Bartacus

Release

September 12, 2016

Contents

Bartacus aims to integrate parts of the Symfony framework into the TYPO3 CMS to gain some advantages from Symfony, like Twig rendering and a really good DI container. Depending on your knowledge and previous experience you will like it more than Extbase and Fluid. Bartacus uses as base the old plugin structure for the sake of simplicity.

CHAPTER 1

User Guide

1.1 Overview

1.1.1 Requirements

- PHP 5.4
- Symfony 2.7

1.1.2 Installation

The only way to install Bartacus is with Composer.

```
composer require bartacus/bartacus-bundle ^0.3
```

Now take a look at the Bartacus Standard Edition to know which extra files and configuration is needed to get it running. The most important file is `typo3conf/AdditionalConfiguration.php` where the main part of Bartacus is initialised and `fileadmin/app/AppKernel.php` where all Symfony bundles and extensions which are turned into bundles are loaded.

1.1.3 The Symfony Cache

The Symfony cache gets cleared from the TYPO3 backend on system and all cache clear commands. If your `TYPO3_CONTEXT` is `Development` or a sub-context of it, Symfony watches all your files for building the container and rebuilds the container automatically. A manual cache clear is only needed if you add new files.

In all other cases if you change anything in the Twig templates, config or service definitions you have to clear the cache from the TYPO3 backend. While doing this, Bartacus calls some cache warmers to, so you never start with a complete empty cache.

1.1.4 License

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

1.2 Changelog

1.2.1 0.3.9

- Add configuration of new content element wizard to plugins.yml style configuration.

1.2.2 0.3.8

- Fix cache warmup

1.2.3 0.3.7

- TYPO3 globals are not checked anymore, before accessing them. This prevents errors with not yet existing globals.

1.2.4 0.3.6

- Initialize backend user for TSFE on Symfony dispatch too

1.2.5 0.3.5

- Add the `BE_USER` global as TYPO3 bridge service

1.2.6 0.3.4

- Add the cache hash calculator as TYPO3 bridge service

1.2.7 0.3.3

- Add full symfony routing/kernel dispatch within TYPO3 eID context and TSFE available.
- Handle redirect responses from content element actions.
- Create a bridge session storage to start session if not already started.
- Fix path to console and eID dispatch if deployed in a symlinked environment.
- Access to frozen `TYPO3_CONF_VARS` within Symfony container.
- Improve the typo3 bridge with predefined services and better docs.

1.2.8 0.3.2

- Add aliases to user obj hooks to allow references like `service_id?:alias`.

1.2.9 0.3.1

- Use `locale_all` from TypoScript config instead of language. Leads to locales with countries.
- Find console command like in normal symfony bundles.

1.2.10 0.3.0

- Clear the Symfony cache from TYPO3 backend.
- The `Plugin` class is deprecated. Create Symfony controllers instead.
- Retrieve globals and `makeInstance` in service configurations.
- Add routing for content elements to controllers.
- Configure Symfony translator with locale from TypoScript setup.
- Add the content object as third parameter to user functions from services.
- The `@BartacusBundle/Resources/config/config.yml` file is removed. Take a look at the Bartacus Standard Edition how to fill your own `config.yml`.

1.3 Quickstart

This page provides a quick introduction to Bartacus and introductory examples. If you have not already installed Bartacus, head over to the *Installation* page.

1.3.1 Extension structure

Below you see a basic extension structure for Bartacus with one content element. Typical TYPO3 extension files are not shown.

```
typo3conf/ext/content
+-- Classes
|   +-- Controller
|   |   +-- TextController.php
|   +-- AcmeContent.php
+-- Resources
    +-- views
        +-- Text
            +-- text.html.twig
```

As you can see, the important class in your extension is the `AcmeContent.php`, which transforms your extension into a Symfony bundle. Obviously it uses similar naming convention as Symfony, so take a vendor name and your extension name and camel case it together. Don't forget to add the `AcmeContent` class to your `AppKernel`.

```php
<?php

namespace Acme\Extensions\Content;

use Bartacus\Bundle\BartacusBundle\Typo3\Typo3Extension;

/**
 * Transforms this extension to a "symfony bundle"
 */
```

```
class AcmeContent extends Typo3Extension
{

}
```

Now the content element controller:

```
<?php
// typo3conf/ext/acme/Classes/Controller/TextController.php

namespace Acme\Extensions\Contact\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TextController extends Controller
{
    public function showAction($data)
    {
        return $this->render('AcmeContent:Text:text.html.twig', [
            'data' => $data,
        ]);
    }
}
```

To get the content element controller registered in the frontend, add the following to your global `plugins.yml`:

```
# fileadmin/app/config/plugins.yml

content_text:
    path: /content/text
    defaults: { _controller: AcmeContent:Text:show }
```

For the backend, add the TCA stuff as usual. More information about content elements as controllers are found in the *Content Elements* section.

1.3.2 Accessing the container

The `Controller` class from Symfony provides some convenient methods to access the container. Alternative the container is accessible via `$this->container`.

```
$service = $this->get('service_id');
// or
$service = $this->container->get('service_id');
```

1.4 Services in TypoScript

Symfony has an excellent service container with dependency injection. But in a while you have to configure some user function in TypoScript or some Hooks, which are expecting the class name. This would prevent the use of proper DI.

Fortunately Bartacus integrates the service container into TYPO3 so you can access a service in a TypoScripts `userFunc` or hooks.

Caution: To get the user functions in TypoScript working Bartacus XCLASSes the `ContentObjectRender` in a very early phase. If you have an extension installed which wants to XCLASS the the same class, the extension wins, and this functionality stops working.

1.4.1 TypoScript `userFunc`

Define your class as service with the tag `typo3.user_func`. This will expose all public function to be accessible in TypoScript. For more information about the service container see the Symfony Service Container Documentation.

- *YAML*

```
services:
    helper.frontend:
        class: Acme\Extensions\Content\Helper\FrontendHelper
        lazy: true
        tags:
            -  { name: typo3.user_func }
```

- *XML*

```
<services>
    <service id="helper.frontend" class="Acme\Extensions\Content\Helper\FrontendHelper" lazy="tr
        <tag name="typo3.user_func"/>
    </service>
</services>
```

- *PHP*

```
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Acme\Extensions\Content\Helper\FrontendHelper');
$definition->setLazy(true);
$definition->addTag('typo3.user_func');
$container->setDefinition('helper.frontend', $definition);
```

Note: The service example above is marked as a lazy service. These is a MUST to have a correct instance injected. Otherwise your service is created too early and you have a wrong dependencies injected.

Now you can use your service in a TypoScript `userFunc` and consorts:

```
site.config.titleTagFunction = helper.frontend->getPageTitle

site.10 = TEMPLATE
site.10 {
    template = FILE
    template.file = fileadmin/mastertemplate.html
    marks {

        LOGO = USER
        LOGO.userFunc = helper.frontend->getLogo

        COPYRIGHT= USER
        COPYRIGHT.userFunc = helper.frontend->getCopyright

        FOOTERMENU < footerMenu
        MAINMENU < mainMenu
        METAMENU < metaMenu

        SUBTEMPLATE = TEMPLATE
        SUBTEMPLATE {
            template = FILE
            template.file.preUserFunc = helper.backend_layout->getLayout
```

```
            marks {
                CONTENT0 < styles.content.get
                CONTENT1 < styles.content.get
                CONTENT1.select.where = colPos=1
            }
        }
    }
}
```

Normally you would get passed the calling `ContentObjectRender` passed into a public property `cObj`. When using services for user functions you get passed the calling content object as third parameter to the method.

1.4.2 Bonus: Hooks

The way the user functions are made accessible is also available for hooks, which use `callUserFunction()`.

```
// ext_localconf.php

$GLOBALS['TYPO3_CONF_VARS']['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']['clearCachePostProc'][] =
```

If the hook uses `getUserObj()` instead, you must add the `typo.user_obj` tag to your service.

```
// ext_localconf.php

$GLOBALS['TYPO3_CONF_VARS']['SC_OPTIONS']['tslib/class.tslib_content.php']['typolinkLinkHandler']['te
```

Note: In future iterations Bartacus will abstract the way of defining hooks. Either with another service tag or through the Symfony event dispatcher.

If there are services which expects user objects, but are special in case of the using syntax like custom TCA eval functions, you can add an alias to the tag e.g. `<tag name="typo3.user_obj" alias="my_alias"/>` and the resulting string for using in user obj is `my_service:&my_alias`.

1.5 TYPO3 bridge and services

The common TYPO3 classes are available in the service container for you:

The `TYPO3\CMS\Core\Cache\CacheManager` is available as `typo3.cache.cache_manager` and the commom caches can be retrieved via `typo3.cache.cache_hash`, `typo3.cache.cache_pages`, `typo3.cache.cache_pagesection` and `typo3.cache.cache_rootline`.

The `TSFE` is available as `typo3.frontend_controller`, the `sys_page` on the TSFE as `typo3.page_repository` and the `cObj` on the TSFE as `typo3.content_object_renderer` service.

The `TYPO3_DB` is available as `typo3.db` service.

The `BE_USER` is available as `typo3.backend_user` service. This service may be `null` if no backend user is logged in.

The `TYPO3\CMS\Core\Resource\FileRepository` for the FAL is available as `typo3.file_repository`.

The `TYPO3\CMS\Frontend\Page\CacheHashCalculator` is available as

1.5.1 Globals and `makeInstace`

Although you have a common set of services available above, sometimes you need access to some of the other TYPO3 globals or retrieve other TYPO3 classes with `GeneralUtility::makeInstance()`. This will clutter your code and is really bad as it makes your services not testable.

Instead you can create services from TYPO3 globals with the factory pattern:

- *YAML*

```yaml
services:
    app.typo3.frontend_user:
        class: TYPO3\CMS\Core\Authentication\FrontendUserAuthentication
        factory: ["@typo3", getGlobal]
        arguments:
            - FE_USER
```

- *XML*

```xml
<services>
    <service id="app.typo3.frontend_user" class="TYPO3\CMS\Core\Authentication\FrontendUserAuthe
        <factory service="typo3" method="getGlobal"/>
        <argument>FE_USER</argument>
    </service>
</services>
```

- *PHP*

```php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\ExpressionLanguage\Expression;

$definition = new Definition(
    'TYPO3\\CMS\\Core\\Authentication\\FrontendUserAuthentication',
    ['FE_USER']
]);
$definition->setFactory([
    new Reference('typo3'),
    'getGlobal'
]);
$container->setDefinition('app.typo3.frontend_user', $definition);
```

The same it possible with classes from `GeneralUtility::makeInstance()`, but the must be set shared to false, so `makeInstance()` is still in control whether you get the same instance or a new one every time you inject the service.

- *YAML*

```yaml
services:
    app.typo3.template_service:
        class: TYPO3\CMS\Core\TypoScript\TemplateService
        shared: false
        factory: ["@typo3", makeInstance]
        arguments:
            - "TYPO3\\CMS\\Core\\TypoScript\\TemplateService"
```

- *XML*

```xml
<services>
    <service id="app.typo3.template_service" class="TYPO3\CMS\Core\TypoScript\TemplateService" s
        <factory service="typo3" method="makeInstance"/>
        <argument>TYPO3\CMS\Core\TypoScript\TemplateService</argument>
```

```
        </service>
    </services>
```

- *PHP*

```
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\ExpressionLanguage\Expression;

    $definition = new Definition(
        'TYPO3\\CMS\\Core\\TypoScript\\TemplateService',
        ['TYPO3\\CMS\\Core\\TypoScript\\TemplateService']
    ]);
    $definition->setShared(false);
    $definition->setFactory([
        new Reference('typo3'),
        'makeInstance'
    ]);
    $container->setDefinition('app.typo3.template_service', $definition);
```

1.5.2 Other caches as service

If you have defined your own cache in your extension, make it available to the service container to. It's the same as getting a global from TYPO3, but instead you are using the cache manager as a factory.

The configured cache in this example is `acme_geocoding`:

- *YAML*

```
    services:
        app.cache.acme_geocoding:
            class: TYPO3\CMS\Core\Cache\Frontend\FrontendInterface
            factory: ["@typo3.cache.cache_manager", getCache]
            arguments:
                - acme_geocoding
```

- *XML*

```
    <services>
        <service id="app.cache.acme_geocoding" class="TYPO3\CMS\Core\Cache\Frontend\FrontendInterfac
            <factory service="typo3.cache.cache_manager" method="getCache"/>
            <argument>acme_geocoding</argument>
        </service>
    </services>
```

- *PHP*

```
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\ExpressionLanguage\Expression;

    $definition = new Definition(
        'TYPO3\\CMS\\Core\\Cache\\Frontend\\FrontendInterface',
        ['acme_geocoding']
    ]);
    $definition->setFactory([
        new Reference('typo3.cache.cache_manager'),
        'getCache'
    ]);
    $container->setDefinition('app.cache.acme_geocoding', $definition);
```

1.6 Translations

String translations are possible with the wonderful translator service from Symfony. The locale for the translator is retrieved from your typoscript configuration, thus depending on the typical TYPO3 `L` url param.

1.6.1 Basic Configuration

Simple add the following to your `fileadmin/app/config/config.yml` if not already exist trough the standard edition:

```
parameters:
    locale: en

framework:
    default_locale:  "%locale%"
    translator:      { fallbacks: ["%locale%"] }
```

This will activate the translator service and defines the default locale as fallback locale

Caution: To get the locale retrieving from TypoScript working Bartacus XCLASSes the `TypoScriptFrontendController` in a very early phase. If you have an extension installed which wants to XCLASS the the same class, the extension wins, and this functionality stops working.

1.6.2 Translation files

One restriction applies. Translations files can only be placed into real Symfony bundles `<bundle>/Resources/translations` dir or under the global `fileadmin/app/Resources/translations` dir. At the moment it is not possible to place translations into a extension "bundle".

1.7 Content Elements

With Bartacus you are able to dispatch content elements to Symfony controller actions. This creates a harmony with the future ability to dispatch routes directly to Symfony and not to TYPO3.

1.7.1 Configuration

To dispatch content elements to Symfony, Bartacus makes a trick with a special plugin routing style. To make this work you have to activate the Symfony routing, although the `routing.yml` can be empty. Your content elements are configured in the `plugins.yml`. Add this to your main `config.yml`:

```
# fileadmin/app/config/config.yml

framework:
    router:
        resource: "%kernel.root_dir%/config/routing.yml"
        strict_requirements: ~

bartacus:
    plugins:
```

```
        resource: "%kernel.root_dir%/config/plugins.yml"
        strict_requirements: ~
```

An example contact form looks like the following:

```
# fileadmin/app/config/plugins.yml

contact_form:
    path: /contact/form
    defaults: { _controller: AcmeContact:Contact:send, _cached: false }
```

You have to take care about the naming convention of the `path` part. The first part is always the extension key and the second part the plugin name. This naming is a MUST. Otherwise it won't work. This would be the equivalent to a `tx_contact_form` plugin class of pi_base plugins.

The `_cached` parameter is optional and if not given, it defaults to true. If false, the content element is created as `USER_INT` and will not be cached.

You can also import the plugin configuration with the usage of a prefix, which simplifies the path a little:

```
# fileadmin/app/config/plugins.yml

contact:
    resource: "@AcmeContact/Resources/config/plugins.yml"
    prefix: /contact
```

```
# typo3conf/ext/contact/Resources/config/plugins.yml

contact_form:
    path: /form
    defaults: { _controller: AcmeContact:Contact:send, _cached: false }
```

Configuration of the TCA for inserting the plugin in the backend and available fields MUST be done in `Configuration/TCA` and `Configuration/TCA/Overrides` as usual.

1.7.2 Usage

The code for the content element is simple like a Symfony controller.

```
<?php
// typo3conf/ext/contact/Classes/Controller/ContactController.php

namespace Acme\Extensions\Contact\Controller;

use Acme\Extensions\Contact\Form\Model\Contact;
use Acme\Extensions\Contact\Form\Type\ContactType;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class ContactController extends Controller
{
    public function sendAction(Request $request, $data)
    {
        $form = $this->createForm(new ContactType(), new Contact());

        $form->handleRequest($request);
        if ($form->isValid()) {
            /** @var Contact $contact */
```

```
            $contact = $form->getData();

            $emailTo = $this->getParameter('contact.email');
            $message = \Swift_Message::newInstance()
                ->setSubject('New message: '.$contact->getSubject())
                ->setSender($contact->getEmail())
                ->setReplyTo($contact->getEmail())
                ->setFrom(is_array($emailTo) ? $emailTo[0] : $emailTo)
                ->setTo($emailTo)
                ->setBody(
                    $this->renderView(
                        'AcmeContact::email.txt.twig',
                        ['contact' => $contact]
                    ),
                    'text/plain'
                )
            ;

            $this->get('mailer')->send($message);

            return $this->render('AcmeContact::thanks.html.twig');
        }

        return $this->render(
            'AcmeContact::show.html.twig',
            [
                'header' => $data['header'],
                'form' => $form->createView(),
            ]
        );
    }
}
```

The data which is usually retrieved via `$this->cObj->data` in old pi_base plugin is now injected into the `$data` parameter of the method if it exists.

Note: Bartacus mocks the Symfony http foundation kernel requests, which means you have access to the `Request` instance as a sub request as seen above and must return a `Response` instance, but none of the usual kernel events are dispatched.

1.7.3 TYPO3 new content element wizard

If you want to have a content element in the new content element wizard it's as easy as adding some defaults to the plugin configuartion:

```
# typo3conf/ext/contact/Resources/config/plugins.yml

contact_form:
    path: /form
    defaults:
        _controller: AcmeContact:Contact:send
        _cached: false
        _wizard:
            title: Contact form
```

```
            description: A form for the user to contact you
            icon: contact_form.png
```

The icon is expected to live in `typo3conf/ext/contact/Resources/icons/wizard/contact_form.png` and should be 32x32 pixels big.

Organise in new tab

To put the element into your own tab/header simply add the `header` param to `_wizard`:

```
# typo3conf/ext/contact/Resources/config/plugins.yml

contact_form:
    path: /form
    defaults:
        _controller: AcmeContact:Contact:send
        _cached: false
        _wizard:
            header: Special forms
            title: Contact form
            description: A form for the user to contact you
            icon: contact_form.png
```

Restrict in rootline

Maybe element in the wizard should be only shown in given page rootline? Simply add the `rootline` param:

```
# typo3conf/ext/contact/Resources/config/plugins.yml

contact_form:
    path: /form
    defaults:
        _controller: AcmeContact:Contact:send
        _cached: false
        _wizard:
            header: Special forms
            title: Contact form
            description: A form for the user to contact you
            icon: contact_form.png
            rootline: 181
```

1.8 Ajax / Symfony Routing

Beside content elements with Symfony you can create whole applications or ajax request with the usual Symfony full stack framework and routing.

A full Symfony kernel dispatch is registered as TYPO3 eID script and a TSFE object is initialized for you. So you have access to the usual TYPO3 functionality within the Symfony framework.

1.8.1 Configuration

To not check every request against the Symfony routes you have to configure route prefixes which should be dispatched. Add the dispatch URIs to your main `config.yml`. For example:

```
# fileadmin/app/config/config.yml

bartacus:
    dispatch_uris:
        - /retailer/
        - /shared/
        - /filter/
        - /event/
```

So any `/event/123` or similar URL will be dispatched by the Symfony kernel. Any URL which matches a given dispatch URI, but the route is not found generates a normal 404 error and is not handled back to TYPO3.

1.8.2 Usage

Usage is the same as routing in a full stack Symfony application. Read the docs of the Symfony routing to get familiar with it.

One thing you have to take care of: If not passed the TSFE `sys_language_uid` is `0` and therefore the locale of the translator. You need to pass the `L` parameter explicetely to the route by either adding the `?L=1` query parameter as usual or by encoding it in the route itself:

```
# typo3conf/ext/event/Resources/config/routing.yml

event_show:
    path: /event/{L}/{id}
    defaults: { _controller: AcmeEvent:Event:show, _format: json }
    requirements:
        _format: json
```