Kentico 2020 Beta Documentation

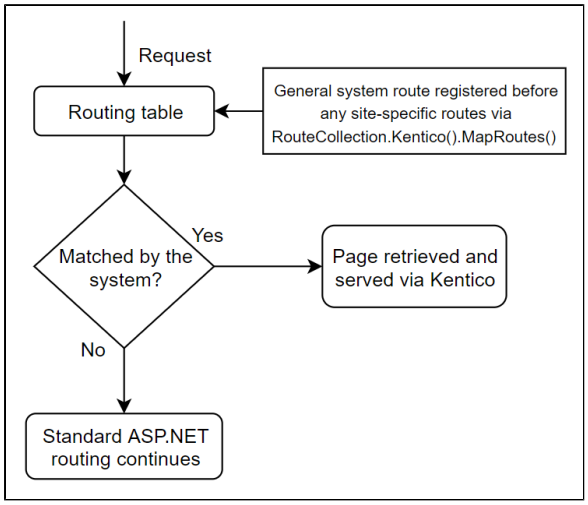
http://docs.kentico.com

Content tree-based routing

With content tree-based routing, the system automatically generates and matches page URLs based on their position in the website's content tree. You do not need to perform any manual route mapping and configuration on the side of the MVC application. All routing logic is handled for you by the system.

This method of routing is not a full replacement of the standard ASP.NET routing model. Instead, it augments the routing pipeline, automatically handling requests targeting the site's pages. You can still set up additional routes for your MVC application.

This simplified diagram models request handling when using content tree-based routing:



Incoming requests are matched against a general route and their target URL compared with existing pages in the system. If there is a match, the system takes over the routing logic. Otherwise, the request continues matching against routes registered further down the routing table using conventional ASP.NET routing.

Once the request is taken over by Kentico, all routing happens automatically. You only need to provide a view providing the desired output formatting. However, you can also customize the processing logic, providing your own controllers and view models as required.

Page URL generation

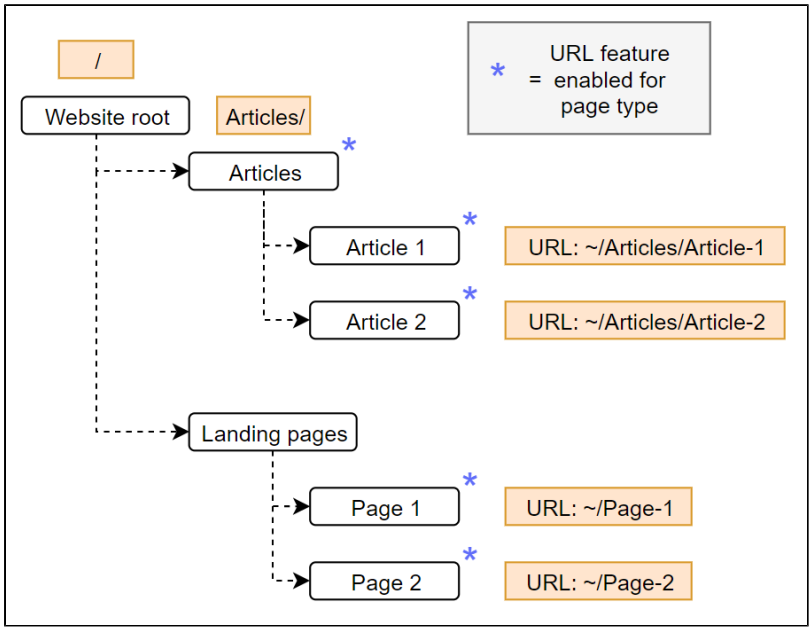
The system does not automatically generate URLs for all nodes (pages) in the content tree. A URL is generated only for pages whose page type has the URL feature enabled. The URL generation occurs when:

- a new page is created
- the URL slug of an existing page is updated

The overall URL is driven by the page's location in the site's content tree:

- The system takes all of the page's ancestors up to the site's root ('/'):
  - If an ancestor has the *URL* feature enabled for its page type, it contributes a segment to the page's resulting URL. All other ancestors are skipped.
  - By default, the segment is formed from the page's title converted to a URL-sanitized string. A page's title is the value of a field specified by the page type's *Page name source field* setting (configured on the page type's *Fields* tab).

The following diagram illustrates how URLs are generated based on a page's position in the content tree and the configuration of its ancestors:



The *Articles* section holds a collection of articles (*Article 1* and *Article 2*). All pages are based on page types with the *URL* feature enabled. As a result, both articles also contain their parent node segment (*Articles*) in their URL.

The *Landing pages* section holds a collection of landing pages (*Page 1* and *Page 2*). Only the landing pages are based on a page type with the *URL* feature enabled. As a result, the URLs of both landing pages do not include their parent.

URL generation on multilingual sites

On multilingual sites, a page URL is generated for all culture versions supported by the site. This is intended as a prevention mechanism against possible URL collisions in the future. It also "reserves" the URL, ensuring requests for yet non-existing language versions of router-handled pages do not match against incorrect routes.

Page language versions served by the router

If a culture version of a page that does not yet exist is requested – only its generated URL exists in the database – the result depends on:

- the Combine with default culture setting, or
- whether the page data was retrieved using DocumentQuery with a CombineWith fallback.

Handling of non-existing or unpublished pages

Once a URL is generated for a page, it persists until the corresponding page is deleted. This includes cases when:

- the page is unpublished
- the version of the page does not exist in the requested culture version and there is no fallback to a default culture configured.

If a request targets such a page, the system responds with the HTTP 404 (Not Found) status and the request does not continue matching against other routes. This is to prevent accidentally matching requests intended to be handled by Kentico against other loosely constrained routes registered further down the routing table. You can use conventional 404 error handling for such cases.

Configuring your project to use content tree-based routing

To start using content-tree based routing in your project:

1. Enable content-tree based routing.
2. Create page types with the URL feature.
3. Set up the environment in the connected MVC application.

Enabling content tree-based routing

Content tree-based routing needs to be enabled in both the MVC live-site application and the **Settings** application in the administration interface.

Kentico administration application

The Kentico administration application needs to have its **Routing** mode set to *Based on content tree*:

> New installations of Kentico have their routing mode set to *Based on content tree* by default.

1. Open the **Settings** application in the Kentico administration application.
2. Switch to the **URLs and SEO** tab.
3. Under the **Routing** section, select **Based on content tree** for the **Routing mode** setting.
4. **Save** the changes.

The administration application is now configured to use content tree-based routing. The setting is global and applies to all sites in the system.

MVC live-site application

On the side of the MVC application, you need to enable content tree-based routing as a feature used by your project.

1. Open your MVC project in Visual Studio.
2. Enable the feature at the start of your application's life cycle, for example in the **Application_Start** method of your project's **Global.asax** file.

   > MVC projects created by the installer contain the *ApplicationConfig* class, whose *RegisterFeatures* method is called in the *Application_Start* method by default. You can use this class to encapsulate all of your *ApplicationBuilder* code (enabling and configuring of Kentico MVC features).

   > **Note**: The feature must be enabled **before** you register routes into the application's *RouteTable*. The *Kentico().MapRoutes()* method adds routes that handle automatic routing when the feature is enabled.

3. Call the **UsePageRouting** method of the *ApplicationBuilder* instance.

```
using Kentico.Content.Web.Mvc.Routing;
using Kentico.Web.Mvc;

...

protected void Application_Start()
{
    ...

    // Gets the ApplicationBuilder instance
    // Allows you to enable and configure selected Kentico MVC integration
features
    ApplicationBuilder builder = ApplicationBuilder.Current;

    // Enables content tree-based routing
    builder.UsePageRouting();

    ...
}
```

Content tree-based routing is now enabled for the MVC application. The *Kentico().MapRoutes()* method called in your MVC application's *RouteConfig* registers a general route that matches URLs generated by the system for page types with the URL feature enabled.

Continue by configuring your project.

Setting up content tree-based routing

Content tree-based routing handles URL generation and routing for pages of appropriately configured page types. Upon creation, each page is granted a URL based on its position in the content tree and the configuration of its ancestor nodes. For a detailed overview of this feature, see Content tree-based routing.

This page details both necessary and optional steps you need to perform on the side of the MVC application to fully integrate the routing feature into your project.

There are two main approaches you can use when configuring content tree-based routing:

- Basic – requires you to provide MVC view files to define the output format of pages. A corresponding controller and view model are handled for you by the system.
- Advanced – builds upon the basic routing scheme. Allows you to provide a custom controller and view model to handle requests for specific page types and sections of the content tree in an individual manner.

Both routing schemes require a page type with the URL feature enabled. The URL feature indicates that you want pages created from this page type to be accessible using a URL. When used with content tree-based routing, it configures the system to automatically generate URLs for new pages. See Creating page types.

Additionally, when handling page types with the **page builder** feature enabled, there are a couple of additional prerequisites to keep in mind when setting up the routing scheme. See the page builder section for more information.

Setting up basic routing

The basic routing scheme is the simplest form of routing. You only need to provide a view formatting the page output. The controller, view model, and all routing logic is handled for you by the system.

1. In your MVC project, create a view under **Views/Shared/PageTypes**. The view needs to be named after the class name of the created page type. Based on this convention, the system automatically matches requests for pages of particular page type to the corresponding view.
   - For example: assuming you have a page type called *MySite.Article* (namespace.codename), name the view "*MySite.Article.cshtml*". The full path to the view is then: *~/Views/Shared/PageTypes/MySite.Article.cshtml*
2. Implement the view according to your requirements. The system provides access to the data of the requested page via the **Page** property of a generic **PageViewModel<TPage>** view model (*Kentico.Content.Web.Mvc.Routing* namespace). Substitute the *TPage* generic with one of the following types:
   - **CMS.DocumentEngine.TreeNode** – a general class representing the requested page in the system. Allows you to directly access properties common to all pages, such as DocumentName, DocumentPublishFrom, etc. To access fields specific to your given page type, use the *GetValue* method, or generate a wrapper class for the page type.

     ```
     @using CMS.DocumentEngine

     @* Declares the generic model using the 'TreeNode' class *@
     @model Kentico.Content.Web.Mvc.Routing.PageViewModel<TreeNode>

     @* Renders the name of the page *@
     @Model.Page.DocumentName

     @* Accesses the 'Text' field of the page type *@
     @Html.Raw(Model.Page.GetValue("Text"))
     ```

   - Generated wrapper class – a class generated by the system that provides direct access to all custom fields of the page type.

```
@* Includes the namespace of the generated page type wrapper class *@
@using CMS.DocumentEngine.Types.MySite

@model Kentico.Content.Web.Mvc.Routing.PageViewModel<Article>

@* Renders the name of the page *@
@Model.Page.DocumentName

@* Fields declared for the specific page type can be accessed directly from
the generated wrapper class.
   This line of code loads data from the 'Text' field of the page type. *@
@Html.Kentico().ResolveUrls(Model.Page.Text)
```

When a page matching this page type is requested, the system automatically displays the view using a default controller and the generic view model. The system also ensure all contextual data of the requested page (matching the URL of the request).

Output caching support

Output caching is not supported for the basic routing scheme. If you wish to use output caching, you need to use a custom controller as described in the advanced section.

If you desire finer control over the processing logic and the data passed into the view, use the advanced mode

Setting up advanced routing

When developing view for pages with more advanced content or functionality, you can provide your own controller together with a view model. For example, use the advanced routing scheme for pages where data stored within a single page in Kentico is insufficient (e.g., when displaying additional lists of related articles, a catalog of products, etc.). This grants you full control over the data and processing logic. You can execute arbitrary code within the controller, pass additional data required by the view, or switch between different views as needed.

When configuring advanced routing, you need to perform the following steps:

1. Decide which page types and sections of the content tree will be managed using your route.
2. Implement the controller logic (together with view models and views).
3. Register the controllers in the system together with the routes for which they are responsible.

Advanced page routing overview

Under the advanced routing scheme, requests are still matched by a system route. However, instead of routing the request to a hidden controller, the system reroutes the request to a matching custom controller. A matching controller is found using the page type and node alias path (location in the content tree) of the requested page. You provide this information using the **RegisterPageRoute** assembly attribute (*Kentico.Content.Web.Mvc.Routing* namespace).

The *RegisterPageRoutes* attributes takes the following required parameters:

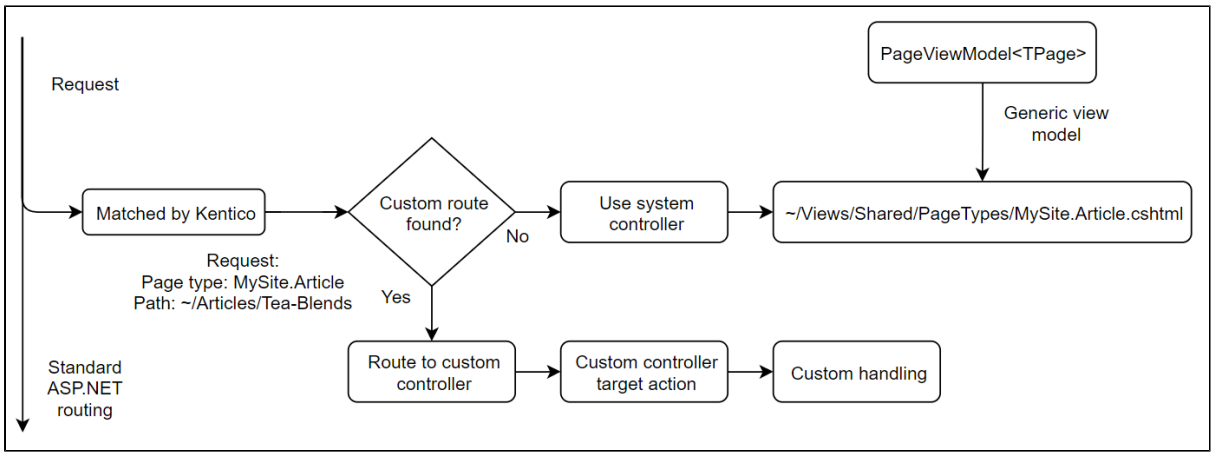
- **string ClassName** – the class name of the page type this controller handles. The class name is in the *namespace.codename* format. Both are set during page type creation. Generated classes expose the class name under the *CLASS_NAME* constant.
- **Type ControllerType** – the *Type* of the controller assigned to handling the requests for the page type.

When a controller is registered using only these required parameters, the system invokes the controller's **Index** action in response to all requests matched with pages of the supplied page type.

To further tailor the response, you can provide additional parameters when registering the route:

- **string Path** – the node alias path of an item in the content tree for which you want to use this controller. Use this property if you wish to provide special handling for a particular page.
    - If there are multiple matches to a single request, the longest (most specific) path wins.
    - If not specified, the registered controller is used to respond to all requests for pages of the corresponding page type.
- **string ActionName** – allows you to target which controller action responds to the matched request. If not specified, the default **Index** action is used.

The following diagram compares the basic and advanced routing schemes:



The registration parameters can be combined to, for example:

- have multiple page types handled by a single controller and action
- have multiple page types handled by a single controller and different actions
- have a single page type handled by different controllers depending on the location of specific pages in the content tree

> **Multiple identical registrations**
>
> Multiple identical registrations (e.g., you registered multiple controllers with the same path and action for a single page type) are not supported. If the system detects duplicate registrations, it fails with an exception during application startup.

Examples

The following are example of custom route registrations:

Multiple page types - single controller - single action

```
// Requests for all pages of the 'Coffee', 'Brewer' and 'ElectricGrinder' page types
are handled using the 'Detail' action of the Product controller
[assembly: RegisterPageRoute(Coffee.CLASS_NAME, typeof(ProductController), ActionName
= "Detail")]
[assembly: RegisterPageRoute(Brewer.CLASS_NAME, typeof(ProductController), ActionName
= "Detail")]
[assembly: RegisterPageRoute(ElectricGrinder.CLASS_NAME, typeof(ProductController),
ActionName = "Detail")]
```

Multiple page types - single controller - different actions

```
// Requests for all pages of the 'ArticleSection' page type are handled using the
'Index' action of the Articles controller
[assembly: RegisterPageRoute(ArticleSection.CLASS_NAME, typeof(ArticlesController))]
// Requests for all pages of the 'Article' page type are handled using the 'Show'
action of the Articles controller
[assembly: RegisterPageRoute(Article.CLASS_NAME, typeof(ArticlesController),
ActionName = "Show")]
```

Single page type - multiple controllers - each for a different page in the content tree

```
// Requests for a page of the 'ProductSection' page type located at '/Store/Brewers'
is handled using the 'Index' action of the Brewers controller
[assembly: RegisterPageRoute(ProductSection.CLASS_NAME, typeof(BrewersController),
Path = "/Store/Brewers")]
// Requests for a page of the 'ProductSection' page type located at '/Store/Grinders'
is handled using the 'Index' action of the Grinders controller
[assembly: RegisterPageRoute(ProductSection.CLASS_NAME, typeof(GrindersController),
Path = "/Store/Grinders")]
// Requests for a page of the 'ProductSection' page type located at '/Store/Coffees'
is handled using the 'Index' action of the Coffees controller
[assembly: RegisterPageRoute(ProductSection.CLASS_NAME, typeof(CoffeesController),
Path = "/Store/Coffees")]
```

Implementing a custom route

As described in the previous section, a custom route is nothing but a controller that takes over when the request matches certain criteria. When the system delegates handling to such a controller, the rest of the processing logic is completely in your hands. You can run any custom code within the controller, pass any type of required data to the view, or switch between completely different views based on the current scenario.

The development process for custom routes follows standard MVC practices:

1. Create a new controller class in your project.
2. Implement the *Index* action and any other required actions.
    - If you need to access the data of the currently requested page, use the *GetPage<TPageType>()* method of the *IPageDataContextRetriever* service. The method takes either *CMS.DocumentEngine.TreeNode* or a page type wrapper class as its generic parameter and returns a page object of the specified page type. We recommend using dependency injection to initialize an instance of the service.

```
private readonly IPageDataContextRetriever dataRetriever;

public ArticlesController(IPageDataContextRetriever dataRetriever)
{
        // Initializes an instance of a service that provides data context
of pages matching the requested URL
        this.dataRetriever = dataRetriever;
}

public ActionResult Show()
{
...
        // Gets the page of the Article page type matching the currently
requested URL
        var article = dataRetriever.GetPage<Article>();
...
}
```

3. Create any required view model classes used to pass data from the controller to the view.
4. Create views required by the controller actions.
    - The output must be a full HTML page, so the view must include the following:
        - Full HTML markup, including the *html, head* and *body* elements
        - Links to all necessary resources, such as stylesheets and scripts
    - Use MVC layouts with the view for any shared output code (for example your site's main layout).
5. Register the route in the system.

The registered controller takes over for the default system controller when the request matches the criteria specified in the route registration attribute.

Handling page builder-enabled page types

For page types that use content tree-based routing and also have the page builder feature enabled, you need to ensure certain prerequisites based on the employed routing scheme and the use of page templates.

When using **basic** routing and

- The page is based on a page template:
    - You do not need to create a view corresponding to the page type under the *~/Views/Shared/PageTypes/className.cshtml* location. However, if a matching view is detected, it takes precedence over the page template.
- The page is not based on a page template:
    - The view for the corresponding page type (*~/Views/Shared/PageTypes/className.cshtml*) needs to include page builder scripts and styles (either directly or through the assigned layout).

When using **advanced** routing and

- The page is based on a page template:
    - The controller action handling the request needs to return a **TemplateResult** object. The object needs to be initialized with the ID of the requested page. You can obtain the data context of the requested page using **IPageDataContextRetriever** (see Implementing a custom route). The identifier of the page is stored in the **DocumentID** property.

      ```
      return new TemplateResult(dataRetriever.GetPage<Article>().DocumentID);
      ```

- The page is not based on a page template:

- The view needs to include page builder scripts and styles to ensure page content is loaded correctly (either directly or through the assigned layout).

Retrieving generated page URLs using the API

URLs generated for pages can be retrieved using the following API:

- **TreeNode.RelativeURL** – the **RelativeURL** property contains the virtual (relative) URL of the page. You can use the **Content** method of the *System.Web.Mvc.UrlHelper* class to convert a virtual relative path to an application absolute path in the code of your views.
  - The *RelativeURL* property can also be accessed via the generated page type wrapper classes.
- **Url.Kentico().PageUrl(string nodeAliasPath)** – extension method for *System.Web.MVC.UrlHelper* intended for use in the code of views. Returns the application absolute URL of a page from a specified location in the content tree. You can provide the following optional parameters:
  - string cultureCode – the culture code of the desired language variant.
  - string siteName – the code name of the site from which to retrieve the page.

  Example

  ```
  @using Kentico.Content.Web.Mvc

  @* Renders a link tag targeting the 'Tea Blends' article in the 'en-us'
  culture from the current site (mapped to the domain of the current request)
  *@
  Html.HtmlLink(Url.Kentico().PageUrl("/Articles/Tea-Blends", "en-us"), "link
  text")
  ```

  Instead of using string constants with node alias paths directly in the code of your views, use a helper class to store the paths to frequently targeted pages as constants. For example:

  ```
  public static class ContentItemIdentifiers
  {
          public const string HOME = "/Home";
          public const string ARTICLES = "/Articles";
  }
  ```

- **Url.Kentico().PageMainUrl()** – returns the main URL of the page in cases where a URL rewrite to an alternative URL occurred.
  - The method is located in the *Kentico.Content.Web.Mvc.Routing* namespace.
  - The value is in the format of a virtual relative path (starting with ~/). You can use the UrlHelper.Content method to convert the virtual path to an application absolute path.
  - If an alternative URL rewrite did not occur, the method returns an **empty string**.

Passing the culture code from routed pages

On multilingual sites, the correct thread culture for routed pages is supplied by the router (based on the culture of the requested page). If you wish to invoke custom actions from the context of a routed page – that is, you wish to call an action that maps to a custom route defined for your application – you need to ensure the thread culture of the handling thread matches the culture supplied by the router. Otherwise, when clicking links handled by custom actions, users may be redirected to a different language version of the page.

For this purpose, the system provides the **CultureCodeRouteValuesKey** property. The property can be set using a **PageRoutingOptions** object that is optionally provided as a parameter to the **UsePageRouting** method of the *ApplicationBuilder* instance (used to enable content tree-based routing on the side of the MVC application).

```
IApplicationBuilder.UsePageRouting(new PageRoutingOptions
{
        CultureCodeRouteValuesKey = "culture"
});
```

The *CultureCodeRouteValuesKey* property:

- Sets the name of the key under which the culture code of the current router-handled page is stored in the RouteValueDictionary object of the request.
- Enables you to retrieve the culture from route values in an IRouteHandler assigned to your custom routes. The handler sets the correct thread culture and thread UI culture for the processing thread.

See Setting up multilingual MVC projects for an example route handler that fulfills a similar purpose.