xm_tools Documentation

Release 0.4

Steve Lenz, Sebastian Gierth, Wolfram Eberius

December 08, 2016

Contents

A framework for TYPO3 extensions.

Contents:

CHAPTER 1

Overview

xm_tools was built to facilitate common use cases a TYPO3 extension developer runs into. It is an extension for extension developers, so you will not find any frontend or backend plugin in it. The purpose was to generalize certain workflows and so outsource commonly used source code.

A common requirement for TYPO3 extensions is to list and filter data. Using session data more quickly for developers is described here. Sometimes we want to access another extension from our extension, such as the other extension's configuration, assets or translations. You can learn how to do this in the section Extension Manager.

You can use xm_tools to store global translations that you want to use from other extensions as well. Some usefull Fluid view helpers are also included. Hooks for existing extensions we sometimes use are covered here.

Another part of xm_tools is to help connect an extensions to an API. There is a workflow described here which covers the generation of an extension based on the data structure and how to query it. xm_tools makes it possible to use your entities and repositories the 'extbase way' as if the data was in your TYPO3 database.

CHAPTER 2

Configuration

Adding the static template *Xima Tools* makes four configuration settings available:

- **plugin.tx_xmtools.settings.loggingIsEnabled**: enable some logging through the *TYPO3CMSCoreLogLogger*
- **plugin.tx_xmtools.settings.devModeIsEnabled**: currently not in use
- **plugin.tx_xmtools.settings.jsSupportIsEnabled**: use the integrated Javascript functions, e.g. serve parameters in Javascript.
- **plugin.tx_xmtools.settings.jsL10nIsEnabled**: use global and translations from other extensions in Javascript as well.

CHAPTER 3

Tools

Contents:

3.1 Session handling

xm_tools provides a convient interface to the wer server's session. To write data to and retrieve data from the session, use the `SessionManager`:

```
class MyController
{
    /**
     * session
     *
     * @var XmTools\Classes\Typo3\SessionManager
     * @inject
     */
    protected $session;

    function updateSessionData()
    {
        $data = $this->session->get('myData');
        $date['newStuff'] = 'new text';

        $this->session->set('myData', $data);
    }
}
```

This session is kept for the current extension and the current page. If you want to store data somewhere specific to be able to use it on other pages and/or by another extensions, you can use (see `SessionManager::set`) with key of your choice:

```
function updateSessionDataForSomewhereElse()
{
    $data = $this->session->get('myData');
    $date['newStuff'] = 'new text';

    //send it to another extension on another page
    $this->session->set('myData', $data, 100, 'anotherExtensionKey');
}
```

3.2 Query

xm_tools provides a basic query class which can be used to represent a user's query to the database. The `QueryTrait` simply provides common filters and their *getter* and *setter* functions:

- *limit*
- *currentPage*
- *searchTerm*
- *lang*
- *sort*
- *context* (currently used in connection with the API: different contexts decide about which properties (or just all) of an entity are to be sent back)

In order to set up a query object specific for your domain, create your custom query object and use the `QueryTrait` in it:

```
class BlogQuery
{
    use \Xima\XmTools\Classes\Typo3\Query\QueryTrait
    {
        getParamKeys as traitGetParamKeys;
    }

    /**
     * subject
     *
     * @var string
     */
    protected $subject;

    public function getSubject() {

        return $this->subject;
    }

    public function setSubject($subject) {

        $this->subject = $subject;
        return $this;
    }
}
```

3.3 Paging

The `Paginator` generates links to walk through a paged result set. It returns an array of links (see `Paginator::getPageBrowser`).

```
$paginator = new \Xima\XmTools\Classes\Helper\Paginator();
$pageBrowser = $paginator->getPageBrowser(
        $countAllResult,
        $countItemsPerPage,
        $currentPage,
        $url
```

```
    );

$this->view->assign('pageBrowser', $pageBrowser);
```

A template to render the returned array is not yet part of xm_tools. An example template would look like:

```
{namespace xmTools = Xima\XmTools\Classes\ViewHelpers}

<f:if condition="{pageBrowser.countPages}">
    <nav class="pager-nav nav-js">
        <ul class="pagination">
            <f:if condition="{pageBrowser.prev}">
                <li>
                    <f:link.action arguments="{page: pageBrowser.prev, tab: tab}" title="{dict.list_p
                        <span class="icon icon-backward"></span>
                    </f:link.action>
                </li>
            </f:if>

            <f:if condition="{xmTools:object.ArrayCheck(array:pageBrowser.pages,needle:1,check:'NOT_I
                <li>
                    <f:link.action arguments="{page: 1, tab: tab}">
                        1
                    </f:link.action>
                </li>
                <f:if condition="{xmTools:object.ArrayCheck(array:pageBrowser.pages,needle:2,check:'N
                    <li>
                        ...
                    </li>
                </f:if>
            </f:if>

            <f:for each="{pageBrowser.pages}" as="page" key="n">
                <f:if condition="{page.current} == 0">
                    <f:then>
                        <li>
                            <f:link.action arguments="{page: n, tab: tab}">
                                {n}
                            </f:link.action>
                        </li>
                    </f:then>
                    <f:else>
                        <li class="active">
                            <a>{n}</a>
                        </li>
                    </f:else>
                </f:if>
            </f:for>

            <f:if condition="{xmTools:object.ArrayCheck(array:pageBrowser.pages,needle:pageBrowser.co
                <f:if condition="{xmTools:object.ArrayCheck(array:pageBrowser.pages,needle:pageBrowse
                    <li>
                        ...
                    </li>
                </f:if>
                <li>
                    <f:link.action arguments="{page: pageBrowser.countPages, tab: tab}">
                        {pageBrowser.countPages}
```

```
                    </f:link.action>
                </li>

            </f:if>

            <f:if condition="{pageBrowser.next}">
                <li>
                    <f:link.action arguments="{page: pageBrowser.next, tab: tab}" title="{dict.list_p
                        <span class="icon icon-forward"></span>
                    </f:link.action>
                </li>
            </f:if>
        </ul>
    </nav>
</f:if>
```

3.4 Filter-List-Flow

A common use case of TYPO3 extensions is to list, filter and show data. The suggested pattern to do this repeated task described and support by xm_tools consists of two plugins: one for the query and one for the list. They share the query submitted by the user by using the session.

3.4.1 The query plugin

Create a query class that use the `QueryTrait` and add the properties you want to filter the data for (see here). Create a controller and a template for the filter you want to show to your user.

```
abstract class AbstractController
{
    /**
     * paginator
     *
     * @var \Xima\XmTools\Classes\Helper\Paginator
     * @inject
     */
    protected $paginator = null;

    /**
     * @param $className
     * @throws \TYPO3\CMS\Extbase\Mvc\Exception\NoSuchArgumentException
     */
    protected function initializeQuery($className)
    {
        //the filter object
        $query = $this->session->get('query');
        if (!is_a($query, $className)) {
            $query = $this->objectManager->get($className);
        }

        //paging
        if ($this->request->hasArgument('page')) {
            $query->setCurrentPage($this->request->getArgument('page'));
        }
```

```
        //limit
        $query->setLimit($this->settings ['flexform'] ['limit']);

        $this->query = $query;
    }
}

class BlogQueryController extends AbstractController
{

    /**
     * action new
     *
     * @return void
     */
    public function newAction()
    {
        $this->initializeQuery('Xima\BlogExampleExtension\Domain\Model\Query\BlogQuery');

        //example form data: get tags to allow for filtering on them
        $tags = $this->objectManager->get('Xima\BlogExampleExtension\Domain\Repository\TagRepository
        $this->view->assign('tags', $tags);

        $this->session->set('query', $this->query);
        $this->view->assign('query', $this->query);

    }

    /**
     * action create
     *
     * @param \Xima\BlogExampleExtension\Domain\Model\Query\BlogQuery $newBlogQuery
     * @return void
     */
    public function createAction(\Xima\XmDwiDb\Domain\Model\Query\BlogQuery $newBlogQuery)
    {
        $this->session->set('query', $newBlogQuery);
        $this->redirect('new');
    }

}
```

3.4.2 The list plugin

In your list action, retrieve the query from the session and let your entity repository filter your data:

```
class BlogController extends AbstractController
{

    ...

    /**
     * action list
     *
     * @return void
     */
    public function listAction()
```

```
    {
        $this->initializeQuery('Xima\BlogExampleExtension\Domain\Model\Query\BlogQuery');

        $items = $this->repository->findAllByQuery($query);
        $this->view->assign('items', $items);
    }
}
```

Note: The repository function *findAllByQuery* is so far only implemented for the `ApiRepository` class (see `ApiRepository::findAllByQuery`).

3.5 I10n

Translating extensions is often needed for frontend plugins. One challenge is to offer those translations for Javascript functions, another is to share translations between extensions, so that common labels are stored in one place. The `Localization` class tries to accomplish all those goals, while additionally offering a quick access to translations from fluid templates. Translations are to be stored in the common xliff files. If you want to share a global dictionary with all your extensions, rename any *Resources\Private\Language\locallang.xlf.dist* file to *.xlf and/or add more languages.

3.5.1 Retrieve a dictionary and assign it to the view

By default, the returns all translations from your current extension. You can pass an array of additional extension names you want to have included in your dictionary:

```
In your controller action:
...
$this->view->assign('dict', Localization::getDictionary());
...

In your fluid template:
...
{dict.subject}: {post.subject}
...
```

3.5.2 Use the dictionary in Javascript

If you want to have your translations available in Javascript as well, enable it by configuring xm_tools to do so (see Configuration). This will generate a Javascript file and make it load. Your translations are available in Javascript through:

```
...
xmTools.getTranslation('subject');
...
```

3.5.3 Use the dictionary in PHP

Of course you can also access your translations of the dictionary by using the `Dictionary::getTranslations` method or by accessing the translations directly via the magic `Dictionary::__call` method, e.g.:

```
...
$dictionary = Localization::getDictionary();
echo $dictionary->subject();
...
```

3.6 Parameters

The meaning of parameters in this extension is the availability of some configuration or other information to all extensions. Imagine you want all your dates formatted in the same way, being it called from PHP, Fluid or Javascript. Just copy *parameters.yml.dist* from xm_tools' root folder to *parameters.yml* and fill your data you want to have available anywhere. Add any configuration in YAML style.

The Servcies class looks for an existing *parameters.yml*, parses (and caches it) and makes it available. If you want to have the parameters available in Javascript, enable it by configuring xm_tools to do so (see Configuration). This will generate a Javascript file and make it load. Your translations are available in Javascript through:

```
...
xmTools.getParameter('dateFormat');
...
```

3.7 Caching

The `CacheManager` offers a simple way to write and restore data to the file system. Currently, a cache counts as valid when it's not older than 24 hours. See `CacheManager::write` and `CacheManager::get`.

3.8 Extension manager

The xm_tools extension offers to access other extensions, meaning their settings, configuration, assets or translations. Get an extension by injecting the `ExtensionManager` and calling it's method `ExtensionManager::getExtensionByName`:

```
class BlogController
{
    /**
     * @var \Xima\XmTools\Classes\Typo3\Extension\ExtensionManager
     * @inject
     */
    protected $extensionManager;

    public function exampleAction()
    {
        $someOtherExtension = $this->extensionManager->getExtensionByName('SomeOtherExtensionName');
    }
}
```

This will give you a `Extension` object.

3.9 Services

The `Services` class can be seen as a facade that you can inject to your controller (or by extending the `AbstractController`). Once initialized, there is a collection of usefull functions you can use:

- `Services::getLang`: get the current language
- `Services::getExtension`: get the current extension (the one you are currently developing)
- `Services::getParameters`: get the system wide parameters (see here)
- `Services::addFlexforms`: convenient function to add flexforms to a plugin
- ...

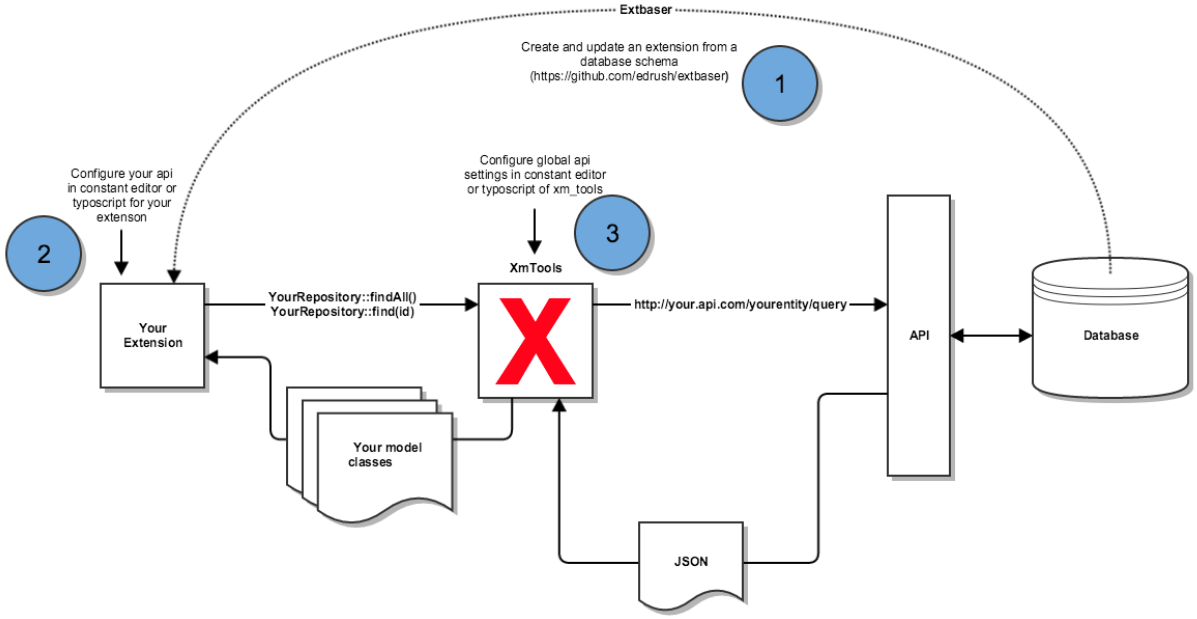
For a full list see the API documentation.

3.10 REST-API Connector

This section describes how to create a TYPO3 extension that retrieves data from and sends data to an API, while acting as an Extbase extension on the TYPO3 side.

Given there is an API that publishes data. We want our extension to have entity classes mapping to the structure the API returns. By using the tool Extbaser, you can set up your new extension's skeleton, the file *ExtensionBuilder.json*. Using this and the TYPO3 backend extension Extension Builder you can create your extension.

The following image describes the way to create your API extension and how data flows.



Enabling the new entities' repositories to retrieve data from the API requires the following steps:

- configure your API connection in the Typoscript template of your plugin or in the extension settings (*ext_conf_template*, then leave the *settings* out):

```
...
settings {
       api {
        # cat=plugin.tx_yourextension/api/001; type=string; label=Api-URL
        url =
        # cat=plugin.tx_yourextension/api/002; type=string; label=Api-Key
        key =
        # cat=plugin.tx_yourextension/api/003; type=string; label=Api-Schema (available placeholders
        schema = #e.g.  "[Api-URL]/[Api-Route]?api_key=[Api-Key]"
        # cat=plugin.tx_yourextension/api/004; type=string; label=Route for finding one result by id
        routeFindById =
```

```
        # cat=plugin.tx_yourextension/api/005; type=string; label=Route for finding by query (availa
        routeFindByQuery =
        # cat=plugin.tx_yourextension/api/006; type=string; label=Route for creating entities (avail
        routeCreate =
        # cat=plugin.tx_yourextension/api/007; type=string; label=Route for updating entities (avail
        routeUpdate =
        # cat=plugin.tx_yourextension/api/008; type=boolean; label=Use Api-Cache
        isCacheEnabled =
    }
...
```

- make your model classes extend *\Xima\XmTools\Classes\API\REST\Model\AbstractEntity*
- make your repositories extend *\Xima\XmTools\Classes\API\REST\Repository\ApiRepository* and implement *\Xima\XmTools\Classes\Typo3\Extension\ExtensionAwareInterface*

You can then use your new extension's repositories just the same way as native Extbase repositories, e.g.:

```
...
$repository = $objectManager->get('\Xima\BlogExampleExtension\Domain\Repository\BlogRepository');
$blogs = $repository->findAll();
...
```

This wil retrieve all country entites from the API (considering the API offers a corresponding route, meaning the demanded entity's name e.g. *http://your.api/country/query?&api_key=your_key*). The retrieved data gets mapped to your extension's entity class, e.g. *XimaBlogExampleExtensionDomainModelCountry*.

3.11 ErrorHandler and ProductionExceptionHandler

TYPO3 allows to change handlers for Errors and Exceptions. Configure the following handlers and set up your email address to receive mails about these events:

1. Go to *Install Tool*, switch to *All configurations*
2. Set *[SYS][errorHandler]=\Xima\XmTools\Typo3\Handler\ErrorHandler*
3. Set *[SYS][productionExceptionHandler]=\Xima\XmTools\Typo3\Handler\ProductionExceptionHandler*
4. Set the recipient e-mail address in *TypoScript* (xmTools.errorHandler.recipient =) or in *Constant Editor* (Multiple mail addresses possible as CSV)

3.12 Extensions

Some relief for some extensions...

Contents:

3.12.1 ke_search

To ease the use of Custom indexers for the search extension ke_search, xm_tools offers two components:

- Tha value object `IndexEntry` class for new index entries.
- The `AbstractIndexer`, which your custom indexer may extend to care for registering the indexer and storing *IndexEntry* objects.

Usage example:

```
class BlogIndexer extends AbstractDWIDBIndexer
{
    const TYPE = 'blog';

    public function customIndexer(&$indexerConfig, &$indexerObject)
    {
        $content = '';

        if ($indexerConfig['type'] == self::TYPE) {

            $repository = $objectManager->get('\Xima\BlogExampleExtension\Domain\Repository\BlogRepos
            $blogs = $repository->findAll();

            foreach ($blogs as $blog)
            {
                $indexEntry = new IndexEntry();
                $indexEntry->setTitle($blog->getTitle());
                $indexEntry->setAdditionalFields(...);
                // add more entry data

                parent::storeInIndex($indexEntry, $indexerObject, $indexerConfig);
            }

            $content = parent::getReport($indexerConfig, $blogs);
        }

        return $content;
    }

    protected function getType()
    {
        return self::TYPE;
    }

    protected function getName()
    {
        return 'BlogIndexer';
    }
}
```

3.13 More tools

- Helper functions in API documentation: Helper
- Logger: The `Logger` class uses the *\TYPO3CMS\Core\Log\LogManager* to log data. It must be enabled, see Configuration.

```
        $logger = $objectManager->get('Xima\XmTools\Classes\Typo3\Logger');
        /* @var $logger \Xima\XmTools\Classes\Typo3\Logger */
        $logger->log('Usefull log information...');
```

- FalHelper: ...
- FEUser: ...
- FlexFormHelper: ...

ViewHelper

- **ResponsiveImageViewHelper** The `ResponsiveImageViewHelper` can be used to process and render images with <*picture*> and subsequent <*source*> tags for images fitting the device loading the image.
- **URLSafeViewHelper** The `URLSafeViewHelper` renders links, be it internal or external (workaround for https://forge.typo3.org/issues/72818).

4.1 Control

- **ForViewHelper** The `ForViewHelper` represents the for loop (e.g. For loops in PHP). You can name the loop variable if you want to access it in your template.
- **IfViewHelper** The `IfViewHelper` can be used to render a template part depending on multiple conditions that can be AND or OR joined.

4.2 Form

- AdvancedSelectViewHelper: ...

4.3 Object

- **ArrayCheckViewHelper** The `ArrayCheckViewHelper` offers checks on arrays, such as *in_array()*, *empty()*. Possible conditions are 'IN','NOT_IN', 'NOT_FIRST', 'NOT_LAST', 'EMPTY', 'NOT_EMPTY', 'IS_ARRAY', 'IN_KEYS', 'NOT_IN_KEYS'. Usage example:

```
{namespace xmTools = Xima\XmTools\Classes\ViewHelpers}
<f:if condition="{xmTools:object.ArrayCheck(array:yourArray,needle:1,check:'IN')}">
    ...
</f:if>
```

- **ArrayExplodeViewHelper** The `ArrayExplodeViewHelper` encapsulates PHP's explode() function. Usage example:

```
{namespace xmTools = Xima\XmTools\Classes\ViewHelpers}
<f:for each="{xmTools:object.ArrayExplode(delimiter:',',string:someString)}" as="item">
    ...do something with {item}
</f:for>
```

- **ArrayImplodeViewHelper** The `ArrayImplodeViewHelper` encapsulates PHP's implode() function and displays the output. You can specify a key if the array is an array or a property or function if the array is an array of objects. Usage example:

```
{namespace xmTools = Xima\XmTools\Classes\ViewHelpers}
<xmTools:object.ArrayImplode glue=", " array="{someArray}">
```

- **StrReplaceViewHelper** The `StrReplaceViewHelper` encapsulates PHP's str-replace() function.
- **StrtolowerViewHelper** The `StrtolowerViewHelper` encapsulates PHP's strtolower() function.
- **StrtoupperViewHelper** The `StrtoupperViewHelper` encapsulates PHP's strtoupper() function.

CHAPTER 5

Templates

Pake-Link (workaround for internal and external Links in TYPO3 7 < 7.6.3)

see https://forge.typo3.org/issues/72818

The Partial "Resources/Private/Partials/Link.html" can be used as a workaround to correctly render internal and external links:

```
<v:render.template file="EXT:xm_tools/Resources/Private/Partials/Link.html" variables="{parameter: da
```

API documentation