Toolkit Dokumentation

Release 3.0

netzmacht David Molineus

Aug 15, 2022

Contents

Welcome to the developer toolkit documentation for the Contao Content Management System. Toolkit is developed by netzmacht David Molineus and is heavily used in extensions provided by netzmacht.

This documentation explains the provided tools which should simplify the development of Contao extensions even then simultaneously increase the code quality.

CHAPTER 1

Introduction

I develop Contao extensions for many years. Although Contao provides a useful library some aspects which are required in the daily life are missing. Furthermore some developer concepts which has grown in popularity, especially dependency injection, wasn't really usable in Contao. Working in projects which have high quality standards some improvements were required.

1.1 Goals

Developing and providing Toolkit was made with following goals in mind:

- Provide often required features
- Increase code quality
- Create testable code
- Fasten development

1.2 Developer tools

Toolkit does not replace existing developer tools which are provided by other Contao developers. It rather complement and uses the features provided by the other tools.

contao-haste The haste libary is one of the more comprehensive developer tools for Contao developers. There are some intersections between Toolkit and haste but the main focus is different.

The dependency-container, event-dispatcher and translator got obsolete since Contao 4 based on Symfony.

1.3 Examples

If you want to see how Toolkit is used in real contao extensions you can have a look at following examples:

- contao-leaflet-maps
- contao-form-validation
- contao-timelinejs

CHAPTER 2

Components

Contao uses content elements and frontend modules to generate content. Toolkit uses the term components to describe them. Toolkit allows to develop lightweight components. They does not require to extend from the Contao class stack. Toolkit also provides a way using dependency injection for components.

2.1 Content element

Focusing on the main goal for Toolkit there is an easy way to create lightweight decoupled content elements.

Hint: If you're developing for Contao 4.5 you might look at the **'fragments support'_**. If you have to stay on Contao LTS 4.4 you maybe want to read further.

2.1.1 Interface ContentElement

The only requirement for an content element supported by Toolkit is that the interface ContentElement is implemented. It extends the Component interface.

The Main difference between Contao content element and the interface is that the data attributes has to be accessible by the explicit *get* and *set* methods. Magic *__get* and *__set* are not recommended.

2.1.2 Register a content element

Since Toolkit supports dependency injection for content elements you have to create factories which creates them. A factory has to implement the ComponentFactory interface and must be registered as a tagged service.

Provided tags:

netzmacht.contao_toolkit.component.content_element_factory Tag your factory service with this tag so that toolkit will use it to create the elements.

netzmacht.contao_toolkit.component.content_element Tag your factory with this tag to provide information about the supported content elements. You have to define the *category* and *type* attribute as well.

```php
<?php

// src/ExampleFactory.php
class ExampleFactory implements Netzmacht\Contao\Toolkit\Component\ComponentFactory
{
    // ...
}

// src/Example.php
class Example implements␣
↪Netzmacht\Contao\Toolkit\Component\ContentElement\ContentElement
{
    // ...
}
```

```
// services.yml
foo.content_element.factory:
    class: ExampleFactory
    tags:
        - { name: 'netzmacht.contao_toolkit.component.content_element', category:
↪'texts', type: 'example' }
```

Hint: It's possible to create multiple types with a factory. Just add multiple tags.

You don't have to register you content element in the *config.php*. Toolkit will do it for you.

2.1.3 Extending AbstractContentElement

To simplify creating a new content element implementing the provided interface you can use the AbstractContentElement which itself is a subclass of AbstractComponent.

There are several extension points where you can hook customize the behaviour. Some of the methods are just empty placeholders which doesn't have to be called when being overriden.

$templateName The name of the template. Same as *strTemplate* in Contao

deserializeData(array $row) Method deserialize the given raw data coming form the database entry or model. You should call the parent method when overriding this one. Deserialization of the headline is done here.

isVisible() Is called to decide if content element should be generated. You should call the parent to keep the default behaviour.

compile() Is called before the template data are prepared.

prepareTemplateData(array $data) Prepares the data which are passed to the template.

postGenerate($buffer) Is triggered after the content element is parsed.

2.2 Frontend module

Focusing on the main goal for Toolkit there is an easy way to create lightweight decoupled frontend modules.

Hint: If you're developing for Contao 4.5 you might look at the **'fragments support'_**. If you have to stay on Contao LTS 4.4 you maybe want to read further.

2.2.1 Interface Module

The only requirement for an frontend module supported by Toolkit is that the interface Module which extends the Component interface.

The Main difference between Contao frontend module and the interface is that the data attributes has to be accessible by the explicit *get* and *set* methods. Magic *__get* and *__set* are not recommended.

2.2.2 Register a frontend module

Since Toolkit supports dependency injection for content elements you have to create factories which creates them. A factory has to implement the ComponentFactory interface and must be registered as a tagged service.

Provided tags:

netzmacht.contao_toolkit.component.frontend_module_factory Tag your factory service with this tag so that toolkit will use it to create the elements.

netzmacht.contao_toolkit.component.frontend_module Tag your factory with this tag to provide information about the supported content elements. You have to define the *category* and *type* attribute as well.

```php
<?php

// src/ExampleFactory.php
class ExampleFactory implements Netzmacht\Contao\Toolkit\Component\ComponentFactory
{
    // ...
}

// src/Example.php
class Example implements Netzmacht\Contao\Toolkit\Component\Module\Module
{
    // ...
}
```

```
// services.yml
foo.frontend_module.factory:
    class: ExampleFactory
    tags:
        - { name: 'netzmacht.contao_toolkit.component.frontend_module', category:
↪'texts', type: 'example' }
```

Hint: It’s possible to create multiple types with a factory. Just add multiple tags.

You don’t have to register you frontend module in the *config.php*. Toolkit will do it for you.

2.2.3 Extending AbstractModule

To simplify creating a new frontend module implementing the provided interface you can use the AbstractModule which itself is a subclass of AbstractComponent.

There are several extension points where you can hook customize the behaviour. Some of the methods are just empty placeholders which doesn’t have to be called when being overriden.

$templateName The name of the template. Same as *strTemplate* in Contao

$template frontend module template implementing Template

$renderInBackendMode If true the module is generated in the backend. Otherwise the *be_wildcard* placeholder is generated. Default is *false*.

deserializeData(array $row) Method deserialize the given raw data coming form the database entry or model. You should call the parent method when overriding this one. Deserialization of the headline is done here.

compile() Is called before the template data are prepared.

prepareTemplateData(array $data) Prepares the data which are passed to the template.

postGenerate($buffer) Is triggered after the frontend module is parsed.

generateBackendView() Is used to generate the backend view if $renderInBackendMode is false.

generateBackendLink() Is triggered to create the backend edit link.

2.3 Hybrid

2.3.1 The interface

The hybrid is a special kind of component. It's interface inherits from the interfaces of *Content element* and *Frontend module* both. Components implementing the interface can be used as frontend modules and content elements at the same time.

2.3.2 Extending AbstractHybrid

Contao also knows the concept of hybrids but in Contao hybrids are a bit limited. The assume that a foreign table is used for the actual content. It's used for frontend forms in Contao, for instance.

The AbstractHybrid does not autoload any foreign dataset for you. instead it combines the behaviour of the abstract content element and the abstract module described the the documentation. Each customization point is available here as well.

Hint: If you're developing for Contao 4.5 you might look at the fragments support. If you have to stay on Contao LTS 4.4 you maybe want to read further.

CHAPTER 3

Data

Toolkit provides data related tools. There is a repository implementation deal with issues of the Contao models. Also tools for generating an alias or update a dca based dataset is provided.

3.1 Repositories

Contao uses it's own implementation for Models. Even though it's called models the pattern behind it is rather the active record pattern. The main issue of this implementation is the hidden access to the database connection. Implementing rich models and writing tests is really painful.

To overcome this issue I recommend to use the repository pattern. Toolkit provides some tools to ease using repositories within Contao. I recommend to provide interfaces for each repository you have to create. The real implementation can be easily switched now.

3.1.1 Repository manager

The repository manager is designed to provide a simple access to all model repositories. If you want to use a custom repository for your model, you have to register it. If no repository is registered, a default repository is returned.

Getting a repository from the manager

```php
<?php

// Provided as service "netzmacht.contao_toolkit.repository_manager"
$repository = $repositoryManger->getRepository(\Contao\ContentModel::class);
```

Register a custom repository

You can provide a custom repository by simply tagging a service with the tag *netzmacht.contao_toolkit.repository*. The service has to implement the `Netzmacht\Contao\Toolkit\Data\Model\Repository` interface.

```
// your services.yml
services:
   custom.repository.example_model:
      class: Custom\Model\ExampleRepository
      tags:
         { name: "netzmacht.contao_toolkit.repository" model:
↪"Custom\Model\ExampleModel" }
```

If you register a custom repository the model you specified in the model tag is automatically added to Contao's `$GLOBALS['TL_MODELS']`.

Warning: You should **never** define repositories for 3rd party models unless you are not really sure what you're doing.

3.1.2 Default repository

Toolkit includes an base Repository interface. It simply provide an interface for Contao common find* and count* methods. Furthermore it introduces Specifications. The implementation of the interface is ContaoRepository which simply delegates all methods to the model class.

Hint: Instead of implementing the Repository interface it's recommend to introduce own interfaces describing each required method. It doesn't matter if the provides ContaoRepository is used as the implementation as your code only trust the interface.

3.1.3 QueryProxy

If you want to access all magic finders and callers of your model class from the repository, you can add the QueryProxy trait to your repository. The default repository already uses it.

3.1.4 Specification

Toolkit provides a common Specification interface implementing the specification pattern. It can be used with the *ContaoRepository* to *findBySpecification* or *countBySpecification*.

3.2 Alias generator

Toolkit provides an flexible, configurable alias generator which is able to creates unique aliases in different formats.

3.2.1 Filter based alias generator

The core of this tool is a filter based alias generator. It implements the alias generator interface and uses a set of filters to generate the alias. It uses a validator to ensure a valid alias is generated.

Features

- Different alias generation strategies
- Combining multiple columns

- Customizable separator string

Provided filters

All provided filters have different strategies to modify an alias value. They can combine strings or just return a simple value.

ExistingAliasFilter This filter just passes the existing alias value. Usually it should be the first filter in the chain. If you always want to recreate an alias just don't use this filter.

RawValueFilter This filter uses the alias values

SlugifyFilter Generates a standardized value like Contao *standardize* helper function does.

SuffixFilter This filter adds a numeric suffix which is count until an valid alias is generated.

Provided validators

UniqueDatabaseValueValidator This validator checks the database if an unique value exists. It only has a global scope which means the unique value has to be unique in the whole data set.

3.2.2 Default factory

Toolkit has a default factory which creates an alias generator which has the default behaviour how Contao generates alias:

1. Only create alias value if field is empty
2. standardize the value
3. Add suffix value

```php
<?php
// Default alias generator factory is provided as service with service id:
// "netzmacht.contao_toolkit.data.alias_generator.factory.default_factory"

/** @var \Netzmacht\Contao\Toolkit\Data\Alias\Factory\AliasGeneratorFactory
↪$factory */
$aliasGenerator = $factory->create('my_table', 'alias_field', ['value', 'fields']);
```

3.2.3 Alias Generator callback

Toolkit also provides an alias generator callback. See *Callbacks* for it.

3.3 Updater

The data container definitions (dca) of Contao provides flexible, extendible data structures. One goal of Toolkit is to provide these features outside the internal data container drivers of Contao.

3.3.1 Interface Updater

Like the most tools of Toolkit the updater also depends on an interface. It's described as the Updater.

3.3.2 Usage of the database row updater

Toolkit ships with a database row updater supporting versioning and data callbacks are supported. It's provided as a service. The updater is provided as a service with service id `netzmacht.contao_toolkit.data.database_row_updater`.

```php
<?php

// Context object is passed to the save callback. Usually an instance of␣
↪\DataContainer is passed here.
$context = ...;
$data    = ['name' => 'New Name'];

// Following this are done:
// 1. Check if user has access
// 2. Check if versioning is supported. Initialize if supported
// 3. Execute the save callbacks
// 4. Save the entry recognizing alwaysSave and doNotSaveEmpty setting
// 5. Create a new version if versioning is supported.
// 6. Return the saved data.
$savedData = $updater->update('my_table', ID, $data, $context);
```

CHAPTER 4

Data container definition

The data container definitions is a powerful way of Contao to define extensible data structures. Toolkit ships some handful tools to ease manipulation and access of definitions, create often used callbacks and format values using the definitions.

4.1 Definition

Contao uses data definition array *dca* to describe data containers like tables, files or the file system. Unfortunately there is no API do address and manipulate the information. Toolkit provides a limited but simple API to access the definitions. Accessing superglobals is not required anymore which makes your code more testable.

4.1.1 DCA Manager

The entry point to access a data container file is the dca manager. If you want to access information from a dca definition, simply get the definition from the dca manager.

```php
<?php

/** @var Netzmacht\Contao\Toolkit\Dca\Manager $dcaManager */
$dcaManger  = $container->get('netzmacht.contao_toolkit.dca.manager');

/** @var Netzmacht\Contao\Toolkit\Dca\Definition $definition */
$definition = $dcaManger->getDefinition('tl_content');
```

4.1.2 Access definition

If you want to access some information from the definition there are two methods: *get* to retrieve information and *set* to add information. To get the information path there are two syntax supported: Array notated syntax or a string separated with dots.

```php
<?php

/** @var Netzmacht\Contao\Toolkit\Dca\Definition $definition */
$definition = $dcaManger->getDefinition('tl_content');
```

(continues on next page)

(continued from previous page)

```
// Read from the definition
$driver = $definition->get('config.dataContainer');
$driver = $definition->get(['config', 'dataContainer']);

// Write to the definition.
$definition->set('fields.text.eval.tl_class', 'w50');
$definition->set(['fields', 'text', 'eval', 'tl_class'], 'w50');
```

4.2 Callbacks

Data definition containers are driven by callbacks which extends the configuration, provides options and handles actions. There are some useful tools which are provided by Toolkit.

4.2.1 Developing own callbacks

If you want to create a callback class for your data container *tl_example* Toolkit provides an abstract listener containing the Data container manager.

```
<?php

 // ExampleCallbacks.php
 class ExampleCallbacks extends␣
↪Netzmacht\Contao\Toolkit\Dca\Listener\AbstractListener
 {
     /**
      * Name of the data container.
      *
      * @var string
      */
     protected static $name = 'tl_example';

     public function onSubmit()
     {
     }
 }
```

The base *Callbacks* class provides even more helpers:

getDefinition() Gets the definition of the current data container *(tl_example)* See *Definition* section for more details.

getDefinition('tl_custom') Gets the definition of a specific data container. See *Definition* section for more details.

getFormatter() Gets the formatter of the current data container *(tl_example)*. See *Formatter* section for more details.

getFormatter('tl_custom') Gets the formatter of a specific data container. See *Formatter* section for more details.

4.2.2 Provided callbacks

Alias generator callback

The alias generator uses the *Alias generator* to create an alias callback. By default a predefined alias generator is used. You may use the configurations to the *toolkit.alias_generator* configuration. The *fields* configuration is required.

```
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['alias']['save_callback'][] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Save\GenerateAliasListener::class,
     'onSaveCallback'
 ];

 $GLOBALS['TL_DCA']['tl_example']['fields']['alias']['toolkit']['alias_generator']␣
↪= [
     'factory' => 'netzmacht.contao_toolkit.data.alias_generator.factory.default_
↪factory',
     'fields' => ['title']
 ];
```

For more details please have a look at the GenerateAliasListener.

State button callback

The state button callback is used to generate the state toggle button to toggle the active state of an entry. The *stateColumn* configuration is required.

```
<?php

 $GLOBALS['TL_DCA']['tl_example']['list']['operations']['toggle']['button_callback
↪'][] = [
    ␣
↪Netzmacht\Contao\Toolkit\Dca\Listener\Button\SaveButtonCallbackListener::class,
     'onButtonCallback'
 ];

 $GLOBALS['TL_DCA']['tl_example']['list']['operations']['toggle']['toolkit'][
↪'state_button'] = [
     'disabledIcon' => 'custom-invisible-icon.png,
     'stateColumn'  => 'published',
     'inverse'      => false
 ];
```

For more details please have a look at the StateButtonCallbackListener.

Color picker wizard

The color picker wizard provides a wizard to choose a rgb color. Every configuration is optional.

```
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['color']['wizard'][] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Wizard\ColorPickerListener::class,
     'onWizardCallback'
 ];

 $GLOBALS['TL_DCA']['tl_example']['fields']['color']['toolkit']['alias_generator']␣
↪= [
     'title'     => null,
     'template'  => null,
     'icon'      => null,
     'replaceHex' => null,
 ];
```

For more details please have a look at the ColorPickerListener wizard.

File picker wizard

The file picker wizard provides a popup wizard to choose a file.

```php
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['file']['wizard'][] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Wizard\FilePickerListener::class,
     'onWizardCallback'
 ];
```

For more details please have a look at the FilePickerListener wizard.

Page picker wizard

The page picker wizard provides a popup wizard to choose a page.

```php
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['page']['wizard'][] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Wizard\PagePickerListener::class,
     'onWizardCallback'
 ];
```

For more details please have a look at the PagePickerListener wizard.

Popup wizard

The popup wizard opens a link in a popup overlay. The *href*, *title* and *icon* configuration is required.

```php
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['article']['wizard'][] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Wizard\PopupWizardListener::class,
     'onWizardCallback'
 ];

 $GLOBALS['TL_DCA']['tl_example']['fields']['article']['toolkit']['popup_wizard']␣
↪= [
     'href'        => null,
     'title'       => null,
     'linkPattern' => null,
     'icon'        => null,
     'always'      => false,
 ];
```

For more details please have a look at the PopupWizardListener wizard.

Get templates callback

The get templates callback get all available templates.

```php
<?php

 $GLOBALS['TL_DCA']['tl_example']['fields']['template']['options_callback'] = [
     Netzmacht\Contao\Toolkit\Dca\Listener\Options\TemplateOptionsListener::class,
     'onWizardCallback'
 ];
```

(continues on next page)

(continued from previous page)

```
$GLOBALS['TL_DCA']['tl_example']['fields']['template']['toolkit']['template_
↪options'] = [
    'prefix' => '',
    'exclude' => null,
];
```

For more details please have a look at the TemplateOptionsListener wizard.

4.2.3 Invoker

If you want to trigger a callback form your code you don't have to worry about the different supported callback formats. For this case toolkit provides an invoker which is registered as a service.

```
<?php

/** @var Netzmacht\Contao\Toolkit\Dca\Callback\Invoker $invoker */
$invoker = $container->get('netzmacht.contao_toolkit.callback_invoker);

// Invoke the callback and get the return values.
$options = $invoker->invoke($GLOBALS['TL_DCA']['tl_example']['fields']['template
↪']['options_callback'], [$dc]);

// Invoke a list of callbacks and define which value should changed after␣
↪invoking a callback.
// The last argument indicates that the first argument of the arguments array (
↪$value) should be changed
$value = $invoker->invokeAll($GLOBALS['TL_DCA']['tl_example']['fields']['save_
↪callback'], [$value, $dc], 0);
```

4.3 Formatter

Contao uses the data container definition to describe stored data. Also some information about the data should be presented are stored. Unfortunately there is no way to access the formatting using an API. Even more the formatting differ between different places (e.g. show view and parent view).

Toolkit provides a formatter framework which allows you to easily access the default formatting which is used in Contao. Furthermore it also allows you to customize the behaviour by registering custom formatter. This way it's much more flexible then other helper libraries (e.g. Contao Haste).

If you don't want customize the behaviour you don't have to worry about the cration of the formatter framework. Simply get the formatter for your data container:

```
<?php

/** @var Netzmacht\Contao\Toolkit\Dca\Manager $dcaManager */
$dcaManger  = $container->get('netzmacht.contao_toolkit.dca.manager');

/** @var Netzmacht\Contao\Toolkit\Dca\Formatter\Formatter $formatter */
$formatter = $dcaManager->getFormatter('tl_content');
```

4.3.1 Format a value

To format a value use the formatter and pass field name and the value:

```
<?php

 echo $formatter->formatValue('singleSRC', $contentModel->singleSRC);

 // If you use it in the backend dca view you should pass the data container␣
↪driver as the context object.
 // Some formatter requires it.

 echo $formatter->formatValue('singleSRC', $contentModel->singleSRC, $dc);
```

4.3.2 Format labels

Not only values can be formatted. Also field labels and descriptions get formatted (translated).

```
<?php

 echo $formatter->formatFieldLabel('singleSRC');
 echo $formatter->formatFieldDescription('singleSRC');
```

4.3.3 Format options

If your field contains some options the data container also stores information how to translate/format the options.

```
<?php

 echo $formatter->formatOptions('multiSRC', $contentModel->multiSRC);
```

4.3.4 Customize formatter

Toolkit provides an event driven approach to customize the formatter. Each formatter contains a chain of formatter. The chain contains three steps:

- **Pre filters** (e.g. deserialize values)
- **Formatter**
- **Post filters** (e.g. flatten arrays)

To customize the behaviour you only have to subscribe the `Netzmacht\Contao\Toolkit\Dca\Formatter\Event\Crea`

```
// config/event_listeners.php
use Netzmacht\Contao\Toolkit\Dca\Formatter\Event\CreateFormatterEvent;

return [
    CreateFormatterEvent:NAME => [
        function (CreateFormatterEvent $event) {
            if ($event->getDefinition()->getName() != 'tl_my_example') {
                return;
            }

            // CustomFilter has to be an instance of␣
↪Netzmacht\Contao\Toolkit\Dca\Formatter\Value\ValueFormatter
            $formatter = new CustomValueFormatter();
            $event->addFormatter($formatter);
        }
    ]
];
```

CHAPTER 5

View

Toolkit provides some view improvements to the default behaviour of Contao. It provides extended templates with template view helper support and an assets manager.

5.1 Templates

Toolkit provides some improvements to the default Contao template. It provides a common interface for templates, supports template helpers and a simple to use template factory.

5.1.1 Template renderer

The toolkit provides a template renderer service which wraps the rendering of the Contao templates and twig templates.

```php
<?php

declare(strict_types=1);

$renderer = $container->get('netzmacht.contao_toolkit.template_renderer');
assert($renderer instanceof␣
↪\Netzmacht\Contao\Toolkit\View\Template\TemplateRenderer);

echo $renderer->render('toolkit:be:be_main', ['foo' => 'bar']);
echo $renderer->render('toolkit:fe:fe_pgae', ['foo' => 'bar']);
echo $renderer->render('templates/views.html.twig', ['foo' => 'bar']);
echo $renderer->render('@Bundle/templates/views.html.twig', ['foo' => 'bar']);
```

Since toolkit supports template helpers creating a template would require to pass all registered helpers to a template. Using the template renderer makes it easy. You don't have to worry about any helpers.

Hint: Right now the Symfony templating engine is wrapped. Due of the deprecation of the templating support in Symfony 5 you should not rely on it. It get's removed

5.1.2 Helpers

Some templates requires helpers to improve template development and code quality by reusing helper codes. Instead of working with singletons or static helpers, Toolkit encourages you to use view helpers. Every object could be registered as a view helper for your templates.

Default helpers

Toolkit provides with *assets* and *translator* two helpers by default which are registered to every template. The assets helper is an instance of *Netzmacht\Contao\Toolkit\View\Assets\AssetsManager*. The translator helper is an instance of *ContaoCommunityAlliance\Translator\TranslatorInterface*.

```
<?php

// Inside of your template
$this->helper('translator')->translate('foo', 'bar');

$this->helper('assets')->addJavascript('foo/bar.js');
```

Register helpers

Toolkit uses an event driven approach to register template helpers. This approach allows you to register different helpers for different templates.

```
<?php

// MyCustomTemplateHelperListener.php

class MyCustomTemplateHelperListener
{
    public function␣
↪onGetTemplateHelpers(Netzmacht\Contao\Toolkit\View\Template\Event\GetTemplateHelpersEvent␣
↪$event)
    {
        if ($event->getTemplateName() === 'foo' && $event->getContentType() ===
↪'bar') {
            $event->addHelper('foo', new FooBarHelper())
        }
    }
}
```

```
// src/Resources/config/listeners.yml in your bundle

my.custom.template-helpers-listener:
  class: MyCustomTemplateHelperListener
  tags:
    - { name: 'kernel.event_listener', event: 'netzmacht.contao_toolkit.view.get_
↪template_helpers'}
```

5.2 Asset Management

Contao uses the superglobals `$GLOBALS['TL_CSS']` and `$GLOBALS['TL_JAVASCRIPT']` to register required assets on the fly. Toolkit provides a simple wrapper API to register assets. The benefit of the wrapper is that you can change the implementation.

For instance, then creating an ajax response the assets could get easily collected and returned as json array.

```
<?php

/** @var Netzmacht\Contao\Toolkit\View\AssetsManager\AssetsManager $assetsManager␣
↪*/
$assetsManager = $container->get('netzmacht.contao_toolkit.assets_manager');

// Set media type
$assetsManager->addStylesheet('files/css/print.css', 'print');

// Set media type and static flag
$assetsManager->addStylesheet('files/css/print.css', 'print', 'static');

// Set media type, static flag and an unique name
$assetsManager->addStylesheet('files/css/print.css', 'print', 'static', 'project-
↪print-css');
```

Toolkit also automates the handling of *static* assets. For debug reasons combining all static assets could be a pain. To avoid combining all assets when debugging a pain, the assets manager uses the `AssetsManager::STATIC_PRODUCTION` flag by default. This means that only in production mode the assets get the *static* flag.

Hint: The production mode is set by the dependency container using the symfony debug environment setting.

5.2.1 Template helper

The assets manager is also registered as template helper named *assets*. Read the *Helpers* section of the templates documentation.

CHAPTER 6

Insert tags

Toolkit introduces an object oriented way to implement own insert tags and also add an API to replace insert tags.

6.1 Replacing insert tags

Why does Toolkit provides an own API entry point to replace insert tags? Looking at Contao's history the way how insert tags could be replaced has changed some times. The method was protected for a while, got public static later and finally got extracted into an own InsertTags class.

Supporting different Contao versions is a bit complicated here even then not extending every class form the *Controller* class. That's why Toolkit v1 provided a standard way to replace insert tags.

```php
<?php

 // Both method does the same

 /** @var ContaoInsertTags $replacer */
 $replacer = $container->get('contao.controller.insert_tags');
 $buffer   = $replacer->replace($buffer);

 /** @var Netzmacht\Contao\Toolkit\InsertTag\Replacer $replacer */
 $replacer = $container->get('netzmacht.contao_toolkit.insert_tag.replacer');
 $buffer   = $replacer->replace($buffer);
```

Hint: You should be careful using the InsertTags class provided by Contao. It hassles with the right order of the Contao object stack.

6.2 Register a insert tag parser

The insert tag component of Toolkit allows you to provide a custom insert tag parser which do the replacing stuff for you. The `Netzmacht\Contao\Toolkit\InsertTag\Parser` interface is used for it.

```
// 1. Define your parser
class SmileyParser implements Netzmacht\Contao\Toolkit\InsertTag\Parser
{
    // Check if tag is supported. Note that not the whole insert tag is passed␣
↪here, only the part before the first ::
    public function supports($tag)
    {
        return $tag === 'smiley';
    }

    // An insert tag value, e.g. smiley::lol?color=blue would be passed this way
    // $raw = 'smiley::lol?color=blue'
    // $tag = 'smiley'
    // $params = 'lol?color=blue' (All after the first ::)-
    public function parse($raw, $tag, $params = null, $cache = true)
    {
        // Do the parsing here.
    }
}
```

```
// 2. Register your parser

// src/Resources/config/services.yml in your bundle
my.custom.smiley-insert-tag-parser:
  class: SmileyParser
  tags:
    - { name: netzmacht.contao_toolkit.insert_tag.parser}
```

Index