Best practices with extbase and fluid

Oliver Klee | `http://www.oliverklee.de/` | `@oliklee`

Version as of February 16, 2017

Contents

1 SQL and TCA

1.1 Formatting

For consistency with PSR-2, use 4 spaces for indentation.

1.2 Remove unneeded default-generated columns

To improve performance, remove the following columns if you do not need them (or do not add them in the first place):

hidden, starttime, endtime if you do not need this restrictions

sys_language_uid, l10n_parent, l10n_diffsource, KEY language if the records in this table will never be translated

t3ver_* if you do not need versioning or workspaces for this table

1.3 Table naming

1.3.1 M:n association tables

Name association table names semantically instead of only with the names of the related tables.

Bad:

```
tx_programs_domain_model_frequentlyaskedquestioncategory
```

Good:

```
tx_programs_domain_model_categorization
```

1.4 Bulk inserts/reads

If you need to read or write thousands or records (e. g., for the initial import), consider using raw SQL instead of using the models. If you use the models, this will consume huge amounts of memory as each and every model will be instantiated and kept in memory.

1.5 Security headers in the TCA files

Always add the security line on top of all TCA files, `ext_tables.php` and `ext_localconf.php`:

```
defined('TYPO3_MODE') or die('Access denied.');
```

2 Domain models

2.1 Naming

2.1.1 Singular/Plural

Use the singular with CamelCase, not plural.

Bad:

```
class OliverKlee\Books\Domain\Model\Users {...}
```

Good:

```
class OliverKlee\Books\Domain\Model\User {...}
```

2.1.2 DDD contexts as namespaces

If you have a lot of classes, practice good domain-driven design (DDD), group you model by contexts, and use the contexts as sub-namespaces.

Good:

- `Domain\Model\Identity\Organization`
- `Domain\Model\Identity\User`
- `Domain\Model\Place\PostalAddress`
- `Domain\Model\Workshops\Tag`
- `Domain\Model\Workshops\Workshop`

2.1.3 Method names

Only getters should be named get*. Other methods that are not regular getters should be named differently, e. g., calculate*, retrieve*, determine* etc.

Bad:

```
public function getMostRecentItem($index) {...}
```

Good:

```
public function retrieveMostRecentItem($index) {...}
```

2.2 Queries

Don't use repositories, SQL or queries in your models. Instead, use relations (usually, with lazy loading). Or move the corresponding method into a repository

Bad:

```php
class Brochure extends AbstractEntity
{
    public function getSubCategories()
    {
        $programCategoryRepository = $this->objectManager->get(
          ProgramCategoryRepository::class
        );

        return $programCategoryRepository->findAllSubCategoriesByMainCategory(
          $this->getFirstCategory()
        );
    }
}
```

Instead of creating getters that iterate over data and do repository calls for these, use find methods in repositories.

2.3 Associations

2.3.1 Lazy associations

Always use `@lazy` for your associations. The only exception is if you *always* use the association if you use the model.

If you use lazy n:1 or 1:1 associations and you want to pass the return value of the getter into a type-hinted method, you'll need to resolve the lazy loading in the getter:

```php
/**
  * @var \OliverKlee\Books\Domain\Model\Group
  * @lazy
  */
protected $group = null;

public function getGroup()
{
    if ($this->group instanceof LazyLoadingProxy) {
        $this->group = $this->group->_loadRealInstance();
    }

    return $this->group;
}
```

2.3.2 Count methods on collections

Calling `count` on a collection results in a query each time it is called. This is even the case if the association has been iterated over before.

If your application is performance-critical, consider fetching the count directly from the relation counter cache. You'll need to update the cache each time the relation is set or an item is added or removed, though.

Example:

```php
trait CachedRelationCount
{
    /** @var int[] */
    protected $cachedRelationCountsCount = [];

    /**
     * @param string £propertyName relation name (plural, lower camelCase)
     * @return int
     */
    protected function getCachedRelationCount($propertyName)
    {
        if (array_key_exists($propertyName, $this->cachedRelationCountsCount)) {
            return $this->cachedRelationCountsCount[$propertyName];
        }

        $this->cachedRelationCountsCount[$propertyName]
          = $this->getUncachedRelationCount($propertyName);

        return $this->cachedRelationCountsCount[$propertyName];
    }

    /**
     * @param string £propertyName relation name (plural, lower camelCase)
     * @return void
     */
    protected function flushRelationCountCache($propertyName)
    {
        unset($this->cachedRelationCountsCount[$propertyName]);
    }

    /**
     * Retrieves the relation count for the given property.
     *
     * This methods tries to avoid database accesses by using the relation
     * counter cache if the relation is still a LazyObjectStorage. Otherwise,
     * the normal COUNT query will be performed as a fallback.
     *
     * This method does not cache its results.
     *
     * @param string £propertyName relation name (plural, lower camelCase)
     * @return int
     */
    protected function getUncachedRelationCount($propertyName)
    {
        if ($this->$propertyName instanceof LazyObjectStorage) {
            $reflectionClass = new \ReflectionClass(LazyObjectStorage::class);
            $reflectionProperty = $reflectionClass->getProperty('fieldValue');
            $reflectionProperty->setAccessible(true);
            $count = (int)$reflectionProperty->getValue($this->$propertyName);
        } else {
            $count = $this->$propertyName->count();
        }

        return $count;
    }
}
```

The application in a model then looks like this:

```
public function setThreads(ObjectStorage $threads)
{
    $this->flushThreadsCountCache();
    $this->threads = $threads;
}

public function getNumberOfThreads()
{
  return $this->getCachedRelationCount('threads');
}

public function addThread(Thread $thread)
{
    $this->flushThreadsCountCache();
    $this->threads->attach($thread);
}

public function removeThread(Thread $threadToRemove)
{
    $this->flushThreadsCountCache();
    $this->threads->detach($threadToRemove);
}
```

In the fluid template, you then use `item.numberOfThreads` instead of `item.threads.count`:

```
<p>
  Number of threads: {item.numberOfThreads}
</p>
```

2.3.3 Association type hinting

Always use the fully-qualified class name (FQDN) for the type annotation for attributes. (Extbase is not namespace-aware in this regard.)

You'll need to stop PhpStorm from over-eagerly using the short class name for you.

Bad:

```
use \OliverKlee\Books\Domain\Model\Group;

class Forum {
  /**
    * @var Group
    * @lazy
    */
  protected $group = null;
}
```

Good:

```
class Forum {
  /**
    * @var \OliverKlee\Books\Domain\Model\Group
    * @lazy
    */
  protected $group = null;
}
```

2.4 Traits

If you have the same attribute(s) and the corresponding getters/setters in multiple models, consider either creating a common subclass or an interface plus Traits. Which version (if any) is right depends on what these attributes mean semantically.

Examples for traits:

- `Authored` for models that have an association to an author
- `CreationDatable` for models that have a getter for the creation date
- `CachedRelationCount` for providing the code for accessing a cached relation count (as mentioned on page 3)

3 Repositories

3.1 Naming

3.1.1 Singular/Plural

Use the singular with CamelCase, not plural.

Bad:

```
class OliverKlee\Books\Domain\Repository\UsersRepository {...}
```

Good:

```
class OliverKlee\Books\Domain\Repository\UserRepository {...}
```

3.2 Performance

3.2.1 Storage page

If you don't need the storage page, setRespectStoragePage(false) for better performance:

```
public function initializeObject()
{
  /** @var QuerySettingsInterface £querySettings */
  $querySettings = $this->objectManager->get(QuerySettingsInterface::class);
  $querySettings->setRespectStoragePage(false);
  $this->setDefaultQuerySettings($querySettings);
}
```

3.2.2 Sorting

If you don't need sorting, don't set any sorting/ordering (for better performance). There is no code to add for this.

3.2.3 Sub-queries

If possible, move the `find*` method to another repository altogether. In this example, the method now is in the `GroupMembershipRepository` instead of the `GroupRepository`:

```
class GroupMembershipRepository extends Repository
{
  public function findByGroup(Group $group)
  {
    $query = $this->createQuery();
    $query->matching($query->equals('group', $group))

    return $query->execute();
  }
}
```

You can also use queries that create JOINs instead of querying other repositories in a loop:

```
class GroupMembershipRepository extends Repository
{
  public function findByGroupName(string $groupName)
  {
    $query = $this->createQuery();
    $query->matching($query->equals('group.name', $name))

    return $query->execute();
  }
}
```

3.3 Safe LIKE queries

Make sure to always correctly escape in LIKE queries to avoid SQL injections.

Bad:

```
public function findBySearchTerm($term)
{
  $query = $this->createQuery();
  $query->like($searchField, '%' . $searchTerm . '%', false);

  return $query->execute();
}
```

Good:

```
public function findBySearchTerm($term)
{
  $safeSearchTerm = $this->getDatabaseAdapter()
    ->escapeStrForLike($searchTerm, $this->tableName);

  $query = $this->createQuery();
  $query->like($searchField, '%' . $safeSearchTerm . '%', false);

  return $query->execute();
}
```

4 Controllers

4.1 Naming

4.1.1 Singular/Plural

Use the singular with CamelCase, not plural.

Bad:

```
class OliverKlee\Books\Controller\UsersController {...}
```

Good:

```
class OliverKlee\Books\Controller\UserController {...}
```

4.2 What to put where

4.2.1 Slim controllers

Controllers should be slim and mostly just pass data between repositories and views (and do the occasional forward, redirect or flash message). All other business logic should be in the models or in services instead. Data querying logic should be in repositories instead.

4.2.2 Iterating over data

Instead of creating methods that iterate over data and do repository calls for these, use find methods in repositories.

Bad:

```
public function getSpecialAndNonSpecialArticlesForCountries(
  array $countryIds, int $age = 0, int $minAge = 0, int $maxAge = 0,
  bool $showMainArticle = true, bool $showSubArticles = true
)
{
    $specialProgramArticle = new ObjectStorage();
    $programArticle = new ObjectStorage();

    foreach ($countryIds as $countryId) {
        $country = $this->countryRepository->findByUid($countryId);
        if ($country) {
            if (!$this->settings['showOnlySpecials']) {
                $programArticle->addAll(
                  $country->getPublishedNonSpecialArticles(
                    $age, $minAge, $maxAge, $showMainArticle, $showSubArticles
                  )
                );
            }
            $specialProgramArticle->addAll(
              $country->getPublishedSpecialArticles(
                $age, $minAge, $maxAge, $showMainArticle, $showSubArticles
              )
            );
        }
    }

    return [
      'special' => $specialProgramArticle,
      'nonSpecial' => $programArticle,
    ];
}
```

Good:

```
public function sidebarAction(Group $group = null)
{
  $profile = $this->profileRepository->findOneOrCreateForLoggedInUser();

  $this->view->assign('profile', $profile);
  $this->view->assign('selectedGroup', $group);
}
```

4.3 Parameters

Instead of getting data from the request instance, use controller parameters if possible.

Bad:

```php
protected function getBrochuresFromRequest()
{
  $brochureUids = array();
  if ($this->request->hasArgument('brochureUids')) {
    /** @var £brochureUidsFromRequest array */
    $brochureUidsFromRequest = $this->request->getArgument('brochureUids');
    if (is_array($brochureUidsFromRequest)) {
        array_map('intval', $brochureUidsFromRequest);
        $brochureUids = $brochureUidsFromRequest;
    }
  }

  return $this->brochureRepository->findAllByUids($brochureUids);
}
```

Good:

```php
public function moreMembershipsAction(Group $group, int $offset)
{
  $memberships = $this->groupMembershipRepository->findMoreByGroup(
    $group, $offset
  );

  $this->view->assign('group', $group);
  $this->view->assign('memberships', $memberships);
  $this->view->assign(
    'nextOffset', $offset + GroupMembershipRepository::LIMIT_FOR_LATEST_BATCH
  );
}
```

4.4 Access checks

Move access checks to initializeAction or to initialize*Action (if they are specific to an action).

Bad:

```php
public function showBookingAction(Booking $request)
{
  $userId = $GLOBALS['TSFE']->fe_user->user['uid'];
  $hasAccess = false;
  if ($userId > 0) {
    if ($request->getCmsUser()->getUid() == $userId) {
      $this->view->assign('request', $request);
      $this->assignDiscountsAndWithdrawalInformation($request);
      $hasAccess = true;
    }
  }

  if (!$hasAccess) {
    $this->uriBuilder->setRequest($this->request);
    $this->uriBuilder->setTargetPageUid($this->settings['myStepinPid']);
    $uri = $this->uriBuilder->uriFor();
    $this->redirectToURI($uri);
  }
}
```

Good:

```php
public function initializeAction()
{
    if ($this->getFrontEndController() === null) {
      throw new \BadMethodCallException(
        'This method may only be called in the front end.', 1465465879945
      );
    }

    if (!$this->isLoggedIn()) {
      $uriBuilder = $this->buildControllerContext()->getUriBuilder();
      $uriBuilder->reset();
      $uriBuilder->setTargetPageUid((int)$this->settings['pages']['login']);
      $this->redirectToUri($uriBuilder->build(), 0, 403);
    }
}
```

4.5 Dependency injection

Always use the fully-qualified class name (FQDN) for the type annotation of @inject. (Extbase is not namespace-aware in this regard.) Inject methods work fine with short class names, though.

Generally, prefer inject methods over inject annotations as they are faster.

Bad:

```php
/**
  * @var PageRepository
  * @inject
  */
protected $pageRepository = null;
```

Better:

```php
/**
  * @var \OliverKlee\Books\Domain\Repository\PageRepository
  * @inject
  */
protected $pageRepository = null;
```

Best (for good performance):

```php
/**
  * @var PageRepository
  */
protected $pageRepository = null;

/**
  * @param PagesRepository $repository
  *
  * @return void
  */
public function injectPagesRepository(PagesRepository $repository)
{
  $this->pagesRepository = $repository;
}
```

5 Fluid templates

5.1 What to put where

5.1.1 Business logic

Move logic from templates into the models if it's business logic.

5.1.2 JavaScript

Don't use script tags or event handler attributes in your HTML. See the JavaScript section on page 15 for solutions.

5.1.3 Data for JavaScript

Put data for JavaScript in data attributes, not in variables in script tags. This avoids XSS in that case.[1]

5.1.4 Partials

Use partials to avoid deep nesting.

Instead of using lots of arguments for a partial, use specialized partials to improve readability.

5.2 Static images

For static images that do not need to be resized, use plain IMG tags instead of an image view helper. This makes the code shorter and the rendering faster.

5.3 Syntax and formatting

For consistency with PSR-2, use 4 spaces for indentation.

5.4 DOCTYPE and HTML element

Add a DOCTYPE and HTML element to your templates and layouts to help PhpStorm. (Also, the HTML element is the place to put your `xml:ns` attribute.)

[1]`https://speakerdeck.com/helhum/fluid-xss`

5.4.1 Well-formed XML/HTML

Make sure your Fluid templates are well-formed HTML/XML:

- Always close tags (unless they are self-closing), and don't put orphaned opening or closing tags in conditions. This might result in slightly more code (which is okay).
- Use the inline form of view helpers if you need to use a view helper within an opening tag (e.g. to add HTML attributes or attribute values).

5.4.2 Indentation

For consistency with PSR-2, use 4 spaces for indentation.

5.5 Tools

Use and configures PhpStorm's auto-formatting.

Use a converter[2] for converting view helpers in your templates to inline notation.

[2] `http://www.fluid-converter.com/`

6 JavaScript

6.1 Including the JavaScript files

Include your extension JavaScript in the page bottom, not in the HEAD. This keeps the JavaScript from blocking the HTML parsing.

Bad:

```
page {
  includeJS {
    books = EXT:books/Resources/Public/JavaScript/FrontEnd/FrontEnd.js
  }
}
```

Good:

```
page {
  includeJSFooter {
    books = EXT:books/Resources/Public/JavaScript/FrontEnd/FrontEnd.js
  }
}
```

Note: This approach makes it necessary that there is no inline JavaScript in your page that requires the JavaScript file.

6.2 Event handlers

Use your JavaScript file to attach any necessary event handlers to your DOM. Don't have any `onclick` handlers etc. in your HTML.

Bad:

```
<button type="button" onclick="TYPO3.books.submitSearchForm();">
  Search
</button>
```

Good:

```
jQuery(document).ready(function () {
    if (jQuery('.tx-books-pi1').length === 0) {
        return;
    }

    TYPO3.books.initializeSearchWidget();
    TYPO3.books.initializeRegistrationForm();
    TYPO3.books.convertActionLinks();
});
```

6.3 Global names

Don't use global variables or methods. Always namespace them or put them in modules to avoid polluting the global namespace (and to avoid naming collisions).

7 Ajax

7.1 TypoScript rendering

Instead of page types (or the outdated eID calls), use Helmut's TypoScript rendering. This is faster, less work and avoids collisions with page type numbers.

Note: The Ajax requests require jQuery. The parts in the templates do not–so in theory, this will also work without jQuery.

7.1.1 composer.json

```
"require": {
  "typo3-ter/typoscript-rendering": "~1.0.5",
},
```

7.1.2 Fluid views

```
{namespace ts=Helhum\TyposcriptRendering\ViewHelpers}
<f:form method="post" name="post" enctype="multipart/form-data"
        controller="Thread" action="createPost"
        id="js-new-post-form" class="new-post-form"
        additionalAttributes="{data-action-uri:
          '{ts:uri.ajaxAction(action: \'createPost\',
            controller: \'Thread\',
            pluginName: \'CommunityMainContent\'
            )}'
          }">
```

7.1.3 JavaScript Ajax call

```
var form = formContainer.find('form');
const url = form.data('action-uri');

$.ajax({
  type: 'POST',
  url: url,
  data: data,
  success: () => {}
});
```

7.1.4 Links

Blog article: `http://insight.helhum.io/post/↩ 104880845705/dont-use-eid-ajax-dispatchers-for-your-extbase`

TypoScript rending extension source code: `https://github.com/helhum/typoscript_rendering`
Demo extension: `https://github.com/helhum/ajax_example`

8 Tools

8.1 PhpStorm

- Always use PhpStorm for editing PHP files.
- Configure and regularly use PhpStorm's code inspection feature and reduce the warnings to a minimum.
- When having a single extension as the PhpStorm project, include the TYPO3 Core source and the phpunit extension in the include path.
- If you can afford it, use the Fluid and TypoScript PhpStorms by sgalinski. They also include great code completion. They are worth every cent.
- If you can not afford the paid Fluid and TypoScript plugins, you use the free TypoScript plugin (also by sgalinski) and the Fluid schema files by Helmut Hummel:
  - `http://insight.helhum.io/post/85031122475/xml-schema-auto-completion-in-phpstorm`
  - Important: There are updated schema URLs: `http://insight.helhum.io/post/130270697975/updated-fluid-schema-urls`
  - You'll also need to add `xmlns:f` attribute with the URL to the outmost tag(s) in your templates, layouts and partials. (This is not necessary if you

8.2 Debugging

Use Xdebug for debugging. Don't use `echo`, `var_dump` etc. for this.

8.3 Code style

For converting old code to PSR-2, use php-cs-fixer. You can use the configuration from the Core or from oelib:

- `typo3_src/Build/.php_cs`
- `EXT:oelib/Configuration/PhpCsFixer/FixerConfiguration.php`

The call to fix all files in a certain directory looks like this:

```
php-cs-fixer fix --config-file Configuration/PhpCsFixer/FixerConfiguration.php .
```