**How to migrate an EPiServer Composer site to EPiServer 7 CMS**

# Introduction

EPiServer 7 was release in October 2012 with new functionality in the CMS that allowed developers and content editors to divide their pages into smaller parts, a feature that was called Blocks. This feature matched in many ways one that was previously available through the use of a separate module called EPiServer Composer. But with version 7 it was integrated as a core CMS feature, making blocks a first class content type, something that didn’t allow it to be fully backwards compatible with Composer.

This document is describing how to migrate a site built based on Composer to a new solution based on 7 CMS. The migration process is broken down into two main parts, one about how to migrate the solution and one about migrating the content. Each step is made with the assistance of migration tools that has been made available by EPiServer.

We will assume that you already have general knowledge of EPiServer 7 CMS and the new concepts in this version. If not, it is recommended that you familiarize yourself with these by reading up on the topic and spending some time with the Alloy sample site provided with the release. It is further strongly recommended that you read the [Breaking Changes document in the SDK](http://sdkbeta.episerver.com/SDK-html-Container/?path=/SdkDocuments/CMS/7/Knowledge%20Base/Developer%20Guide/Upgrading/Breaking%20Changes.htm&vppRoot=/SdkDocuments//CMS/7/Knowledge%20Base/Developer%20Guide/) before starting the migration as this guide does not include any instructions on how to upgrade your solution from a CMS 6 to CMS 7 perspective.

Please note that customizations made to your Composer site may limit your success in following this process and using these tools.

The process described here is not an upgrade that is done in place over the current database, but a migration process that will bring content over to a new site. It requires manual steps and tasks that need to be completed by a developer. The migration process is based on utilizing the import/export functionality in CMS, which means that it has a few limitations in what content it includes, e.g. only published versions of content is included. See section below for a more comprehensive list of the limitations of the export/import process.

As the feature set in CMS 7 doesn’t completely match the one provided by Composer, there may be functionality that cannot be brought across to the new CMS site. Only features that have a direct equivalent in CMS are supported. See section below for a more comprehensive list of differences and migration limitations.

## System Requirements

* **EPiServer Composer 4 R2** - The migration tools and process have only been verified with Composer 4 R2. It is recommended that you upgrade the solution to this version before starting the migration.
* **EPiServer 7 CMS 7.1 Patch 3** - The target solution should be set up with CMS version 7.1 update with Patch 3. If you want to use a later version, you can upgrade your CMS after the migration has been completed.
* **.NET 4.0** - The computer where the code generation tool is run must have .NET 4.0 installed.

# Migration task workflow

The following list provides a brief overview and checklist of the steps required to complete the migration of a Composer site to CMS 7.

1. Export data from the Composer site
   1. Export Page types
   2. Export settings (Frames, Tabs, Categories and Visitor Groups)
   3. Export content (Pages and Dynamic Properties)
2. Create a new EPiServer 7 CMS solution
   1. Import settings package
   2. Configure site
   3. Install the EPiServer.ComposerMigration package
   4. Migrate users and groups (optional)
3. Migrate the solution
   1. Copy the old solution files and configuration.
   2. Generate Content Types classes and add to solution
   3. Refactor Composer templates
4. Migrate the content
   1. Import the Composer content package
5. Remove the EPiServer.ComposerMigration package.

# Migrating your solution

Migrating a Composer solution to CMS requires several tasks to be completed. This document aspires to describe the steps required by most solutions, additional step may however be necessary to migrate your specific solution.

The first step of the migration is to set up your new CMS 7 solution. You can either start with an empty solution and copy the files and configuration of your old solution that you want to keep or alternatively take your current solution and update it to CMS 7. In either case you should start with an empty database created through the EPiServer Deployment Center.

Once you have the new site running you need to configure it in the same way as your previous site. Some settings can be migrated using the export/import feature. Create an export package containing all settings that you want to bring across to the new site. This includes Frames, Tabs, Categories and Visitor Groups.

It is recommended that you import this package into your empty solution before adding any content types. This way you will ensure that all tabs are added correctly and that you don’t have to do any manual corrections afterwards.

Not all settings can be migrated this way, so you will have to do some configuration manually. If you have a globalized site, especially make sure that the website languages are setup in the same way. See the limitation section for a more complete list of settings that are not included when exporting and importing a site.

If your solution includes any custom property types these may need some modification to work in CMS 7. Please refer to EPiServer World for more information about changes to the interface in regards to custom properties.

## Content types

The recommended method to define page types in CMS 7 is to create attributed model classes in your solution. For block types this is the only method as you cannot create blocks from Admin mode. It will therefore be required to define all Composer Functions using model classes in your CMS solution.

To alleviate this process EPiServer has created a console application (cm.exe) that generates classes for your content types.

If you are already using Page Type Builder or related module to enable strongly typed page types it is recommended that you keep your existing page types and instead only update the syntax to match the CMS 7 one. In that case you might want to use the tool for some syntax hints. You can also limit the code generator to only generate block types.

First, create an export package that includes all page types from your Composer site including the Composer Function page types. If you want strongly typed page types, include these in the package as well.

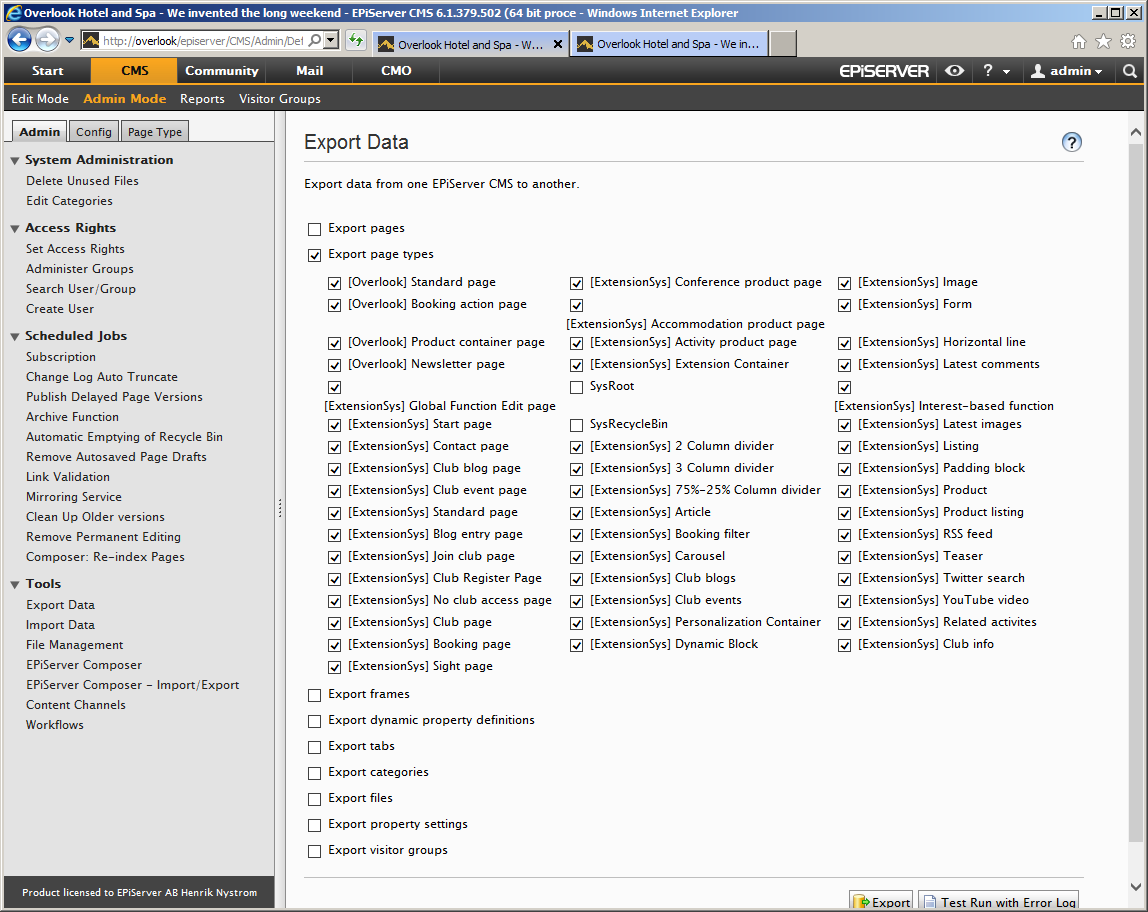


Figure - Export of all page types

Tip: double click the checkbox of the last page type to select all. The code generator will exclude the system page types automatically, so there is no problem if you happen to include these as well.

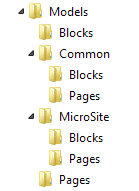
Then, run the console application, providing the location of the export file and it will generate content type class files for all Block and Page Types to be included in your solution. An exported page type is assumed to be a block and created as a block type if the template filename has an .ascx extension.

Figure Example of resulting folder structure

Pages and Blocks are by default placed in separate namespaces and folders. Types with a name using the de facto group naming standard, e.g. “[Group] Name” will be placed in a separate namespace/folder named after the group name.

You can provide arguments to the application to control the way that the classes are generated, including namespaces, type suffixes and imports. The table below describes all arguments that can be passed into the application. Note that all Boolean arguments doesn’t take a value and will be set to true if included.

|  |  |  |
| --- | --- | --- |
| **Argument** | **Default value** | **Description** |
| -s, --source | ExportedData.episerverdata | Path to the source package that you want to generate classes from. The path can be either relative or absolute. |
| -o, --output | output | Path to the folder where class files should be generated. Can be relative or absolute. |
| -n, --namespace | MySolution.Models | Base namespace that all generated model classes should be placed in. |
| --page-namespace | Pages | Namespace where all page type classes will be placed. Can be set to empty to place in the base namespace. |
| --page-suffix | Page | Suffix that all page type class names must end with. Can be set to empty if no suffix should be used. |
| -p, --page-base-class | EPiServer.Core.PageData | Base class that all page type classes will inherit from. |
| --block-namespace | Blocks | Namespace where all block type classes will be placed. Can be set to empty to place in the base namespace. |
| --block-suffix | Block | Suffix that all block type class names must end with. Can be set to empty if no suffix should be used. |
| -b, --block-base-class | EPiServer.Core.BlockData | Base class that all block type classes will inherit from. |
| -i, --imports | (none) | Namespaces that should be imported (using) in all classes. Separate each namespace with a space.  Note that some namespaces are added automatically by the tool if they are needed. |
| -g --group-namespaces | false | Indication if content types that has a group name prefix should include this name as a part of the namespace. |
| -w, --legacy-wrapper | false | Indication if properties of unknown types should get a UIHint to force a legacy property wrapper to appear. |
| -u, --nullable | false | Indication if properties of value types should be generated using their nullable equivalents. |
| --editable | false | Indication if properties not visible in edit mode should be generated using the Editable attribute rather than with ScaffoldColumn. |
| -?, --help |  | Displays this table of all arguments that can be provided. |

Table 1 - Command line arguments for Content type generator

Example usage:   
cm.exe -s “..\ContentTypes.episerverdata” -o "C:\Output\MySolution.Web\Models" -n "MySolution.Models" -i “MySolution.Utils” “MySolution.Core” -ugw

Names of content types and their properties are modified to be valid identifiers.

* Spaces and symbols are removed and Pascal casing is applied, e.g. “My article!” becomes “MyArticle”. Underscores are allowed.
* Numbers at the start of the name is replaced by its textual name, e.g. “2 Column page” becomes “TwoColumnPage”.
* Non-English Latin characters are replaced by their simplified visual equivalent where possible, e.g. ‘å’ is replaced by ‘a’. Some Latin characters and all non-Latin characters are not supported.

The name conversion takes a best effort approach and doesn’t guarantee that valid identifiers are generated. There are edge cases where already valid identifiers will be modified and other where invalid identifier names will be created. If some of your type or property names are not converted properly the recommendation would be to change these on the Composer site before starting the migration.

While you can change most other settings on your content types after the code generation is completed it is important that you do not change the Display Name of the Content types or any Property names before running the content import. The Display name is used by the CMS import functionality to match content with types and the property names to import the property data correctly.

All Block types are set to be available in edit mode as Composer Function page types are set to be unavailable to not be displayed in the Create page dialog.

Properties based on a value type are mapped to their equivalent type. If you prefer to use nullable types, you can provide the code generation tool with the -n argument and all standard value types will be changed to their Nullable equivalent. If you only want to use Nullable types for some specific properties you can change the property type manually after generation.

Composer content areas will be changed to CMS content areas. They are also changed to be Culture Specific and Visible in Edit mode by default. This can be changed on the model classes if desirable.

Function Picker properties as used by Dynamic Functions will be converted to Content Reference properties.

Limited support for Default values are added through the inclusion of a *SetDefaultValues* method. If you are using a default Archive page, Frame and/or Publishing date offsets you will have to include these manually.

The content type generation tool does not support the following settings in its current version:

* Available page types
* Access Rights on Block types

The tool only supports generation of C# classes.

## Templates

The next step is to update your page and block templates to work with CMS controls rather than the Composer specific ones. You will be able to continue to use your WebForms templates in the same format as they were set up with Composer. For most solution there are only a few steps required to get them working with CMS 7. You will need to update your Block templates as well as your Page templates that has any Composer content areas.

First of all you need to remove any references to Composer namespaces. These will all begin with “Dropit.Extension”. This includes any using and @Import statements.

### Pages

In the .aspx templates, replace all controls of type *ExtensionContentArea*with the standard EPiServer *Property*control. For the same control add a *PropertyName*attribute with the same value as the *ID*attribute or simply change the attribute name to *PropertyName*if you are not referencing the controls using this ID. The *Description*attribute can be removed completely as it is no longer used.

#### Example:

**Composer**   
<Extension:ExtensionContentArea ID=”SomeAreaName” Description=”Some description” />

**CMS 7**  
 <EPiServer:Property PropertyName=”SomeAreaName” />

If you are moving to strongly typed page types, change your template classes to inherit from *TemplatePage<T>*. If you are currently using your own base class for your page templates and for some reason cannot change the base class to inherit *TemplatePage<T>*, you can get around this by adding the *IRenderTemplate<T>* interface instead.

### Blocks

First of all, replace all usage of the Composer Property control with the standard EPiServer Property control. If you don’t mind keeping the Extension prefix around you can map it to the *EPiServer.Web.WebControls* namespace in the *EPiServer.Web.WebControls* assembly in web.config.

#### Example:

**Composer**  
<Extension:Property PropertyName=”SomeProperty” />

**CMS 7**  
 <EPiServer:Property PropertyName=”SomeProperty” />

If the block is a layout block, i.e. contains any content areas, you will need to do the same control replacement as you did on your page templates.

Next, change your template classes to inherit from *BlockControlBase<T>* instead of *BaseContentFunction*. If you are currently using a base class for your block templates and for some reason cannot change the base class to inherit *BlockControlBase<T>*, you can get around this by adding *IRenderTemplate<T>* interface in the same way as you did for your page templates.

If the namespace of your templates doesn’t match your folder structure, you will have to attribute the template with a *TemplateDescriptor* and provide it with the *Path* argument pointing out the location of your template.

Where your current template code accesses block properties using *ContentFunctionData*, update it to use *CurrentBlock* instead. It is recommended to use the strongly typed accessors to work with property values, but you can still access *PropertyData* objects and values through the *Property* collection. Be aware of that the indexer property of these two classes is implemented slightly different. Where *ContentFunctionData[“PropertyName”]* refers to the property object itself, *CurrentBlock[“PropertyName”]* is a reference to the value of the property.

#### Example:

**Composer**  
PropertyData property = ContentFunctionData[“PropertyName”];

PropertyData property = ContentFunctionData.Property[“PropertyName”];

object propertyValue = CurrentBlock.Property[“PropertyName”].Value;

**CMS 7**  
PropertyData property = CurrentBlock.Property[“PropertyName”];

object propertyValue = CurrentBlock[“PropertyName”];

object propertyValue = CurrentBlock.Property[“PropertyName”].Value;

If you have used the *IsEditMode* property on *BaseContentFunction* to know if the page is rendered in edit mode, you can get the same information from *EPiServer.Editor.PageEditing.PageIsInEditMode*.

### Dynamic Function Blocks

There is no feature in CMS that is direct equivalent to Dynamic Functions in Composer. To work around this all Composer Function Picker properties are converted into Content Reference properties. This will allow us to load the referenced block data and render it using its own template. The code snippet below shows an example on how the code behind of such a Dynamic Block template could look like. The corresponding .ascx file for this template is empty.

public partial class DynamicBlock : BlockControlBase<Models.Blocks.DynamicBlock>

{

protected override void OnInit(EventArgs e)

{

base.OnInit(e);

var contentRenderer = new ContentRenderer

{

CurrentData = Locate.ContentLoader().Get<BlockData>(CurrentBlock.FunctionPicker)

};

Controls.Add(contentRenderer);

}

}

The Composer API has additional functionality for working with Function Blocks and Content Areas that has not been described here. While some of that functionality does not have a direct equivalent in the CMS API there are likely other ways to solve the same problems. Please provide feedback if there are such areas that you believe are important for any Composer migration.

# Migrating your content

The second part of the migration is getting the Composer content migrated to the new CMS 7 site. This will be done by utilizing the import/export functionality in CMS together with a small plugin provided by EPiServer. The plugin will transform the content data coming from a Composer site to allow it to be imported as CMS Blocks.

## Exporting the content

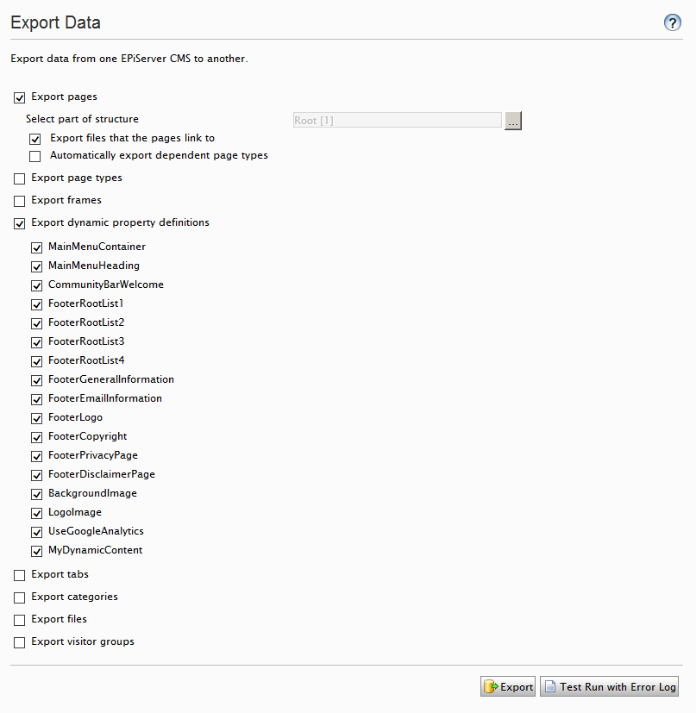
To get content from Composer we will need create an export package containing the main content, including all Pages and Dynamic Properties. Include all pages by selecting the root page as the part of the structure you want to export. If you want your files migrated as well, keep the “Export files…” option selected.

Figure - Main content export

If you for some reason want to break up your export into smaller packages and not just export everything off the root page you will have export your global blocks separately. This is done by using the dedicated Composer export tool. This tool will allow you to select the global blocks that you want to export.

## Importing packages

Before you start to import your content package, make sure that your solution is fully deployed before running the import. That includes any Visitor Groups, Content Types, Custom Properties and Dynamic Content definitions.

Start the import by selecting your package and then choosing the root page as the destination page where you want your page imported. If you have any dynamic or page properties that are set on the root page these will unfortunately not be imported and they will have to be set manually.

If you have broken down your export into several smaller packages you should always start with your global blocks package and then move on to your page packages.

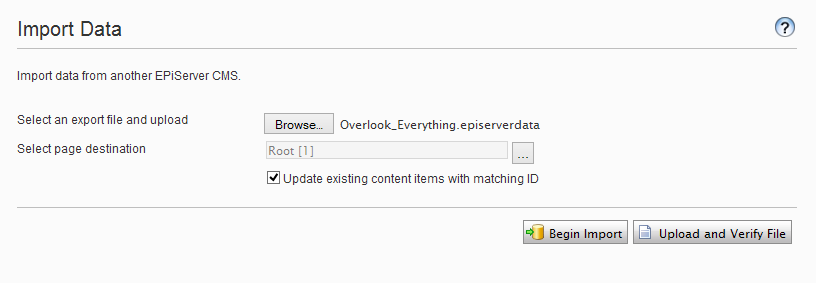
When running the import, make sure that you have checked “Update existing content items with matching ID”.

Figure - Content import

Once the import is started the migration module will scan the content and collect information about the items in the package. This includes block layout, personalization and language settings.

As the import starts to process each content item, the module will inspect and transform each Composer content item to match the new CMS 7 format. The following transformations will take place during this stage:

* Global blocks in Composer are placed directly in the Global block repository
* Composer Containers are replaced with Block Folders and placed in the Global block repository.
* Blocks are given a name based on their type plus the value of any heading property if found.
* Block folders are given the same name as the page it is holding blocks for.
* The language branch of local Blocks are reset to mirror the one of the page it was placed on.
* Composer Content Areas are changed into CMS Content Areas.
* The block layout is extracted from the parent page and recreated in CMS format in the right Content Area of the Page or Layout Block placed on the page.
* Personalization information is extracted from their Composer Container and applied to the Content Area data where this information is stored in CMS 7. Only the standard Composer Personalization blocks are supported.
* Composer specific properties and page types are excluded from the import completely.
* Page Type names are stripped of any Composer prefixes.
* Property name are modified to match valid identifiers as created by the Code Generator. Page type properties can be excluded from this transformation by configuring a setting as described below.
* Composer Function Picker properties are changed into Content Reference properties and their values converted to match the new type.

Please note that a Composer page that has not yet been published (regardless of publishing dates) will not include the block configuration and will therefore not be migrated properly. Blocks included on any unpublished pages will be included in the export package but the migration tool will filter these out unless configured otherwise. Note that if they are included, they are not guaranteed to be imported in the correct language or placed in a block folder name or associated with the page.

Migrated Global blocks will always have the primary language of the Composer site regardless of the language of the pages they are included on. If you want to a Global block to be available for another language you can translate the block manually using the CMS interface.

As all local block folders will be placed directly under the Global block folder, this result in a somewhat hard to manage structure, especially if there are a lot of pages using the same name. But once imported, there is nothing preventing the folders from being restructured. Note that with EPiServer 7.5 CMS every page will get a dedicated block folder which only will be visible when that particular page is selected.

The following configuration options can be provided to the migration module by adding an entry to the AppSettings collection.

|  |  |  |
| --- | --- | --- |
| **Key** | **Default value** | **Description** |
| ComposerMigration:Enabled | true | Set this to false to disable the migration module completely. |
| ComposerMigration: BlockNameProperties | Heading, Header, Headline, Title, Name | A list of properties that will be used to give imported blocks a friendly name. |
| ComposerMigration: TransformPageTypeIdentifiers | true | Set this to false if you don’t want the module to modify the names of properties on your page types. This can be useful if you haven’t generated model classes for your page types. |
| ComposerMigration: IncludeUnusedBlocks | false | Set this to true if you want to import blocks in the package that is not placed on any page. |

Table 2 - Import settings

# Limitations

## Export/Import

* Unpublished Composer pages will not include any content area information, e.g. block layout, as Composer only saves any layout information when the page is published.
* Only the published version of a page is included, any version history will be lost.
* Only files that are used on any page are included when exported unless manually selected.
* Data stored in the Dynamic Data Store must be programmatically included in the package for it to be brought across. Code showing how this can be done can be found in the [Code section of EPiServer World](http://world.episerver.com/Code/Paul-Smith1/Exporting--Mirroring-XForm-Posts/).
* Form definitions are included, but any form data collected must be exported manually from their respective pages.
* Any dynamic or page property values set on the root page will not be included in the import.
* Users and groups will not be migrated. If a local membership database is used this can be done by following a separate procedure such as the one [described here](http://www.leggetter.co.uk/2010/10/15/migrating-asp-net-membership-users-from-one-database-to-another.html).
* Many system settings will have to be re-configured. This includes but may not be limited to:
  + Scheduled job configuration
  + Website languages and access rights to these languages
  + Mirroring
  + Workflows
  + Content channels
  + Search configuration
* Some default value on page types will not be exported properly
  + Start and stop publish offsets
  + Archive page

## Differences between Composer and CMS

There are a number of features in Composer that have no direct equivalent in CMS 7. These features will not be supported by the migration tool either. For additional description of these features, see the Composer documentation on EPiServer World.

* Any **Composer Dynamic Properties** will be imported to a plain text property containing the raw xml data.
* **Composer templates** have no direct CMS equivalent and cannot be migrated.
* **Composer categories** are not used by the CMS and there is no reason to migrate these. If categorization of blocks are desired, use the group name setting of the Content Type attribute.
* There is no equivalent configuration to the feature **Available Blocks for Areas** in Composer and it will therefore not be migrated.
* Composer supports annotating **Global Blocks with Description and Notes**. CMS has no equivalent and these will not be migrated.

# Final note

As every implementation is different there can always be steps described here that doesn’t work for you or there are steps that has not been covered at all. If you have completed a Composer migration and have experienced this, feedback is always welcomed so that we can improve this guide to be as helpful as possible.

Good luck with your upgrade!