Migration process of module configuration and websites

Introduction to the migration

This page explains the migration of module configuration from 4.4 to 4.5. It explains modifications made to configuration of modules shipped with Magnolia CE and EE and your own modules. You only need to migrate modules that provide templating features (basic templating or STK-based templating). The goal of this process is to change module configuration so it adheres to the new structure. The structure of websites is also changed.

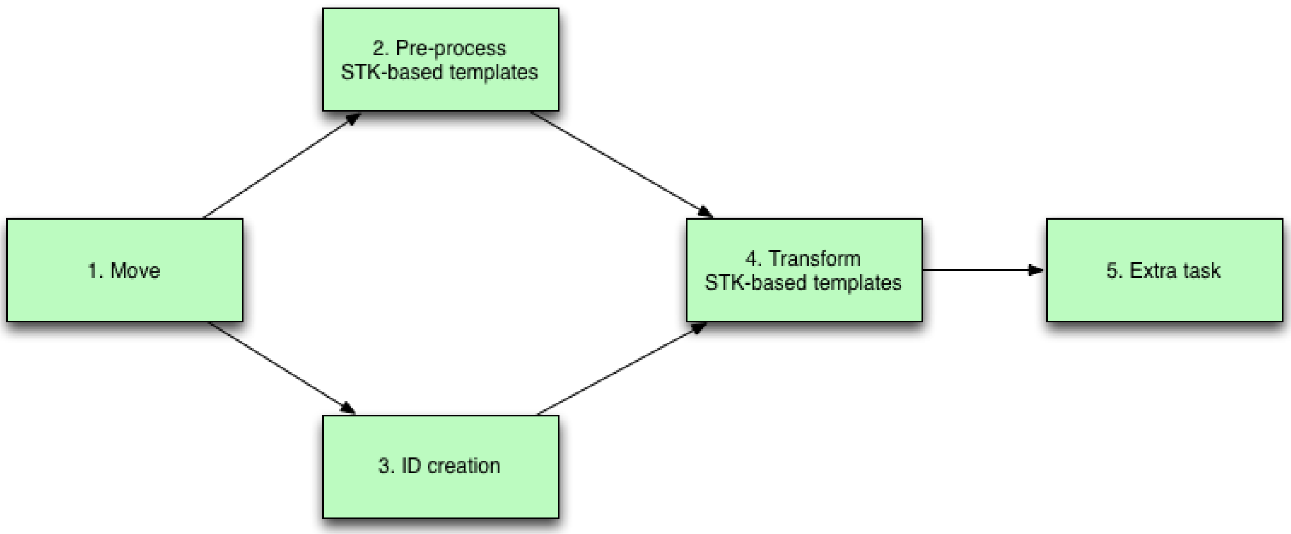
The process is made of several steps. Each step has its own goal. All the steps are represented by Groovy scripts. You can find all Groovy scripts in the module magnolia-4-5-migration.

Important:

- Run each migration step on all modules. For example, run the Move step on all your modules before proceeding to the ID Creation step.
- Run the migration process for Magnolia's module as well as your own modules. The best way to this is to add your module names to the collection of modules to be processed by each step. The step-by-step instructions below ask you to do this.

STK-based module

If your module is based on the Standard Templating Kit (STK) then follow this process.


2. Pre-process STK-based templates
1. Move
4. Transform STK-based templates
5. Extra task
3. ID creation

1. Move

Groovy class: `TemplatesAndParagraphsToPagesAndComponents`Workspace: `config`Mandatory: yes

The goal of this step is to move the templates and paragraphs into their new locations:

- Page templates to `config:<module name>/templates/pages`
- Component templates (former paragraphs) to `config:<module name>/templates/components`

If you want to run this step on your module, add your module's name to the collection `modulesWithPagesAndComponents` in the Migration. groovy script.

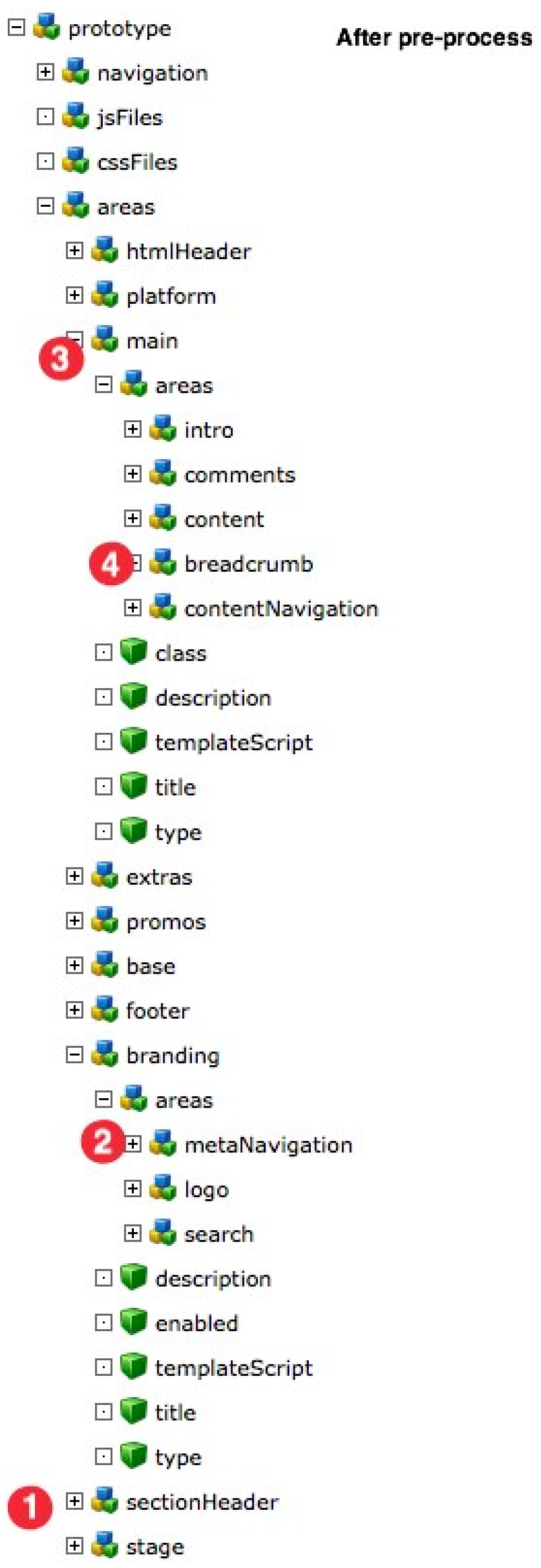
2. Pre-process STK-based templates

Groovy class: `PreProcessSTKTemplates`Workspace: `config`Mandatory: Yes
Dependency: Step 1: Move

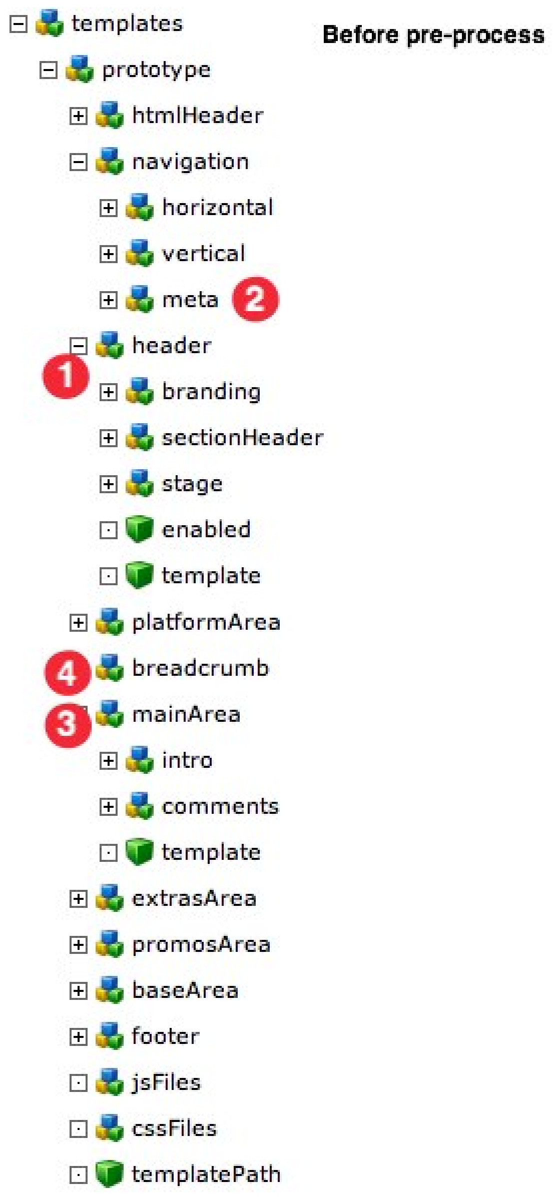
The purpose of this step is to prepare the templates for the transformation step. The step operates only on the templates folder, not on paragraphs. If your STK-based module does not have any page templates, for example it has only paragraphs, you can skip this step.

The step makes the following changes in the template definition:

1. Areas within the `header` (`stage`, `sectionHeader` and `branding`) are moved one level up and then the `header` area is removed.
2. MetaNavigation area is moved from navigation to branding.
3. The `main` area will contain a child area named `content` which will contain all former `main` area content.
4. The `breadcrumb` area is also a subarea of the `main` area.


After pre-process

```
⊟ prototype
  ⊞ navigation
  ⊡ jsFiles
  ⊡ cssFiles
  ⊟ areas
    ⊞ htmlHeader
    ⊞ platform
    ⊟ main  3
      ⊟ areas
        ⊞ intro
        ⊞ comments
        ⊞ content
        ⊞ breadcrumb  4
        ⊞ contentNavigation
      ⊡ class
      ⊡ description
      ⊡ templateScript
      ⊡ title
      ⊡ type
    ⊞ extras
    ⊞ promos
    ⊞ base
    ⊞ footer
    ⊟ branding
      ⊟ areas
        ⊞ metaNavigation  2
        ⊞ logo
        ⊞ search
      ⊡ description
      ⊡ enabled
      ⊡ templateScript
      ⊡ title
      ⊡ type
    ⊞ sectionHeader  1
    ⊞ stage
```


Before pre-process

```
⊟ templates
  ⊟ prototype
    ⊞ htmlHeader
    ⊟ navigation
      ⊞ horizontal
      ⊞ vertical
      ⊞ meta  2
    ⊟ header  1
      ⊞ branding
      ⊞ sectionHeader
      ⊞ stage
      ⊡ enabled
      ⊡ template
    ⊞ platformArea
    breadcrumb  4
    mainArea  3
      ⊞ intro
      ⊞ comments
      ⊡ template
    ⊞ extrasArea
    ⊞ promosArea
    ⊞ baseArea
    ⊞ footer
    ⊡ jsFiles
    ⊡ cssFiles
    ⊡ templatePath
```

If you want to run this step on your module you have to modify the Migration.groovy script.
Add your module's name to the collection `stkBasedModules` in the and the path of your site definitions to the collection `siteDef`.
Then in the step 7. on the main migration add a call to `preProcessSTKTemplates.doPreProcess(stkBasedModules, siteDef)`.

3. ID creation

Groovy classes:

- `BuildDialogIdMap`
- `MigrationUtil.buildPagesIdMapFromListOfModule`
- `MigrationUtil.buildComponentsIdMapFromListOfModule`Workspace: `config`Mandatory: Yes
  Dependency: Step 1, step 2

This step is represented by several Groovy scripts. The purpose is to build a map of dialog, template and component IDs. The ID is used to identify and reference the items.

Prior to 4.5, an item name such as `stkHome` was used as an ID. In 4.5, the name stays but we also create an ID for each item. The ID is of form `<module name>:<path to the item>`. The first part before the colon (😎 is the name of the module folder where the item definition resides. The second part is a relative path. In the case of pages and components, the path starts from under the `templates` without a leading slash. In the case of dialogs, the path starts from under the `dialogs` folder without a leading slash.

For example, STK components are in the Standard Templating Kit module so the first part is `standard-templating-kit`. The relative path starts with the `components` folder, followed by a path to the component content node, such as `components/extras/stkExtrasInternalPage`.

`<standard-templating-kit>:<components/extras/stkExtrasInternalPage>`

This step creates 3 maps containing this qualified name for the dialogs, templates and components. These maps are required by the Transform step. Example map:

```
[
"stkHome" : "standard-templating-kit:pages/stkHome"
]
```

If you want to run this step on your module, add your module's name to the collection `modulesRelevantForIDCreation` in the Migration. groovy script.

4. Transform STK-based templates

Groovy class: `TransformSTKTemplates`Workspace: `config`Dependencies: Step 2, step 3
Mandatory: Yes

Now we are ready for the main migration. This script is the most important script of the migration, it's really migrating from the old world to the new world. The goal of this step is to transform every STK-based template definition (prototype, concrete template and paragraphs). The templating changes introduced in 4.5 are listed in LINK TO OFFICIAL DOCUMENTATION OR RELEASE NOTES.

The entry point is `TransformSTKTemplates.migrateSTK`, it runs the migration of over all templates, paragraphs and prototypes.

The migration of a template is done by `TransformSTKTemplates.migrateTemplateDefinition`. It works the same way with a template and with a paragraph, because in the new definition page templates and component templates represent the same definition.

If you want to run this step on your module you have to modify the Migration.groovy script.
Add your module's name to the collection stkBasedModules in the and the path of your site definitions to the collection siteDef.
Then in the step 9. of the main migration add a call to `transformSTKTemplates.migrateModulesAndSiteDef(stkBasedModules, siteDef)`.

Below is explained the logic of this step:

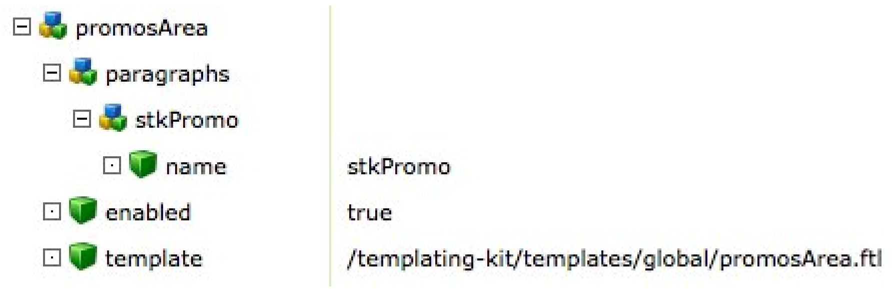
Properties
Update properties to match the new convention. For example, the template `type` property is now `renderType`. We update the name of the property, its value or both. See `MigrationUtil#updateTemplatePropertiesGeneric`.

Areas
Create areas. Some tasks involved are:

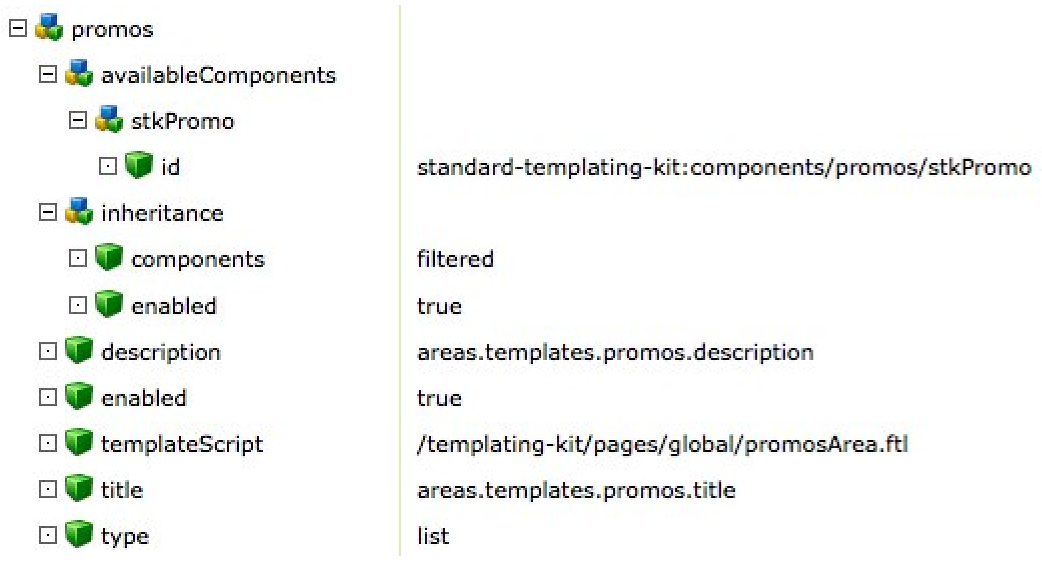
1. Create an `areas` node. This node contains the subareas.
2. Repeat the process of updating properties for areas.
3. If the area contains a `paragraphs` node, it becomes `availableComponents`.
4. Update component IDs with their qualified IDs.
5. Create a new property that defines the `type` of the area: `list`, `single`, `noComponent`.
6. Create a new `optional` property if the area is optional such as `stage`.

7. If an area has autoGeneration or inheritance, trigger special method. See below.
8. Then we execute the same process on the area's children. The list of node becoming areas is found in TransformSTKTemplates#getSTKAreaToTransform and is "htmlHeader","branding","sectionHeader","stage","platformArea","breadcrumb"," mainArea","extrasArea","promosArea","baseArea","footer","meta".

The screenshot below shows the difference in the `promos` area.



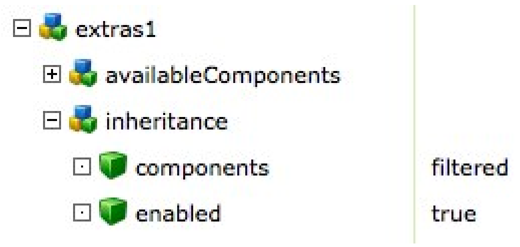
```
⊟ promosArea
   ⊟ paragraphs
      ⊟ stkPromo
         ⊡ name          stkPromo
   ⊡ enabled             true
   ⊡ template            /templating-kit/templates/global/promosArea.ftl
```



```
⊟ promos
   ⊟ availableComponents
      ⊟ stkPromo
         ⊡ id                standard-templating-kit:components/promos/stkPromo
   ⊟ inheritance
      ⊡ components           filtered
      ⊡ enabled              true
   ⊡ description             areas.templates.promos.description
   ⊡ enabled                 true
   ⊡ templateScript          /templating-kit/pages/global/promosArea.ftl
   ⊡ title                   areas.templates.promos.title
   ⊡ type                    list
```

Inheritance
Inheritance is now directly configurable in the definition. An area inheriting content from its parent has a node `inheritance` with the property `enabled` set to `true`.
The `components` property can have values `all`, `filtered` or `none`.

- If value is `all`, all the components from the parent are inherited.
- If the value is `filtered`, only components where `inheritable` property is `true` are inherited.
- If the value is `none`, no component is inherited.

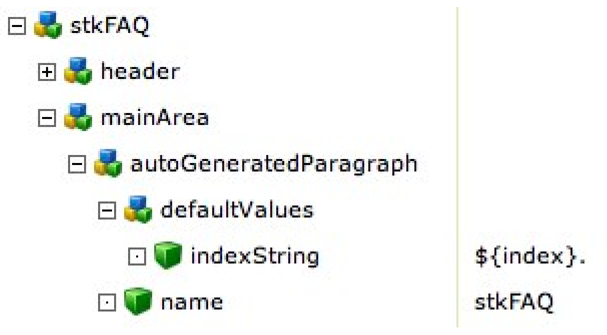


```
⊟ extras1
   ⊞ availableComponents
   ⊟ inheritance
      ⊡ components      filtered
      ⊡ enabled         true
```

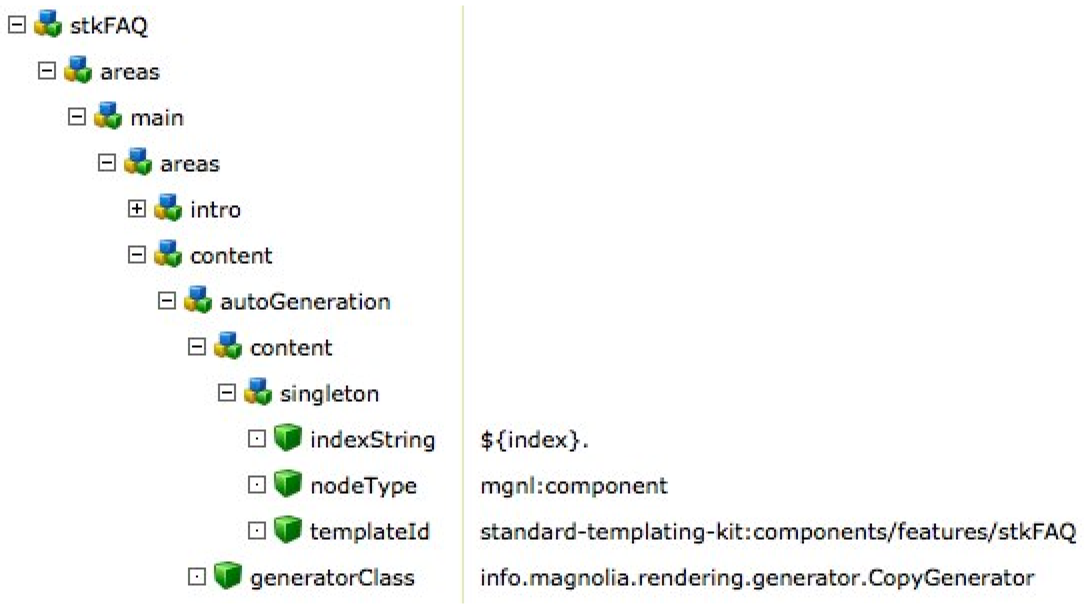
See `TransformTemplate#createInheritance`.

AutoGeneration
An area (and even a page) can add an `autoGeneration` node in its configuration. It replaces the former configuration `autoGeneratedParagraph` but provides a much more powerful mechanism. Before you were able to create only one "single" (singleton) paragraph. But now you can create complex content: a page, an area or a component. Everything inside the `content` node will be created.

In the screenshot you can see that properties that used to be in the `defaultValues` node of an `autoGeneratedParagraph` are now in the `singleton` `node`. This node singleton represents the structure of the content to create in the main area of a FAQ page.



```
stkFAQ
    header
    mainArea
        autoGeneratedParagraph
            defaultValues
                indexString        ${index}.
            name                   stkFAQ
```



```
stkFAQ
    areas
        main
            areas
                intro
                content
                    autoGeneration
                        content
                            singleton
                                indexString     ${index}.
                                nodeType        mgnl:component
                                templateId      standard-templating-kit:components/features/stkFAQ
                        generatorClass          info.magnolia.rendering.generator.CopyGenerator
```

5. Extra task

Workspace: `config` or `website`Dependencies: Step 4
Mandatory: No
Groovy class: Provide your own.

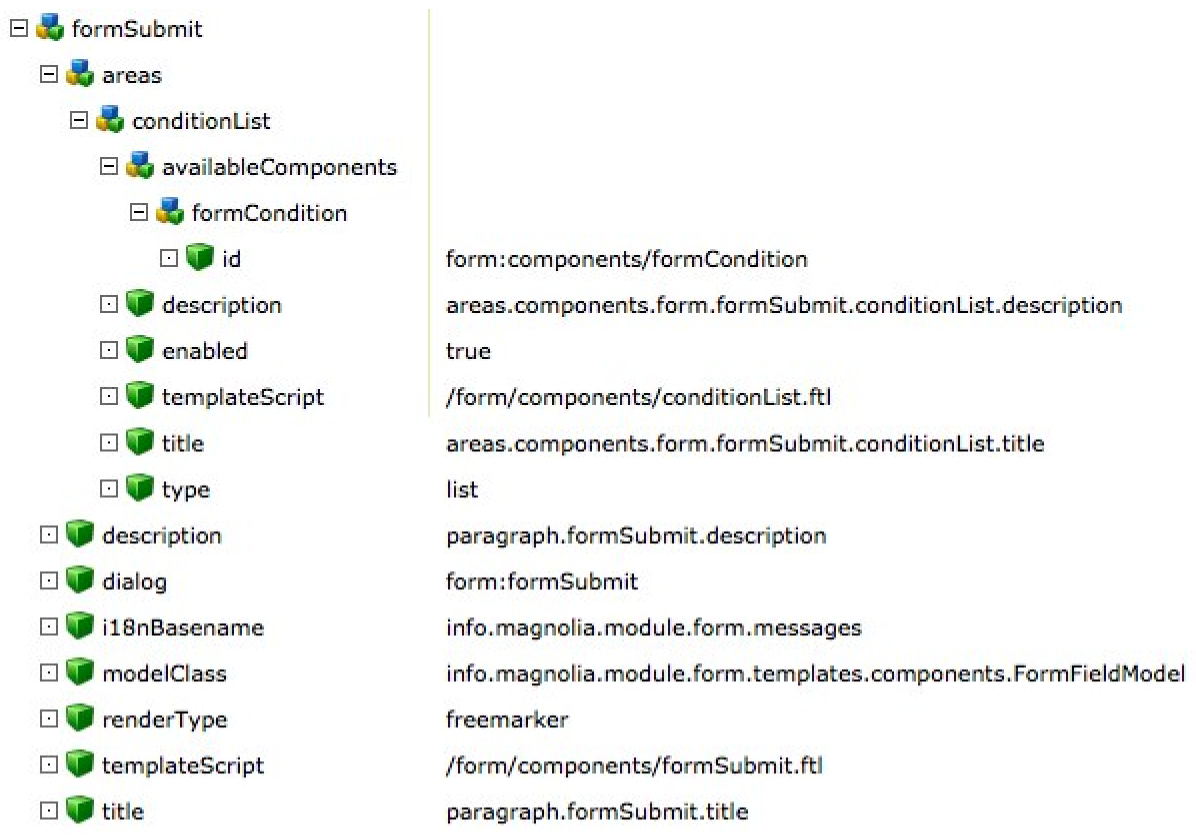
An extra task could remove inconsistency, like for example if a dialog is not properly named and all other stuff like that.
This step has to be executed at the very end of the process.

Here is an example of an extra task that has to be performed on the Form module configuration. We have to configure areas whenever we find a contentNodeIterator in the template script.

```
[@cms.contentNodeIterator contentNodeCollectionName="conditionList"]
   [@cms.includeTemplate /]
[/@cms.contentNodeIterator]
[@cms.newBar contentNodeCollectionName="conditionList" paragraph="formCondition" /]
```

The contentNodeCollectionName becomes the name of the area. The list of paragraphs in the `paragraph` attribute becomes the `availableComponents` list. Since this is an iterator that iterates through a collection of multiple item, the area is of type `list`. For the `templateScript` property you can use the generic area script `/magnolia-module-standard-templating-kit/src/main/resources/templating-kit/generic/listArea.ftl`. This generic script works if the iterator does not write any static HTML elements. If your iterator does, write a custom area script instead.

The resulting area configuration looks like this:



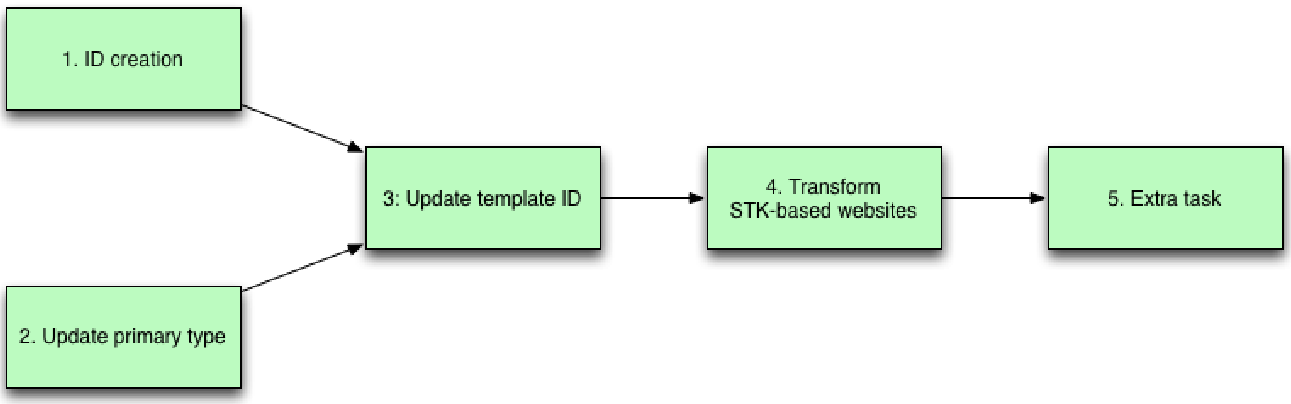
```
formSubmit
  areas
    conditionList
      availableComponents
        formCondition
          id                  form:components/formCondition
      description             areas.components.form.formSubmit.conditionList.description
      enabled                 true
      templateScript          /form/components/conditionList.ftl
      title                   areas.components.form.formSubmit.conditionList.title
      type                    list
  description                 paragraph.formSubmit.description
  dialog                      form:formSubmit
  i18nBasename                info.magnolia.module.form.messages
  modelClass                  info.magnolia.module.form.templates.components.FormFieldModel
  renderType                  freemarker
  templateScript              /form/components/formSubmit.ftl
  title                       paragraph.formSubmit.title
```

To complete this extra task, change the contentNodeIterator tag in the template script to a `cms.area` tag.

If you want to run this step on your module, create your own groovy script providing extra transformation, then add a call to your script in `Migration.groovy`.

STK-based website

An STK-based website is a site whose pages and paragraphs are based on STK templates. This process is executed in the `website` workspace.



1. ID Creation

Perform the ID Creation step as you would for an STK-based module.

2. Update primary type

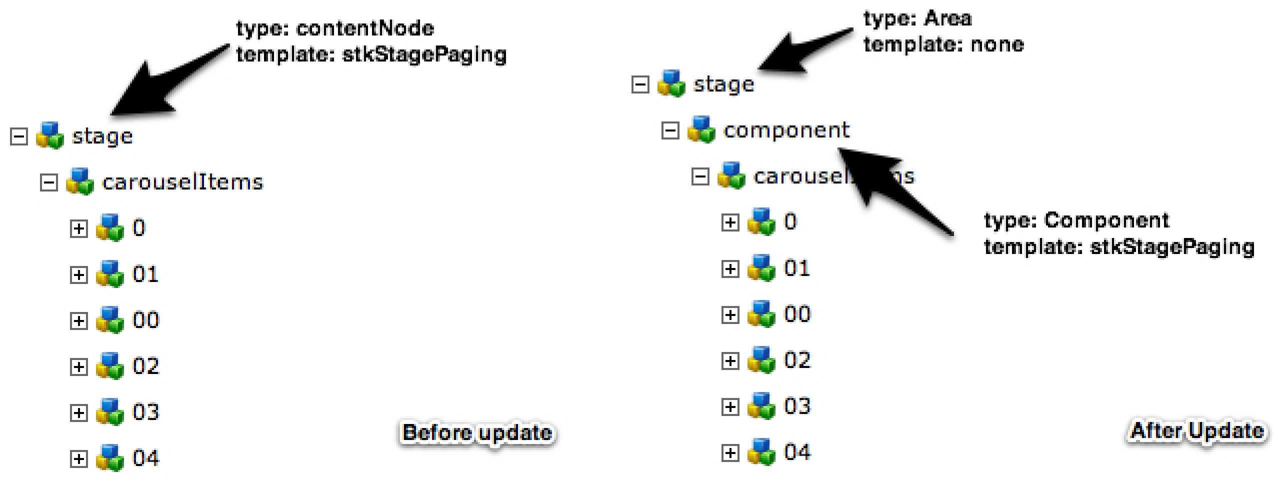
Workspace: `website`Groovy class: `UpdateWebsitePrimaryType`Mandatory: Yes
Dependency: none

The goal is to change the primary type of each node of a website. New types are Page (mgnl:page), Area (mgnl:area), Component (mgnl:component).

Logic:

- `Content (mgnl:content)` becomes `Page`
- `Content Node (mgnl:contentnode)` becomes `Area` if the current node has no property "template"
- `Content Node (mgnl:contentnode)` becomes `Component` if the current node has a property "template"

Special case - single areas:
For areas of type `single` (e.g. `stage`, `opener`) an extra task is run. These areas must have node type `mgnl:area`, but since they contain a template, the logic will transform them into node type `mgnl:component`, which is wrong. To correct this, the script creates an "intermediate" node between the single Area and its children. This intermediate node is of type `mgnl:component` and its associated template is the template that the single Area used to have.



If you want to run this step on your module, add your website's name to the collection `website` in the Migration.groovy script.

3. Update template ID

Workspace: `website`Dependency: Step 1, step 2
Mandatory: Yes
Groovy class: `UpdateMetaDataTemplateId`

The goal of this step is to change the template IDs of the content. See STK-based templates > 3. ID Creation on how the template IDs have changed in 4.5 and what the new IDs look like.

```
<sv:property sv:name="mgnl:template" sv:type="String">
    <sv:value>stkSection</sv:value>
</sv:property>
```

The new value of a template is the qualified name found in the maps of IDs. In this example `stkSection` is replaced by `standard-templating-kit:pages/stkSection`.

If you want to run this step on your module, add your module's name to the collection `website` in the Migration.groovy script.

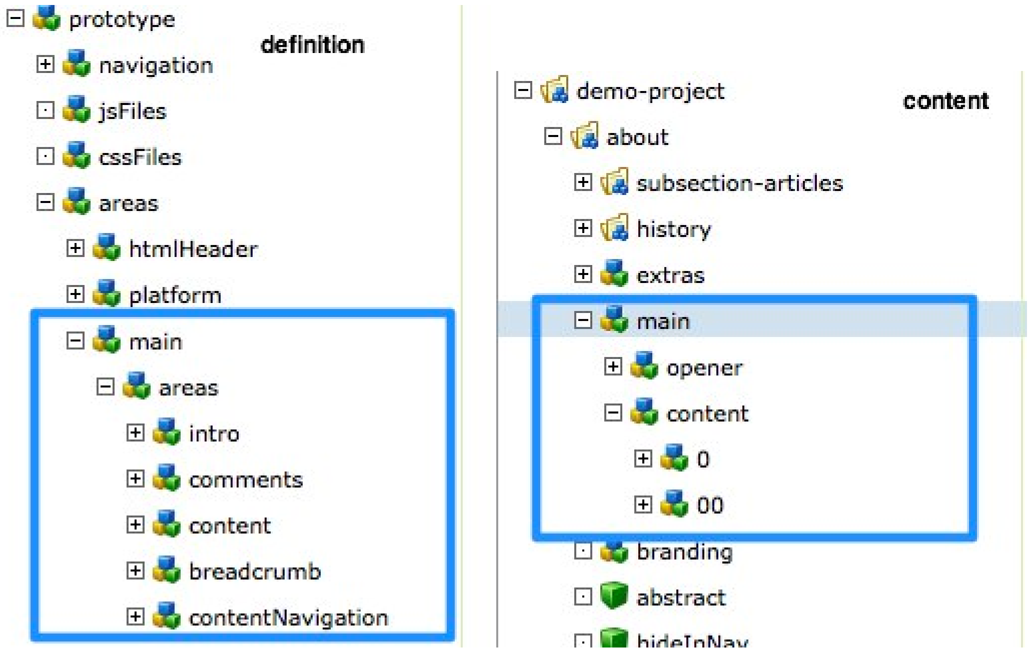
4. Transform STK-based websites

Workspace: `website`
Dependency: Step 3
Mandatory: Yes
Groovy class: `TransformSTKWebsites`

As we saw in the step 7, the main structure of STK definition has been revised, we moved some areas. The content must also reflect this change, for example the `opener` node must be within the `main` node.

We also introduced a new area within the `main` called `content`. The `content` area behaves like `main`, meaning it is the main placeholder for the content of a page. We have to create this node in the website content structure and move all the former content from the `main` area into it. Note that we dont move the `breadcrumb` content node because there was no such node in the content. The same is true for `contentNavigation` and `intro`.

The screenshot below shows how the website node structure now adheres to the new template configuration.



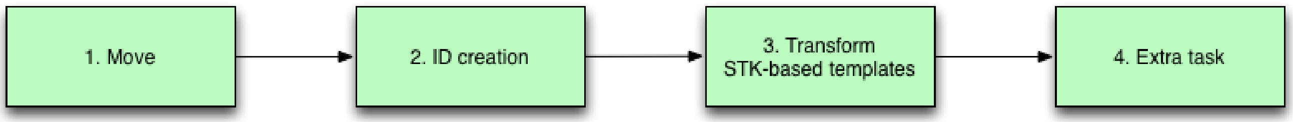
If you want to run this step on your module, add your module's name to the collection `website` in the Migration.groovy script.

5. Extra task

Perform an Extra task step as you would for an STK-based module if needed.

Basic templating-based module

If your module is based on the basic templating framework but does not use the Standard Templating Kit (STK) then follow this process.



1. Move

Perform the Move step as you would for an STK-based module.

2. ID Creation

Perform the ID Creation step as you would for an STK-based module.

3. Transform simple

Workspace: `config`Dependency: Step 2 ID Creation
Groovy class: `TransformSimpleTemplates`

This step does minimal changes:

- Properties are updated
- Dialogs are updated

If you want to run this step on your module, add your module's name to the collection `transformSimpleTemplatesInModules` in the Migration.groovy script.

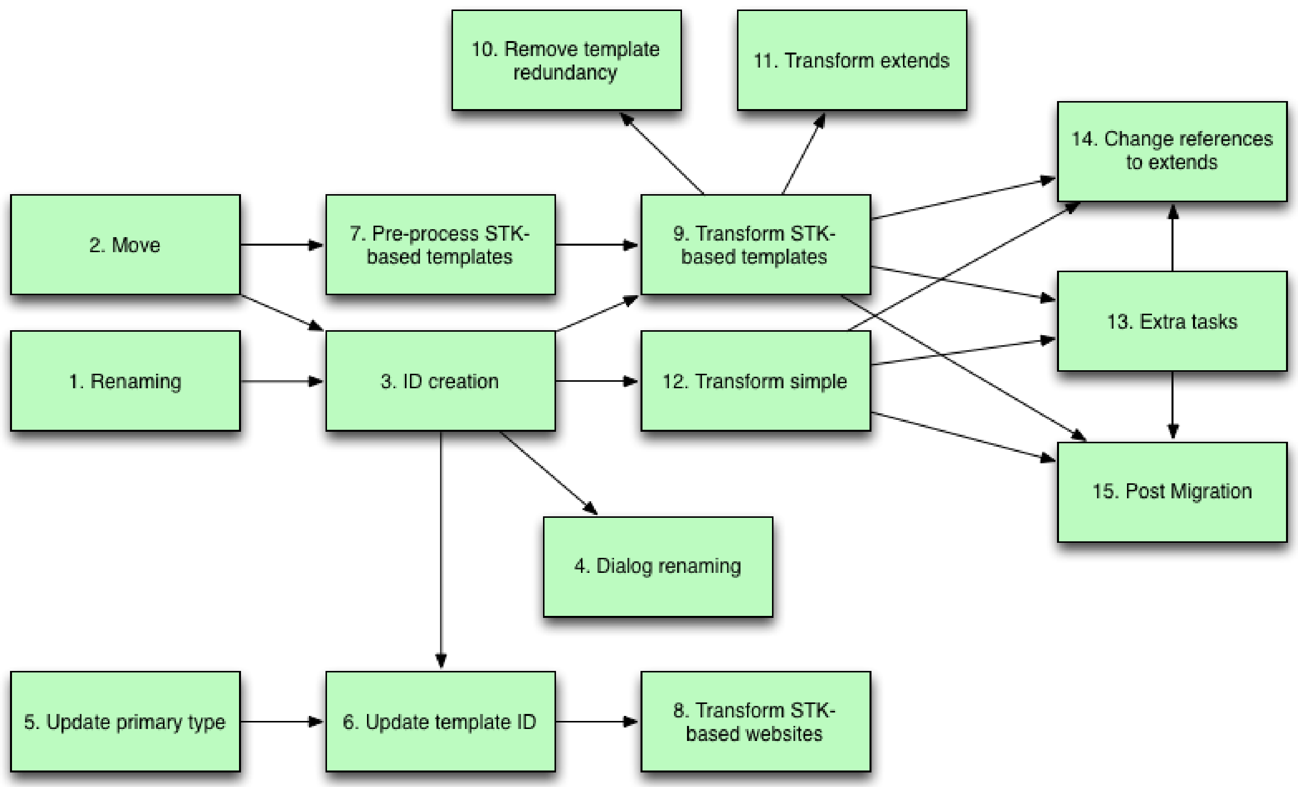
4. Extra task

Perform an Extra task as you would for an STK-based module.

Magnolia modules migration

In this part is detailled the migration of Magnolia modules.
The following modules providing STK-based templates and basic templating-based templates are transformed during the migration:
Standard Templating Kit, Form, Public User Registration, Commenting, Categorization, RSS Aggregator, Forum, Resources.

The diagram below represents the steps of the migration process.



1. Renaming

Groovy class: `PathChangesImpactingID`Workspace: `config`

It's goal is to rename nodes in order to reflect the new naming convention.

If you take a look a the method `PathChangesImpactingID.mapOfNodesToRename`, in `dialog` the `paragraphs` folder is renamed `components`.

If for the migration of your module you want to rename something, you could take this step as example.

2. Move

Perform the Move step as you would for an STK-based module.

3. ID creation

Perform the ID Creation step as you would for an STK-based module.

4. Dialog renaming

Groovy class: `RenameDialogsAndChangeInDialogsMap`Workspace: `config`Dependency: Step 3

This step renames the dialogs and updates the dialog IDs provided by the Move step.

Why we do it now and not in the step 1?
By renaming the dialogs and updating their IDs we still keep the reference (key) of these dialogs. Later we are able to update the dialogs in all the templates.

5. Update primary type

Perform the Update primary type step as you would for an STK-based website.

6. Update template ID

Perform the Update template ID step as you would for an STK-based website.

7. Pre-process STK-based templates

Perform the Pre-process STK-based templates step as you would for an STK-based module.

8. Transform STK-based websites

Perform the Transform STK-based websites step as you would for an STK-based website.

9. Transform STK-based templates

Perform the Transform STK-based templates step as you would for an STK-based module.

10. Remove template redundancy

Groovy class: `RemoveTemplateToPrototypeRedundancy`Workspace: `config`Dependency: Step 9

After the migration of the templates it could happen that a property has the same value on the prototype and on a template.
When a property has the same value in the prototype and in a concrete template, this property is defined as redundant and could be removed.
The goal of this script is to remove this redundancy. When it find a property with the same value on the prototype and on the template, it removes the property from the template.

If you want to execute this step on your module, check `RemoveTemplateToPrototypeRedundancy.doOnModules`.

11. Transform extends

Groovy class: `TransformExtends`Workspace: `config`Dependency: Step 9

This step introduces an important and handy feature of STK: extends. With this you can simply extend an already existing dialog / component / page definition and insert extra controls or labels. STK as is has a lot of duplicate entries since the extends mechanism was introduced later. This is an attempt of cleaning up these duplicates in an automated way.
This step also creates so called masters that can be extended from various places. Masters are like containers that you can quickly extend in order to create a working dialog. Items that are added to the masters are automatically added to all configurations that extend it.
i18n messages are also resolved and compared with the master values - if they are equal they are deleted to avoid unnecessary redundancy.

extends-override
Sometimes a configuration is almost the same but has one control less for example. In this case you still might want to extend the master and simply override the control that you don't need. There are a few situations where a simple property called **`extends`** with value **`override`** is not enough or not the correct way to do it:

- ~~Dialogs expect nodes on the very first level to be tab controls (`controlType=tab`). In order to get rid of a complete tab, you will need the extends=override as well as controlType=tab. Otherwise the dialog rendering will stop there (no errors) and simply not continue adding new tabs /controls or modifiying others. If you want to get rid of the very first tab, you might even end up with an empty dialog!~~ This was fixed in revision 51235, if a tab control has no `controlType` it will simply be ignored now and rendering passes on to the next tab, if there is one.
- If a template has a list of available components and you don't want to have these available in the extending configuration, extends=override would work, but there's a better way to do this. Instead of extends=override you can use enabled=false.
- Controls in dialogs can be simply excluded by using extends=override. No need to include the controlType.

12. Transform simple

Perform the Transform simple step as you would for an basic templating-based module.

13. Extra tasks

Perform the Extra task step as you would for a module.

In our case we perform extra tasks on `form`, `public-user-registration` and `resources`.

14. Change references to extends

Groovy class: `ChangeReferencesToExtends`Workspace: `config`Dependency: Step 9, step 12, step 13

This steps simply update all the `reference` property found in the dialogs by `extends`.

15. Post Migration

Groovy class: `PostMigrationProcessing`Workspace: `config`Dependency: Step 9, step 12, step 13

We could assimilate this step to an extra task of the migration of Magnolia. We use this step to do a cleanup by removing useless things, renaming values and all these kind of tasks.