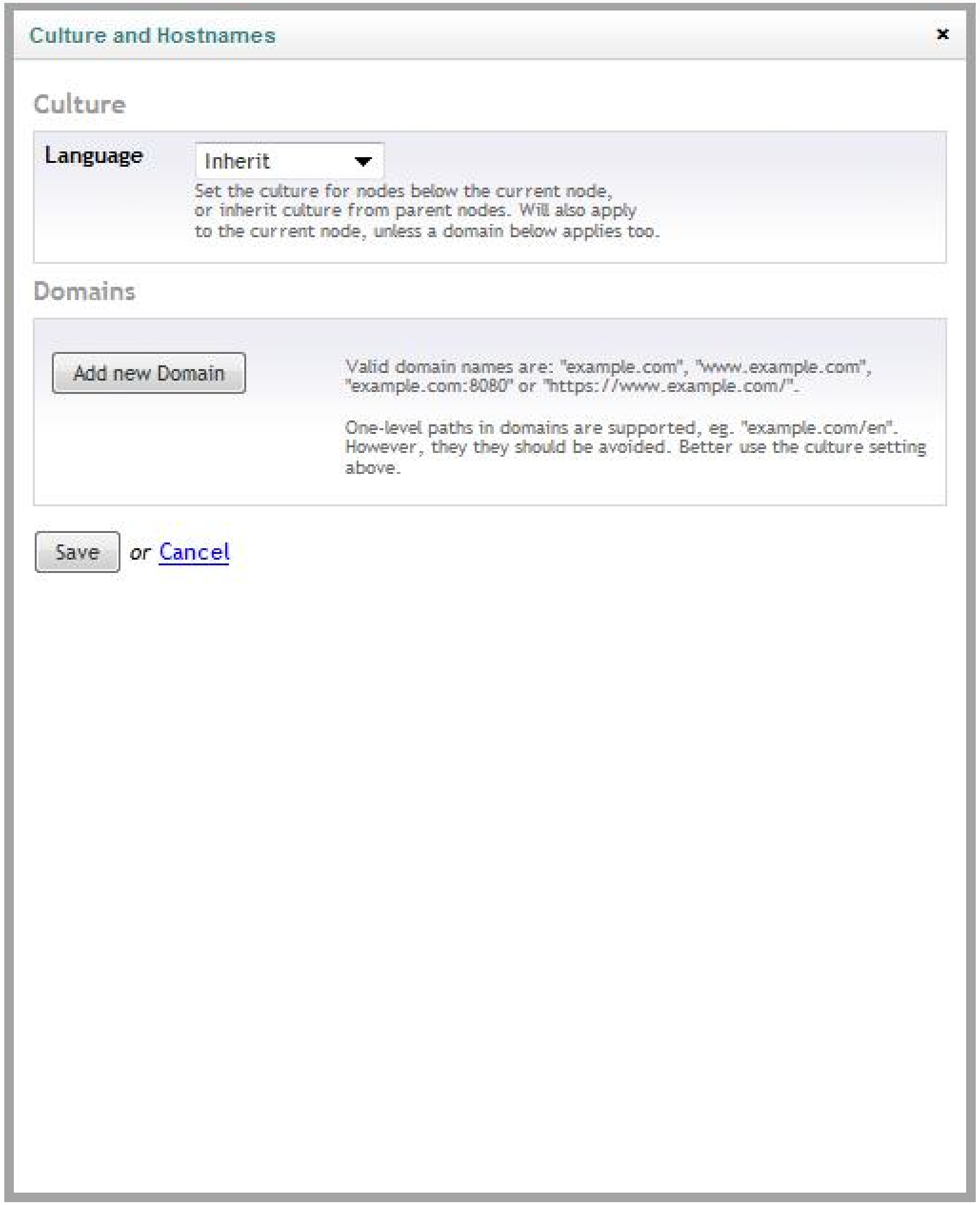
- FindPublishedContent ()
- Handles redirects
- HandlePublishedContent()
- FindTemplate()
- FollowExternalRedirect()
- HandleWildcardDomains()

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
        ↳ void FollowExternalRedirect()
```

- content.GetPropertyValue<string>(“umbracoRedirect”)
- If it’s there, sets the published content request to redirect to the content
- Will trigger an external (browser) redirect

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
```

- FindPublishedContent ()
- Handles redirects
- HandlePublishedContent()
- FindTemplate()
- FollowExternalRedirect()
- HandleWildcardDomains()



Culture and Hostnames

Culture

Language: Inherit

Set the culture for nodes below the current node, or inherit culture from parent nodes. Will also apply to the current node, unless a domain below applies too.

Domains

Valid domain names are: "example.com", "www.example.com", "example.com:8080" or "https://www.example.com/".

One-level paths in domains are supported, eg. "example.com/en". However, they they should be avoided. Better use the culture setting above.

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
     ↳ void FindPublishedContentAndTemplate()
         ↳ void HandleWildcardDomains()
```

- Finds the deepest wildcard domain between
  - domain root (or top)
  - request’s published content
- If found, updates the request’s culture accordingly
- This actually implements separation between hostnames and cultures
- Woot!

```
void ProcessRequest(…)
 ↳ void PrepareRequest() // in PublishedContentRequestEngine
```

- FindDomain()
- Handles redirects
- Sets culture
- FindPublishedContentAndTemplate()
- Sets culture (again, in case it was changed)
- Triggers PublishedContentRequest.Prepared event
- Sets culture (again, in case it was changed)
- Handles redirects and missing content
- Initializes a few internal stuff

```
PublishedContentRequest.Prepared += (sender, args) =>
{
    var request = sender as PublishedContentRequest;

    if (!request.HasPublishedContent) return;

    var content = request.PublishedContent;
    var redirect = content.GetPropertyValue<string>("myRedirect");

    if (!string.IsNullOrWhiteSpace(redirect))
        request.SetRedirect(redirect);
}
```

UmbracoModule will pick the redirect and redirect…
No need to write your own redirects.

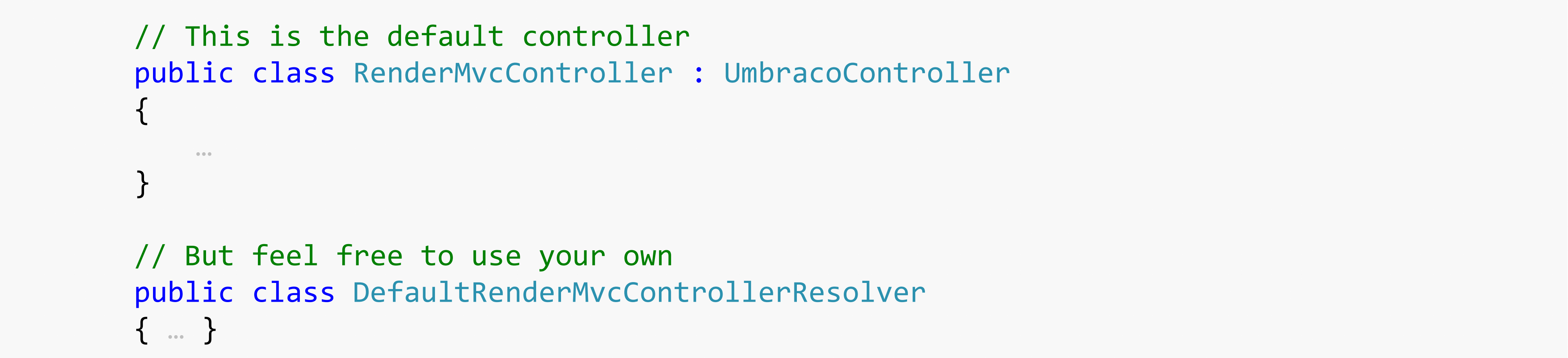
```
void ProcessRequest(…)
 ↳void PrepareRequest() // in PublishedContentRequestEngine
```

- FindDomain()
- Handles redirects
- Sets culture
- FindPublishedContentAndTemplate()
- Sets culture (again, in case it was changed)
- Triggers PublishedContentRequest.Prepared event
- Sets culture (again, in case it was changed)
- Handles redirects and missing content
- Initializes a few internal stuff

```
void ProcessRequest(…) // in UmbracoModule
```

- Ensures Umbraco is ready, and the request is a document request
- Creates a PublishedContentRequest instance
- Runs PublishedContentRequestEngine.Prepare() for that instance
- Handles redirects and status
- Forwards missing content to ugly 404
- Forwards to either WebForms or MVC
  - Missing template goes to MVC
  - WebForms is pretty much back to 4.x
  - MVC has been made possible by the pipeline

- WebForms
  - Back to 4.x
- MVC
  - Model, View, Controller
  - Custom controler, route hijacking…
  - Shannon’s presentation



```
// This is the default controller
public class RenderMvcController : UmbracoController
{
    …
}

// But feel free to use your own
public class DefaultRenderMvcControllerResolver
{ … }
```

This is MVC, so we have a controller
There’s one by default but you can use your own
Still time to change the view

- Route hijacking
  - MyContentTypeController
    - Will run in place of the default controller
    - For every content of type MyContentType
  - Specific action runs if name matches the template alias
  - Otherwise default (Index) action runs

- The case of the missing template

- Route hijacking?
- Otherwise
  - HandlePublishedContent()
  - FindTemplate()
  - Handle redirects, etc.
  - Ugly 404 (w/ message)
  - Transfer to WebForms or MVC…

- What else?
  - Routing the /base Rest service
  - Routing the WebAPI
  - MVC routes
  - …

ZpqrtBnk
Umbraco

Now, feedback and questions!

Stéphane Gay
2013/06/13

sgay@pilotine.com
@zpqrtbnk