FOF Developer's Guide

The compact reference to Joomla!'s Rapid Application Development framework

Nicholas Dionysopoulos

FOF Developer's Guide: The compact reference to Joomla!'s Rapid Application Development framework

Nicholas Dionysopoulos

Publication date July 2013

Table of Contents

List of Tables

Chapter 1. Introducing FOF

1. Introduction

1.1. What is FOF

FOF (Framework on Framework) is a rapid application development framework for Joomla!. Unlike other frameworks it is not standalone. It extends the Joomla! Platform instead of replacing it, featuring its own forked and extended version of the MVC classes, keeping a strong semblance to the existing Joomla! MVC API. This means that you don't have to relearn writing Joomla! extensions. Instead, you can start being productive from the first day you're using it. Our goal is to always support the officially supported LTS versions of Joomla! and not break backwards compatibility without a clear deprecation and migration path.

FOF is compatible with the database technologies used by Joomla! itself: MySQL, SQL Server (and Windows Azure SQL), PostgreSQL. In most cases you can write a component in one database server technology and have it run on the other database server technologies with minimal or no effort.

FOF is currently used by free and commercial components for Joomla! by an increasing number of developers.

1.2. Free Software means collaboration

The reason of existence of FOSS (Free and Open Source Software) is collaboration between developers. FOF is no exception; it exists because and for the community of Joomla! developers. It is provided free of charge and with all of the freedoms of the GPL for you to benefit. And in true Free Software spirit, the community aspect is very strong. Participating is easy and fun.

If you want to discuss FOF there is a Google Groups mailing list [https://groups.google.com/forum/?hl=en&fromgroups#!forum/frameworkonframework]. This is a peer discussion group where developers working with FOF can freely discuss.

If you have a feature proposal or have found a bug, but you're not sure how to code it yourself, please report it on the list.

If you have a patch feel free to fork this project on GitHub [https://github.com/akeeba/fof] (you only need a free account to do that) and send a pull request. Please remember to describe what you intended to achieve to help me review your code faster.

If you've found a cool hack (in the benign sense of the word, not the malicious one...), something missing from the documentation or have a tip which can help other developers feel free to edit the Wiki. We're all grown-ups and professionals, I believe there is no need of policing the wiki edits. If you're unsure about whether a wiki edit is appropriate, please ask on the list.

1.3. Preface to this documentation

FOF is a rapid application development framework for the Joomla! CMS. Instead of trying to completely replace Joomla!’s own API (formerly known as the Joomla! Platform) it builds upon it and extends it both in scope and features. In the end of the day it enables agony-free extension development for the Joomla! CMS.

In order to exploit the time-saving capabilities of the FOF framework to the maximum you need to understand how it's organized, the conventions used and how its different pieces work together. This documentation attempts to provide you with this knowledge.

As with every piece of documentation we had to answer two big questions: where do we start and how do we structure the content. The first question was easy to answer. Having given the presentation of the FOF framework countless times we have developed an intuitive grasp of how to start presenting it: from the abstract to the concrete.

The second question was harder to answer. Do we write a dry reference to the framework or more of a story-telling documentation which builds up its reader's knowledge? Since we are all developers we can read the code (and DocBlocks), meaning that the first option is redundant. Therefore we decided to go for the second option.

As a result this documentation does not attempt to be a complete reference, a development gospel, the one and only source of information on FOF. On the contrary, this documentation aims to be the beginning of your journey, much like a travel guide. What matters the most is the journey itself, writing your own extensions based on FOF. As you go on writing software you will be full of questions. Most of them you'll answer yourself. Some of them will be already answered in the wiki. A few of them you'll have to ask on the mailing list. In the end of the day you will be richer in knowledge. If you do dig up a golden nugget of knowledge, please do consider writing a wiki page. This way we'll all be richer and enjoy our coding trip even more.

Have fun and code on!

2. Getting started with FOF

2.1. Download and install FOF

You can download FOF as an installable Joomla! library package from our repository [https://www.akeebabackup.com/download/fof.html]. You can install it like any other extension under Joomla! 2.x and later.

Using the latest development version

You can clone a read-only copy of the Git repository of FOF in your local machine. Make sure you symlink or copy the fof directory to your dev site's libraries/fof directory. Alternatively, we publish dev releases in the dev release repository [https://www.akeebabackup.com/download/fof-dev.html]. These are installable packages but please note that they may be out of date compared to the Git HEAD. Dev releases are not published automatically and may be several revisions behind the current Git master branch.

2.2. Using it in your extension

The recommended method for including FOF in your component, module or plugin is using this short code snippet right after your defined('_JEXEC') or die() statement (Joomla! 2.x and later):

```
if (!defined('FOF_INCLUDED'))
{
    include_once JPATH_LIBRARIES . '/fof/include.php';
}
```

Alternatively, you can use the one-liner:

```
require_once JPATH_LIBRARIES . '/fof/include.php';
```

From that point onwards you can use FOF in your extension.

2.3. Installing FOF with your component

Important

Joomla! 3.2 will ship with FOF pre-installed. Developers must make sure that they do not accidentally overwrite the FOF library shipped with Joomla!. You can do that with an if(version_compare(JVERSION, '3.2.0', 'ge')) return; in your FOF installation code.

Unfortunately, Joomla! doesn't allow us to version checking before installing a library package. This means that it's your responsibility to check that there is no newer version of FOF installed in the user's site before attempting to install FOF with your extension. In the following paragraphs we are going to demonstrate one way to do that for Joomla! 2.x / 3.x component packages.

Include a directory called fof in your installation package. The directory should contain the files of the installation package's fof directory. Then, in your script.mycomponent.php file add the following method:

```
/**
 * Check if FoF is already installed and install if not
 *
 * @param   object  $parent  class calling this method
 *
 * @return  array            Array with performed actions summary
 */
private function _installFOF($parent)
{
    $src = $parent->getParent()->getPath('source');

    // Load dependencies
    JLoader::import('joomla.filesystem.file');
    JLoader::import('joomla.utilities.date');
    $source = $src . '/fof';

    if (!defined('JPATH_LIBRARIES'))
    {
        $target = JPATH_ROOT . '/libraries/fof';
    }
    else
    {
        $target = JPATH_LIBRARIES . '/fof';
    }
    $haveToInstallFOF = false;

    if (!is_dir($target))
    {
        $haveToInstallFOF = true;
    }
    else
    {
        $fofVersion = array();

        if (file_exists($target . '/version.txt'))
        {
            $rawData = JFile::read($target . '/version.txt');
            $info    = explode("\n", $rawData);
            $fofVersion['installed'] = array(
                'version'   => trim($info[0]),
                'date'      => new JDate(trim($info[1]))
            );
        }
        else
        {
            $fofVersion['installed'] = array(
```

```
            'version'   => '0.0',
            'date'      => new JDate('2011-01-01')
        );
    }

    $rawData = JFile::read($source . '/version.txt');
    $info    = explode("\n", $rawData);
    $fofVersion['package'] = array(
        'version'   => trim($info[0]),
        'date'      => new JDate(trim($info[1]))
    );

    $haveToInstallFOF = $fofVersion['package']['date']->toUNIX() > $fofVersion['instal
}

$installedFOF = false;

if ($haveToInstallFOF)
{
    $versionSource = 'package';
    $installer = new JInstaller;
    $installedFOF = $installer->install($source);
}
else
{
    $versionSource = 'installed';
}

if (!isset($fofVersion))
{
    $fofVersion = array();

    if (file_exists($target . '/version.txt'))
    {
        $rawData = JFile::read($target . '/version.txt');
        $info    = explode("\n", $rawData);
        $fofVersion['installed'] = array(
            'version'   => trim($info[0]),
            'date'      => new JDate(trim($info[1]))
        );
    }
    else
    {
        $fofVersion['installed'] = array(
            'version'   => '0.0',
            'date'      => new JDate('2011-01-01')
        );
    }

    $rawData = JFile::read($source . '/version.txt');
    $info    = explode("\n", $rawData);
    $fofVersion['package'] = array(
        'version'   => trim($info[0]),
        'date'      => new JDate(trim($info[1]))
```

```
        );
        $versionSource = 'installed';
    }

    if (!($fofVersion[$versionSource]['date'] instanceof JDate))
    {
        $fofVersion[$versionSource]['date'] = new JDate;
    }

    return array(
        'required'  => $haveToInstallFOF,
        'installed' => $installedFOF,
        'version'   => $fofVersion[$versionSource]['version'],
        'date'      => $fofVersion[$versionSource]['date']->format('Y-m-d'),
    );
}
```

You need to call it from inside your postflight() method. For example:

```
/**
 * Method to run after an install/update/uninstall method
 *
 * @param   object  $type    type of change (install, update or discover_install)
 * @param   object  $parent  class calling this method
 *
 * @return void
 */
function postflight($type, $parent)
{
        $fofInstallationStatus = $this->_installFOF($parent);
}
```

Warning

Due to a bug/feature in Joomla! 1.6 and later, your component's manifest file must start with a letter before L, otherwise Joomla! will assume that lib_fof.xml is your extension's XML manifest and install FOF instead of your extension. We suggest using the com_yourComponentName.xml convention, e.g. com_todo.xml. There is a patch pending in Joomla!'s tracker for this issue, hopefully it will be accepted sometime soon.

2.4. Sample applications

FOF comes with two sample applications which are used to demonstrate its features, To-Do [https://github.com/akeeba/todo-fof-example] and Contact Us [https://github.com/akeeba/contactus]. These were conceived and developed in different points of FOF's development. As a result they are always in a state of flux and will definitely not expose all of FOF's features.

Another good way to learn some FOF tricks is by reading the source code of existing FOF-based components. Just remember that we are all real world developers and sometimes our code is anything but academically correct ;)

3. Key Features

Some of the key features / highlights of FOF:

Convention over configuration, Rails style.

Instead of having to painstakingly code every single bit of your component, it's sufficient to use our naming conventions, inspired by Ruby on Rails conventions. For example, if you have com_example, the foobar view will read from the #__example_foobars table which has a unique key named example_foobar_id. The default implementation of controllers, models, tables and views will also cater for the majority of use cases, minimising the code you'll need to write.

HMVC today, without relearning component development.

There's a lot of talk about the need to re-engineer the MVC classes in Joomla! to support HMVC. What if we could give you HMVC support using the existing MVC classes, today, without having to relearn how to write components? Yes, it's possible with FOF. It has been possible since September 2011, actually. And for those who mumble their words and spread FUD, yes, it IS HMVC by any definition. The very existence of the FOFDispatcher class proves the point.

Easy reuse of view template files without ugly include().

More often than not you want to reuse view template files across views. The "traditional" way was by using include() or require() statements. This meant, however, that template overrides ceased working. Not any more! Using FOFView's loadAnyTemplate() you can load any view template file from the front- or back-end of your component or any other component, automatically respecting template overrides.

Automatic language loading and easy overrides.

Are you sick and tired of having to load your component's language files manually? Do you end up with a lot of untranslated strings when your translators don't catch up with your new features? Yes, that sucks. It's easy to overcome. FOF will automatically handle language loading for you.

Media files override (works like template overrides).

So far you knew that you can override Joomla!'s view template files using template overrides. But what about CSS or Javascript files? This usually required the users to "hack core", i.e. modify your views' PHP files, ending up in an unmaintainable, non-upgradeable and potentially insecure solution. Not any more! Using FOF's FOFTemplateUtils::addCSS and FOFTemplateUtils::addJS you can load your CSS and JS files directly from the view template file. Even better? You can use the equivalent of template overrides to let your users and template designers override them with their own implementations. They just have to create the directory templates/your_template/media/com_example to override the files normally found in media/com_example. So easy!

Automatic JSON and CSV views with no extra code (also useful for web services).

Why struggle to provide a remote API for your component? FOF makes the data of each view accessible as JSON feeds opening a new world of possibilities for Joomla! components (reuse data in mobile apps, Metro-style Windows 8 tiles, browser extensions, mash-up web applications, ...). The automatic CSV views work on the same principle but output data in CSV format, suitable for painlessly data importing to spreadsheets for further processing. Oh, did we mention that we already have an almost RESTful web services implementation?

No code view templates.

Don't you hate it that you have to write a different view template (in PHP and HTML) for each Joomla! version and, worse, each template out there? Don't you hate it having to teach non-developers how to not screw up your component

with every update you publish? We feel your pain. That's why FOF supports the use of XML files as view templates, rendering them automatically to HTML. Not just forms; everything, including browse (multiple items) and single item views. Even better, you get to choose if you want to use traditional PHP/HTML view templates, XML view templates or a combination of both, even in the same view!

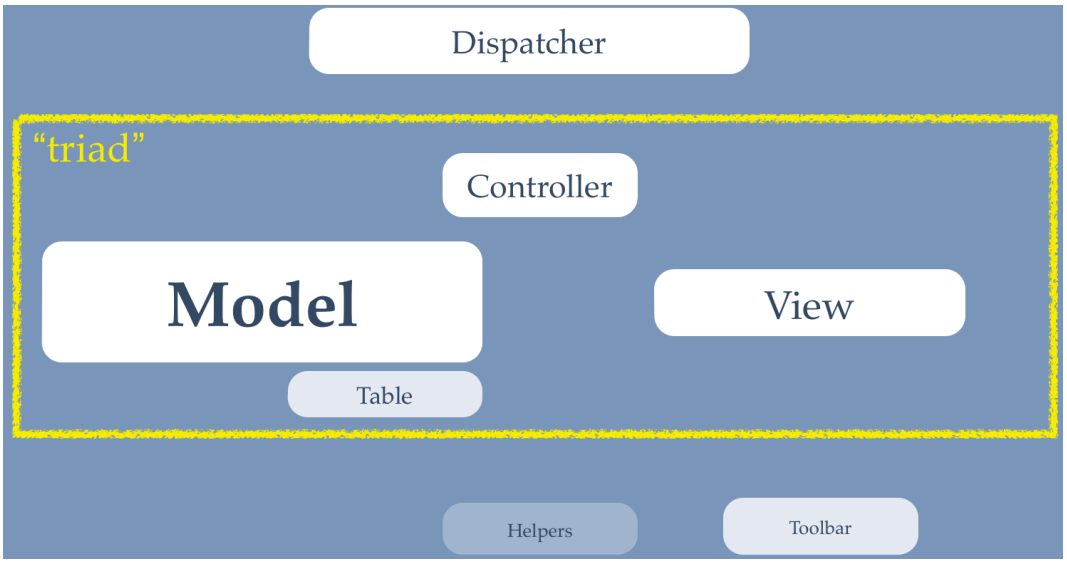
No code routing, ACL and overall application configuration.

Since FOF 2.1 you can define your application's routing, access control integration and overall configuration without routing any code, just by using a simple to understand XML file. It's now easier than ever to have Joomla! extensions with truly minimal (or no) PHP code.

Chapter 2. Component overview and reference

FOF is an MVC framework in heart and soul. It tries to stick as close as possible to the MVC conventions put forward by the Joomla! CMS since Joomla! 1.5, cutting down on unnecessary code duplication. The main premise is that your code will be DRY – not as the opposite of “wet”, but as in Don’t Repeat Yourself. Simply put: if you ever find yourself trying to copy code from a base class and paste it into a specialized class, you are doing it wrong.

In order to achieve this code isolation, FOF uses a very flexible structure for your components. A component's structure looks like this:



The Dispatcher is the entry point of your component. Some people would call this a "front Controller" and this is actually what it is. It's different than what we typically call a Controller in the sense that the Dispatcher is the only part of your component which is supposed to interface the underlying application (e.g. the Joomla! CMS) and gets to decide which Controller and task to execute based on the input data (usually this is the request data). No matter if you call it an entry point, front controller, dispatcher or Clint Eastwood its job is to figure out what needs to run and run it. We simply chose the name "Dispatcher" because, like all conventions, we had to call it something. So, basically, the Dispatcher will take a look at the input data, figure out which Controller and task to run, create an instance of it, push it the data and tell it to run the task. The Controller is expected to return the rendered data or a redirection which the Dispatcher will dully pass back to its caller.

Oh, wait, what is a Controller anyway?! Right below the Dispatcher you will see a bunch of stuff grouped as a "triad". The "triad" is commonly called "view" (with a lowercase v). Each triad does something different in your component. For example, one triad may allow you to handle clients, another triad allow you to handle orders and so on. Your component can have one or more triads. A triad usually contains a Controller, a Model and a View, hence the name ("triad" literally means "a bunch of three things"). The only mandatory member is the Controller. A triad may be reusing the Model and View from another triad – which is another reason why DRY code rocks– or it may even be view-less. So, a triad may actually be a bunch of one, two or three things, as long as it includes a Controller. Just to stop you from being confused or thinking about oriental organised crime and generally make your life easier we decided to call these "views" (with a lowercase v) instead of "triads". See? Now it is so much better.

FOF views follow the "fat Model - thin Controller" paradigm. This means that the Controller is a generally minimalist piece of code and the Model is what gets to do all the work. Knowing this very important bit of information, let's take a look at the innards of a view.

In the very beginning we have the Controller. The Controller has one or more tasks. Each task is an action of your component, like showing a list of records, showing a single record, publish a record, delete a record and so on. With a small difference. The Controller's tasks do not perform the actual work. They simply spawn an instance of the Model and push it the necessary data it needs. This is called "setting the state" of the Model. In most cases the Controller will also call a Model's method which does something. It's extremely important to note that the Controller will work with any Model that implements that method and that the Model is oblivious to the Controller. Then the Controller will create an instance of the View class, pass it the instance of the Model and tell it to go render itself. It will take the output of the View and pass it back to the Dispatcher.

Which brings us to the Model. The Model is the workhorse of the view. It implements the business logic. All FOF Models are passive Models which means that they are oblivious to the presence of the Controller and View. Actually, they are completely oblivious to the fact that they are part of a triad. That's right, Models can be used standalone, outside the context of the view or component they are designed to live in. The Model's methods will act upon the state variables which have already been set (typically, by the Controller) and will only modify the state variables or return the output directly. Models must never have to deal with input data directly or talk to specific Controllers and/or Views. Models are decoupled from everything, that's where their power lies.

Just a small interlude here. Right below the Model we see a small thing called a "Table". This is a strange beast. It's one part data adapter, one part model and one part controller, but nothing quite like any of this. The Table is used to create an object representing a single record. It is typically used to check the validity of a record before saving it to the database and post-process a record when reading it from the database (e.g. unserialise a field which contains serialised or JSON data).

The final piece of our view is the View class itself. It will ask the Model for the raw data and transform it into a suitable representation. Typically this means getting the raw records from the Model and create the HTML output, but that's not the only use for a View. A View could just as well render the data as a JSON stream, a CSV file, or even produce a graphic, audio or video file. It's what transforms the raw data into something useful, i.e. it's your presentation layer. Most often it will do so by loading view templates which are .php files which transform raw data to a suitable representation. If you are using the XML forms feature of FOF, the View will ask the Model to return the form definition and ask FOF's renderer to render this to HTML instead. Even though the actual rendering is delegated to the Renderer (not depicted above), it's still the View that's responsible for the final leg of the rendered data: passing it back to its caller. Yes, the View will actually neither output its data directly to the browser, nor talk to the underlying application. It returns the raw data back to its caller, which is almost always the Controller. Again, we have to stress that the View is oblivious to both the Controller and the Model being used. A properly written View is fully decoupled from everything else and will work with any data provider object implementing the same interface as a Model object and a caller which is supposed to capture its output for further consumption.

Important

All classes comprising a view are fully decoupled. None is aware of the internal workings of another object in the same or a different view. This allows you to exchange objects at will, promoting code reuse. Even though it sounds like a lot of work it's actually less work and pays dividends the more features you get to add to your components.

There's another bit mentioned below the triad, the Toolbar. The Toolbar is something which conceptually belongs to the component and only has to do with views being rendered as HTML. It's what renders the title in the back-end, the actions toolbar in the front- or back-end and the navigation links / menu in the back-end. In case you missed the subtle reference: FOFToolbar allows you to render an actions toolbar even in the front-end of your component, something that's not possible with plain old Joomla!. You will simply need to add some CSS to do it.

Finally we mention the Helpers. The Helpers are pure static classes implementing every bit of functionality that's neither OOP, nor can it be categorised in any other object already mentioned. For example, methods to render drop-down selection lists. In so many words, "Helper" is a polite way of saying "non-OOP cruft we'd rather not talk about". Keep your Helpers to a minimum as they're a royal pain in the rear to test.

Please do keep in mind that this is just a generic overview of how an FOF-based component works. The real power comes from the fact that you don't need to know the internal workings of FOF to use it, you don't need to copy and paste code from it (woe is the developer who does that) and quite possibly you don't even need to write any code. At all. It's all discussed later on.

1. Models

The Model is the workhorse. Business logic goes here. Models never interface input data directly or output data. They are supposed to read data from their state and push the results to their state.

Class and file naming conventions

The convention for naming the model classes is `ComponentModelView`, e.g. `TodoModelItems` for a component named `com_todo` and a view named `items`. The last part SHOULD be plural. Support for singular named models (such as `TodoModelItem`) will be dropped in a future version.

The model file MUST match the last part of the class name. This means that the file for `TodoModelItems` MUST be `items.php`, whereas the file for `TodoModelItem` MUST be `item.php`.

All Model files are located in your component's `models` directories, in the front-end and back-end. If a file is not present in the front-end, it will be attempted to be loaded from the back-end and vice versa. If the Model class is not loaded and a suitable file cannot be found FOF will fall back to one of the following, in this order:

1. The Default model. This is a special model class following the naming conventions `ComponentModelDefault`, e.g. `TodoModelDefault`, found in the `default.php` file inside your models directory.

2. If a default model is not found, FOF will fall back to creating a suitably configured instance of `FOFModel`, using convention over configuration (explained below) to determine what the model object should do.

Database table naming conventions

All FOF models connect, by default, to a database table. You can of course have a model whose corresponding table doesn't exist as long as you do not use its default data processing methods.

Database tables are named as #__component_view, e.g. #__todo_items for a component named `com_todo` and a view named `items`.

The auto increment field is named component_view_id, e.g. todo_item_id for a component named `com_todo` and a view named `items`. If your table does not have an auto incrementing field you will not be able to use the default implementation of FOF's data processing methods.

You can override defaults without copying & pasting code, ever. This is documented in Configuring MVC.

Model behaviours

Models can implement complex, reusable functionality using behaviours. Behaviours use the Observable pattern to "hook" into the onBefore*/onAfter* methods of the Model object to implement a feature, e.g. automatic application of filters based on input parameters. The bundled behaviour classes are currently located inside FOF's `model/behavior` directory. By default, only the `filters` behaviour is being loaded. You can add / modify behaviours in different ways:

- By overriding the model's __construct method and using the addBehavior() method

- With the $config or fof.xml configuration overrides, using the behaviors option key

Please note that in both cases you are providing the last part of the behaviour's name. FOF will look for a view-specific behaviour class and if it doesn't it will try falling back to its default implementation of the behaviour.

The convention for naming the view-specific model behaviour classes is `ComponentModelViewBehaviorName`, e.g. `TodoModelItemsBehaviorFilter` for a component named `com_todo`, a view named `items` and a behaviour called `filters`. The View part MUST be plural.

If the view-specific model behaviour class is not found, FOF will fall back to `FOFModelBehaviorName`, e.g. `FOFModelBehaviorFilter` if you are using the "filters" behaviour name. The built-in behaviours are discussed further down in the Built-in Behaviours section.

Customising a specialised class

Unlike plain old Joomla! you are NOT supposed to copy and paste code when dealing with FOF. Our rule of thumb is that if you ever find yourself copying code from FOFModel into your extension's specialised model class you're doing it wrong.

FOF models can be customised very easily using the `onBefore`*`Something`* / `onAfter`*`Something`* methods. The *`Something`* is the name of the model method they are related to. For example, `onBeforeSave` runs before the `save()` method executes its actions and `onAfterSave` runs right after the `save()` method executes its actions. Specific implementation notes for each case can be found in the docblocks of each event method.

Customising using plugin events

FOF models are designed to call certain plugin events (of "content" plugins) upon certain actions. The events are defined in the model's protected properties as follows:

event_before_delete (default: `onContentBeforeDelete`) is triggered before a record is deleted.

event_after_delete (default: `onContentAfterDelete`) is triggered after a record is deleted.

event_before_save (default: `onContentBeforeSave`) is triggered before a record is saved.

event_after_save (default: `onContentAfterSave`) is triggered after a record is saved.

event_change_state (default: `onContentChangeState`) is triggered after a record changes state, i.e. it's published, unpublished etc.

event_clean_cache (default: none; doesn't run) is triggered when FOF is cleaning the cache.

Moreover, if you are using XML forms you will also see the `onContentPrepareForm` event which runs when the form is being pre-processed before rendering.

These are the same as the standard Joomla! plugin events. This ensures that a plugin written for a core Joomla! component can easily be extended to handle FOF components as well.

Whenever Joomla! requires us to pass a context to the plugin events we use the conventions *`component.view`* e.g. *`com_todo.items`* for a component name `com_todo` and a model for the items `view`.

1.1. Built-in Behaviours

FOF comes with several built-in model behaviours. They are used to provide core functionality. By default only the "filters" behaviour is attached to a model for performance reasons. In this section we will discuss what each behaviour does.

You can combine multiple behaviours at once.

1.1.1. access

Adding this behaviour to a model object filters the front-end output by the viewing access levels the user has access to.

Important

This behaviour REQUIRES the "filters" behaviour. If you have not added the "filters" behaviour it will not have any effect on browse views. It will, however, work on edit and read views.

1.1.2. enabled

Adding this behaviour to a model object filters the front-end output to only items which are published (enabled=1).

1.1.3. filters

Important

This behaviour only works on browse views.

Adding this behaviour to a model object allows FOF to magically apply filters based on the input data. For example, if you pass &foobar=1 in the URL, or –more generally speaking– have a foobar state variable with a value of 1 then the SQL query used to fetch the items list will be filtered by the rows where the foobar column is set to 1.

The filters behaviour is smart enough to recognise the type of your table fields and apply the correct type filter each time. There are several different filtering methods per field type. Besides the default filtering method which is used when you only use a plain value in the state variable you can select a different method. To do that you need to pass a hash (keyed) array in the state variable like this array('method' => 'between', 'from' => 1, 'to' => 10) or, in URL query format, &foobar[method]=between&foobar[from]=1&foobar[to]=10.

So, let's discuss the available match types per field type.

Number fields

For numeric fields you can use the following filtering methods:

exact — This is the default method. You can just pass the value you want to search. If you want to use the hash array format you have the following keys:

- `method` : *`exact`*
- `value` : the value you want to search

partial — For numeric fields this is just an alias to `exact`.

between — Returns records whose field value is inside the space between two numbers, inclusive. You have the following keys:

- `method` : *`between`*
- `from` : Left barrier of the number space
- `to` : Right barrier of the number space

For example from=1 and to=10 will search for any value between 1 to 10, including 1 and 10.

outside — Returns records whose field value is outside the space between two numbers, exclusive. You have the following keys:

- `method` : *`outside`*
- `from` : Left barrier of the number space
- `to` : Right barrier of the number space

For example from=1 and to=10 will search for any value lower than 1 or greater than 10, excluding 1 and 10.

interval Returns records whose field value is following an interval (arithmetical progression)

- `method` : *`interval`*
- `value` : The starting value of the interval
- `interval` : The interval period

For example value=5 interval=2 will search for values 5, 7, 9, 11 and so on.

Boolean fields

For boolean (tiny integer) fields you can use the following methods:

exact This is the default method. You can just pass the value you want to search. If you want to use the hash array format you have the following keys:

- `method` : *`exact`*
- `value` : the value you want to search

Text fields

For text fields you can use the following methods:

partial This is the default method. You can just pass the value you want to search. The records returned have that value somewhere in their fields (partial text search). If you want to use the hash array format you have the following keys:

- `method` : *`partial`*
- `value` : the partial phrase you want to search

exact Performs an exact search. The fields' values must be exactly equal to the value you use here. You have the following keys:

- `method` : *`exact`*
- `value` : the exact phrase you want to search

Date fields

For date and date/time fields you can use the following methods:

exact This is the default method. Performs an exact search. The fields' values must be exactly equal to the value you use here. You have the following keys:

- `method` : *`exact`*

- `value` : the exact phrase you want to search

partial — You can just pass the value you want to search. The records returned have that value somewhere in their fields (partial text search). If you want to use the hash array format you have the following keys:

- `method` : *`partial`*
- `value` : the partial phrase you want to search

between — Returns records whose field value is inside the space between two dates, inclusive. You have the following keys:

- `method` : *`between`*
- `from` : Left barrier of the date space
- `to` : Right barrier of the date space

outside — Returns records whose field value is outside the space between two dates, exclusive. You have the following keys:

- `method` : *`outside`*
- `from` : Left barrier of the number space
- `to` : Right barrier of the number space

interval

Warning

This method currently only works with MySQL.

Returns records whose field value is following an interval (arithmetical progression)

- `method` : *`interval`*
- `value` : The starting value of the interval
- `interval` : The interval period. The interval can either be a string or an array. As a string it contains a sign (+ to go to the future or - to go to the past), the numeric portion of the interval period and the actual interval (days, months, years, weeks). For example:

  `+1 month` to search for values every one month in the future or *`-1 month`* to search for values every one month in the past

  As an array it can look like this array('sign' => '+', 'value' => '1', 'unit' => 'month')

1.1.4. language

Adding this behaviour to a model object filters the front-end output by language, displaying only the items whose language matches the currently enabled front-end language. Obviously this only has an effect on multi-lingual sites when the Joomla! language filter plugin is enabled.

1.1.5. private

Adding this behaviour to a model object filters the front-end output by the created_by user, showing only items that have been created by the currently logged in user. Items not created by the current user will not be displayed.

2. Tables

Tables are strange beasts. They are part data adapter, part model and part controller. Confused? They are used to create an object representing a single record of a database table. They're typically used to check the validity of a record before saving it to the database and post-process a record when reading it from the database (e.g. unserialise a field which contains serialised or JSON data). They can come in very handy to perform automated ("magic") actions when creating / modifying / loading a database record.

Class and file naming conventions

The conventions for naming the table classes is `ComponentTableView`, e.g. `TodoTableItem` for a component named `com_todo` and a view named `items`. The last part MUST be singular. It's logical: a table class operates on a single record, ergo it's singular.

The table file MUST match the last part of the class name. This means that the file for `TodoTableItem` MUST be `item.php`.

All Table files are located in your component's `tables` directories in the back-end. If the Table class is not loaded and a suitable file cannot be found FOF will fall back to creating a suitable configure instance of FOFTable, using convention over configuration (explained below) to determine what the table object should do.

Database table naming conventions

It's exactly as described in the Model reference.

Magic fields

Magic fields have special meaning for FOF. They are:

title	The title of an item. It's used for creating a slug.
slug	That's the alias of an item, typically used as part of generated URLs by your components. By default, it will be generated out of the title using a very basic transliteration algorithm.
enabled	Is this record published or not? It's like the published column in core Joomla! components, but usually it is only supposed to take values of 0 (disabled) and 1 (enabled).
ordering	The sort order of the record.
created_by	The ID of the user who created the record. Handled automatically by FOF.
created_on	The date when the record was created. Handled automatically by FOF.
modified_by	The ID of the user who last modified the record. Handled automatically by FOF.
modified_on	The date when the record was last modified. Handled automatically by FOF.
locked_by	The ID of the user who locked (checked out) the record for editing. Handled automatically by FOF.
locked_on	The date when the record was locked (checked out) for editing. Handled automatically by FOF.
hits	How many read hits this record has received. Handled automatically by FOF.
asset_id	The ID in the #__assets table for the record. Handled automatically by FOF. Only required if you want per item ACL privileges.

access Viewing access level

You can always customise the magic fields' names in your Table class using the _columnAlias array property. For example, if your column is named published instead of enabled:

```
$this->_columnAlias['enabled'] = 'published';
```

Now FOF will know that the published column contains the publish status of the record. This comes in handy when you're upgrading a component from POJ (plain old Joomla!) to FOF.

Using joined tables

Quite often you can have tables that are linked together, while you can setup joins in the list view acting on the `buildQuery`, sometimes you need these joined fields while using the table.

To do that you can simply create the join query, pass it to the `FOFTable` in the construction and you're done! FOF will do the rest.

Let's see it in action with a real an example. Let's you have the table foo linked to the table bar:

Table 2.1. Table foo

foo_id	title
1	First row
2	Second row

Table 2.2. Table bar

bar_id	title	barfield	foo_id
1	Bar table title 1	Unique column name field 1	1
2	Bar table title 2	Unique column name field 2	2

In plain sql you should do something like this:

```
SELECT jos_foo.*, jos_bar.title as bar_title, barfield
FROM jos_foo
INNER JOIN jos_bar ON jos_bar.foo_id = jos_foo.foo_id
WHERE jos_foo.foo_id = 1
```

When setting up the table, FOF already does a query like this:

```
SELECT jos_foo.*
FROM jos_foo
WHERE jos_foo.foo_id = 1
```

So we just have to "inject" the join part. We can do that extending the `FOFTable`, creating a query using Joomla! object and then assigning it to the table:

```
<?php
class FoobarTableFoo extends FOFTable
{
  public function __construct($table, $key, &$db)
  {
      $query = $db->getQuery(true)
                  ->select(array($db->qn('#__bar').'.'.$db->qn('title').' as '.$db->qn('ba
```

```
                ->select('barfield')
                ->innerJoin('#__bar ON #__bar.foo_id = #__foo.foo_id');

        $this->setQueryJoin($query);

        parent::__construct($table, $key, $db);
    }
}
```

As you can see, it's quite easy to setup: you just have to create a query with no from clause (FOF will use the current table one) and you are free to compose as you want. You can quote columns names, table names, using column or table alias (and quote them or not) and so on: it's just a regular query. FOF will now automatically know of all fields in your joined table query.

Customising a specialised class

Unlike plain old Joomla! you are NOT supposed to copy and paste code when dealing with FOF. Our rule of thumb is that if you ever find yourself copying code from `FOFTable` into your extension's specialised table class you're doing it wrong.

FOF tables can be customised very easily using the `onBefore`*`Something`* / `onAfter`*`Something`* methods. The *`Something`* is the name of the table method they are related to. For example, `onBeforeBind` runs before the `bind()` method executes its actions and `onAfterBind` runs right after the `bind()` method executes its actions. Specific implementation notes for each case can be found in the docblocks of each event method.

Customising using plugins

You can customise the actions of tables by using standard "`system`" plugins. `FOFTable` will automatically create plugin events using a fixed naming prefix and appending them with the last part of the table's name. For example, if you have a table called `TodoTableItem` FOF will attempt to run a system plugin event called `onBeforeBindItem`. For the sake of documentation we will be using the suffix *`TABLENAME`*.

The obvious drawback is the possibility of naming clashes. For example, given two tables `TodoTableItem` and `ContactusTableItem` the event to be called before binding data to either table is called `onBeforeBindItem`. How can you distinguish between the two cases? The first parameter passed to the plugin event handler is a reference to the table object itself, by convention called *`$table`*. You can do a `$table->getTableName()` which returns something like `#__todo_items`. Just check if it's the database table you expect to be interacting with. If not, just return true to let FOF do its thing uninterrupted.

The complete list of events is:

onBeforeBind*`TABLENAME`*	is triggered before binding data from an array/object to the table object.
onAfterLoad*`TABLENAME`*	is triggered after a record is loaded
onBeforeStore*`TABLENAME`*	is triggered before a record is saved to the table
onAfterStore*`TABLENAME`*	is triggered after a record has been saved to the table

onBeforeMove*TABLENAME*	is triggered before a single record is moved (reordered)
onAfterMove*TABLENAME*	is triggered after a single record is moved (reordered)
onBeforeReorder*TABLENAME*	is triggered before a new ordering is applied to multiple records of the table
onAfterReorder*TABLENAME*	is triggered before a new ordering is applied to multiple records of the table
onBeforeDelete*TABLENAME*	is triggered before a record is deleted
onAfterDelete*TABLENAME*	is triggered after a record has been deleted
onBeforeHit*TABLENAME*	is triggered before registering a read hit on a record
onAfterHit*TABLENAME*	is triggered after registering a read hit on a record
onBeforeCopy*TABLENAME*	is triggered before copying (duplicating) a record
onAfterCopy*TABLENAME*	is triggered after copying (duplicating) a record
onBeforePublish*TABLENAME*	is triggered before publishing a record
onAfterReset*TABLENAME*	is triggered after we have reset the table object's state
onBeforeReset*TABLENAME*	is triggered before resetting the table object's state

If you return boolean false from an `onBefore` event the operation is cancelled.

As you can easily understand this is an extremely powerful feature as it allows end users and site integrators (of a power user level, granted) to modify or extend the behaviour of FOF-powered extensions with great ease.

3. Controllers

The Controller is the orchestrator of each view. You can call a specific task of the Controller –based on the input variables you pass to it– causing it to execute a specific method. The Controller's job is to create a Model and View object, set the state of the Model based on the request and then either call a Model's method to perform an action (e.g. save a record) or pass the View the Model object and tell it to render itself.

Class and file naming conventions

The convention for naming the controller classes is `ComponentControllerView`, e.g. `TodoControllerItem` for a component named `com_todo` and a view named `items`. The last part SHOULD be singular. Support for plural named controllers (such as `TodoControllerItems`) will be dropped in a future version.

The controller file name MUST match the last part of the class name. This means that the file for `TodoControllerItem` MUST be `item.php`, whereas the file for `TodoControllerItems` MUST be `items.php`.

All Controller files are located in your component's controllers directories, in the front-end and back-end. If a file is not present in the front-end, it will be attempted to be loaded from the back-end and vice versa. If the Controller class is not loaded and a suitable file cannot be found FOF will fall back to one of the following, in this order:

1. The Default controller. This is a special controller class following the naming conventions *`Component`*`ControllerDefault`, e.g. `TodoControllerDefault`, found in the `default.php` file inside your `controllers` directory.

2. If a default controller is not found, FOF will fall back to creating a suitably configured instance of `FOFController`, using convention over configuration to determine what the controller object should do.

View names and handling by a single controller

The convention in FOF is that he view name is plural when you are executing the `browse` method (which returns multiple records) and singular in all other cases. Both views are considered to be part of the same triad and are handled by the same controller. For example, let's consider a component named `com_todo` and a view called `items`. The view name will be `items` when you are producing a list of all items (`browse` task), but `item` in all other cases. Both views will be handled by the `TodoControllerItem` class. This is different than plain old Joomla!. You do not need a different "list" and "form" controller. There's one and only one controller per view.

Customising a specialised class

Unlike plain old Joomla! you are NOT supposed to copy and paste code when dealing with FOF. Our rule of thumb is that if you ever find yourself copying code from FOFController into your extension's specialised table class you're doing it wrong.

FOF controller can be customised very easily using the `onBefore`*`Something`* / `onAfter`*`Something`* methods. The *`Something`* is the name of the controller task they are related to. For example, `onBeforeBrowse` runs before the `browse` task executes and `onAfterBrowse` runs right after the `browse` task executes. Returning false will result in a 403 Forbidden error. Specific implementation notes for each case can be found in the docblocks of each event method.

Extending your controllers with plugin events

As a developer you've probably found yourself in a position like this: a component you found does almost what you want. In order to make it do exactly what you want you need to change how a controller handles a specific task in a specific view. But if you modify the controller ("hack core") you have upgrade and maintenance issues. You can make a feature request to the developer but you don't know if and when the feature will be implemented. If you are the developer of the component you are faced with the dilemma: do I let this client down or do I implement a feature that doesn't quite fit my extension and will become a maintenance burden?

This is where FOF kicks in. Remember how you can customise a specialised class with `onBefore` / `onAfter` methods? Since FOF 2.1.0 you can handle these methods not only with a customised class but also with system plugin events. System plugins are always loaded early in the Joomla load process (as early as the onAfterInitialise call),

making them an excellent choice for providing component customisation code, without the need to over-engineer FOF's handling of Controllers.

You will need to create a method named `onBefore`*`Component`*`Controller`*`ViewTask`*, e.g. `onBeforeFoobarControllerItemsRead` to handle the `onBefore` event of `com_foobar`, items `view`, `read` task. Returning `false` will prevent the task from firing. You can do the same for the `onAfter` event. Do note that plugin events run after the code in your controller and not instead of it or before it. The events must be implemented in system plugins so that they always get loaded by Joomla! before any controller gets the chance to run (remember, HMVC, you may end up calling a controller from a module or plugin).

The signature for these plugins methods is like this:

```
onBefore*Component*Controller*ViewTask* (FOFController &$controller, FOFInput &$input)
```

Both parameters are passed by reference, meaning that you can modify them from your plugin. There's a caveat: by the time the `onBefore` plugin event is called the model and view instances have already been created with the previously existing `FOFInput` instance. If you need to modify the model's state you will have to do something like `$controller->getThisModel()->setState('foo', $myNewFooValue)`

```
onAfter*Component*Controller*ViewTask* (FOFController &$controller, FOFInput &$input, &$ret)
```

The *`$ret`* parameter contains the return value of the task method. It is passed by reference and you can modify it from your plugin.

4. Views

The Views are the last wheel of an MVC triad. Their sole purpose in life is to render the data in a suitable representation that makes sense. Usually this means rendering to HTML but they can also be used to render the data as JSON, XML, CSV or even as images, sound and video. It's up to you to decide what a "suitable" representation means in the context of your application.

Class and file naming conventions

The convention for naming the view classes is *`Component`*`View`*`Viewname`*, e.g. `TodoViewItem` for a component named `com_todo` and a view named `item`. The last part MUST match the singular/plural name of the specific view you are rendering.

The view file name MUST follow the convention `view.`*`format`*`.php`, e.g. `view.html.php`. The *`format`* is the representation format rendered by this view class. The most common formats –for which FOF provides default implementations– are `html`, `json` and `csv`. The format MUST match the value of the "format" input variable. If none is specified, `html` will be assumed. Exception: there is a format called `form` which is an HTML rendering and will be loaded when the value of the format input variable is set to `html` (or not set at all) as long as there is an XML form for this view.

All View files are located in your component's views directories, in the respective front-end and back-end directory, inside the respective view subfolder. For example, if you have a component called `com_todo` and a back-end view named `items` the view file for the HTML rendering is `administrator/components/com_todo/views/items/view.html.php`

If a file is not present in the front-end, it will be attempted to be loaded from the back-end and vice versa. If the View class is not loaded and a suitable file cannot be found FOF will fall back to one of the following, in this order:

1. The Default view. This is a special controller class following the naming conventions *`Component`*`ViewDefault`, e.g. `TodoViewDefault`, found in the `default/view.`*`format`*`.php` file inside your `views` directory.

2. If a default view is not found, FOF will fall back to creating a suitably configured instance of a `FOFView` format-specific class, using convention over configuration to determine what the model object should do. For example, if the current format is `html` FOF will create an instance of `FOFViewHtml`. If there is no suitable class found you will get an error as FOF has no idea what to render.

View template files and their location

FOF uses default names to generate a list or form to edit, these names are linked to the task being executed.

Table 2.3. View templates' locations

Task name	Filename	Description
`browse`	`default.php` OR `form.default.xml`	This is the file that shows the list page
`read`	`form.php` OR `form.form.xml`	This is the file that shows the edit page
`edit`	`item.php` OR `form.item.xml`	This is the file that shows the data of a single record without being able to edit the record

The location of the files is also pre-defined and based on the view name. This name comes in both singular and plural form where the singular name represents the edit page and the plural name represents the list page. Let's say our view is called `todo`, the template files can be found in the following location:

```
views
|-- todo
 |-- tmpl
     form.php  OR  form.form.xml
     item.php  OR  form.item.xml
|-- todos
 |-- tmpl
     default.php  OR  form.default.xml
```

Customising a specialised class

Unlike plain old Joomla! you are NOT supposed to copy and paste code when dealing with FOF. Our rule of thumb is that if you ever find yourself copying code from any FOFView class into your extension's specialised table class you're doing it wrong.

FOF view can be customised very easily using the `on`*`Task`* methods. The *`Task`* here is the name of the controller task they are related to. For example, `onBrowse` runs when rendering the output of a browse task. There is a catch-all method called `onDisplay` which executes if no suitable method is found in the view class. Returning false from these methods will result in a 403 Forbidden error.

Layouts, sub-templates and template overrides

The default filename of the template file to be used can be overridden with the *`layout`* input variable. For example if you feed an input variable named `layout` with a value of `foobar` FOF will look for `form.foobar.xml` and `foobar.php` in the `tmpl` directory.

You can also specify sub-templates using the *`tpl`* parameter when calling the `display()` method of your view class. By default FOF doesn't use it at all. You can only use it with custom controller tasks. In this case the tpl (a.k.a. subtemplate) will be appended to the layout name with an underscore in between. So for `layout=foo` and `tpl=bar` FOF will be looking for the `foo_bar.php` view template file.

All view template files are subject to template overrides. The view template will first be searched in the `templates/`*`template`*`/html/`*`component`*`/view` directory where *`template`* is the name of your template, *`component`* is the name of your component (e.g. `com_todo`) and *`view`* is the name of your view. This allows end users and site integrators to provide customised renderings suitable for their sites.

Joomla! version specific overrides

It is possible have different view templates per Joomla! version or version family. The correct view template is chosen automatically, without you writing a single line of code.

Let's say that you have a browse view with your lovely `default.php` view template file. And you want your component to work on Joomla! 2.5 and 3.x. Oh, the horror! The markup is different for each Joomla! version, Javascript has changed, different features are available… Well, no problem! FOF will automatically search for view template files (or XML forms) suffixed with the Joomla! version family or version number.

For example, if you're running under Joomla! 2.5, FoF will look for `default.j25.php`, `default.j2.php` and `default.php` in this order. If you're running under Joomla! 3.2, FOF will look for `default.j32.php`, `default.j3.php` and `default.php` in this order. This allows you to have a different view template file for each version family of Joomla! without ugly if-blocks and awkward code.

This feature also works with XML forms, e.g. on Joomla! 2.5 a browse form will be looked for in `form.default.j25.xml`, `form.default.j2.xml` and `form.default.xml` in this order.

Automatic views and web services

FOF can automatically render your component's output in JSON and CSV formats. You do not have to write any code whatsoever. Just pass on an input variable named `format` with a value of `json` or `csv` respectively. In the typical case where you get the input variables from the request this means appending `&format=json` or `&format=csv` respectively. You can, of course, customise the output of either format using view classes if you need to.

The JSON format can be used to provide web services with integrated hypermedia (following the HAL specification). All you need to do is to tell `FOFViewJson` to use hypermedia, either by setting `$this->useHypermedia = true;` in your specialised JSON view class or, much easier, using the `fof.xml` configuration file.

5. Dispatcher

The Dispatcher is what handles the request on behalf of your component (be it a web request or an HMVC request). Its primary job is to decide which controller to create and which task to run. Its secondary job is to handle transparent authentication which comes in really handy if you want to perform remote requests to your component, interacting with access-restricted data or actions (viewing items protected behind a login, performing privileged operations such as creating / editing / deleting records and so on).

Class and file naming conventions

The convention for naming the dispatcher classes is *`Component`*`Dispatcher`, e.g. `TodoDispatcher` for a component named `com_todo`. The last part MUST be `Dispatcher`.

The controller file name MUST be `dispatcher.php`. All Dispatcher files are located in your component's main front-end or back-end directories. If a file is not present in the front-end, it will be attempted to be loaded from the back-end but NOT vice versa. If the Dispatcher class is not loaded and a suitable file cannot be found FOF will fall back to creating a suitably configured instance of `FOFDispatcher`, using convention over configuration to determine what the Dispatcher object should do.

Customising a specialised class

Unlike plain old Joomla! you are NOT supposed to copy and paste code when dealing with FOF. Our rule of thumb is that if you ever find yourself copying code from FOFDispatcher into your extension's specialised table class you're doing it wrong.

FOF dispatcher can be customised very easily using the `onBeforeDispatch` / `onAfterDispatch` methods. `onBeforeDispatch` runs before the dispatcher executes and `onAfterDispatch` runs right after the dispatcher executes. Returning false will result in a 403 Forbidden error. Specific implementation notes for each case can be found in the docblocks of each event method.

5.1. Transparent authentication

Transparent authentication allows FOF to authenticate a user using Basic Authentication or URL parameters. This allows you to create web services or directly access pages which require a logged in users without using Joomla! session cookies.

The authentication credentials can be provided via two methods: Basic Authentication or a URL parameter. The authentication credentials can either be a username and password pair transmitted in plaintext (not recommended unless you are forcibly using HTTPS with a commercially signed SSL certificate) or encrypted. The encrypted information uses Time-Based One Time Passwords (TOTP) to allow you to communicate the credentials securely, without the burden of public key cryptography, while at the same time maintaining an intrinsically very narrow window of opportunity. Furthermore, since the effective encryption key is modified every few seconds it makes an attack against it slightly harder than using regular symmetric AES-128 cryptography.

Transparent authentication is enabled by default, but doesn't use TOTP.

Setting it up

Setting up transparent authentication requires you to modify your component's Dispatcher class, namely its `__construct()` method, to change the values of some protected fields.

The available fields are:

$_fofAuth_timeStep
: The time step, in seconds, for the time based one time passwords (TOTP) used for encryption. The default value is 6 seconds. The window of opportunity for an attacker is 2x-3x as much, i.e. 12-18 seconds using the default value. This is adequately high to be practical and too low to allow a realistic attack by a hacker.

  Important

  If you change this option you have to notify the consumers of the service to make the same change, otherwise your TOTPs will be vastly different and communication will fail.

$_fofAuth_Key
: The Base32 encoded key for TOTP. Please note that this is Base32, not Base64. Only required if you're going to use encryption.

$_fofAuth_Formats
: Which result formats should be handled by the transparent authentication. This is an array, by default `array('json', 'csv', 'xml', 'raw')`. We recommend only using non-HTML formats in here.

$_fofAuth_LogoutOnError
: By default it's true and it means that once the component finishes executing, FOF will log out the user it authenticated using transparent authentication. This is a precaution against someone intercepting and abusing the session cookie Joomla! will be sending back to the client, as well as preventing the sessions table from filling up.

$_fofAuth_AuthMethods An array of supported authentication methods. Only use the ones that make sense for your application. Avoid using the *_Plaintext ones, please. The possible values in the array are:

- **HTTPBasicAuth_TOTP** HTTP Basic Authentication using encrypted information protected with a TOTP (the username must be "_fof_auth")
- **QueryString_TOTP** Encrypted information protected with a TOTP passed in the *_fofauthentication* query string parameter
- **HTTPBasicAuth_Plaintext** HTTP Basic Authentication using a username and password pair in plain text
- **QueryString_Plaintext Plaintext** JSON-encoded username and password pair passed in the *_fofauthentication* query string parameter

When you are using the QueryString_TOTP method you can pass your authentication information as GET or POST variable called *_fofauthentication* with the value being the URL encoded cryptogram of the authentication credentials (see further down).

How to get a TOTP key

Any Base32 string can be used as a TOTP key as long as it expands to exactly 10 characters. If you don't feel like guessing, you can simply do:

```
$totp = new FOFEncryptTotp();
$secret = $totp->generateSecret();
```

You have to share this secret key with all clients wishing to connect to your component via a secure channel. This secret key must also be set in the _fofAuth_Key variable.

How to construct and supply an authentication set

The authentication set is a representation of the username and password of the user you want FOF to log in using transparent authentication. Its format depends on the authentication method.

Before going into much detail, we should consider an FOF authentication key to be a JSON-encoded object containing the keys username and password. E.g.:

```
{ “username”: “sample_user”, “password”: “$3Cr3+” }
```

This is used with all but one authentication methods. Encryption of the FOF authentication key, used with all *_TOTP methods, is discussed further down this document.

If you are using HTTPBasicAuth_Plaintext method, you have to supply your username and password using HTTP Basic Authentication. The username is the username of the user you want to log in and the password is the password of the user you want to log in. This is the easiest and most insecure authentication method.

If you are using the HTTPBasicAuth_TOTP method, you have to supply a username of _fof_auth (including the leading underscore) and as the password enter the encrypted FOF authentication key.

If you are using the QueryString_Plaintext method you have to supply a GET or POST query parameter with a name of *_fofauthentication* (including the leading underscore). Its value must be the URL encoded FOF authentication key.

If you are using the QueryString_TOTP method you have to supply a GET or POST query parameter with a name of *_fofauthentication* (including the leading underscore). Its value must be the URL encoded FOF authentication key.

Encrypting the FOF authentication key

Assuming you are doing this from a FOF-powered component, you can do something like this:

```
$timeStep = 6; // Change this if you have a different value in your Dispatcher
$authKey = json_encode(array(
  'username' => $username,
  'password' => $password
));
$totp = new FOFEncryptTotp($timeStep);
$otp = $totp->getCode($secretKey);
$cryptoKey = hash('sha256', $this->_fofAuth_Key.$otp);
$aes = new FOFEncryptAes($cryptoKey);
$encryptedAuthKey = $aes->encryptString($authKey);
```

If you can get your hands on a TOTP and AES-256 implementation for your favourite programming language you can use talk to FOF-powered components through transparent authentication. Tip: TOTP libraries are usually labelled as being Google Authenticator libraries. Google Authenticator simply uses TOTP with a temp step of 30 seconds. Most such libraries are able to change the time step, thus possible to use with FOF. In fact, that's how FOF's TOTP library was derived.

6. Toolbar

The Toolbar is the part of your components which handles the display of the component's title and toolbar buttons, as well as the toolbar submenu (links or tabs under the toolbar). While usually used in the back-end of your site, FOF components can readily render a toolbar in the front-end part of the component as well. Do note that you will need to provide your own CSS to style the toolbar in the front-end as Joomla! templates lack such a styling.

Class and file naming conventions

The convention for naming the toolbar classes is `Component`Toolbar, e.g. `TodoToolbar` for a component named `com_todo`. The last part MUST be `Toolbar`.

The controller file name MUST be `toolbar.php`. All Toolbar files are located in your component's main directory, in the front-end and back-end. If a file is not present in the front-end, it will be attempted to be loaded from the back-end. If the Toolbar class is not loaded and a suitable file cannot be found FOF will fall back to creating a suitably configured instance of FOFToolbar, using convention over configuration to determine what the controller object should do.

Customising a specialised class

FOF toolbar can be customised very easily using methods following one of the following conventions, from most specific to least specific:

on*View-nameTaskname*	for example `onItemsBrowse`. The name consists of the word on in lowercase, followed by camel cased view and task names, in this order. When the task is `Browse` the view name MUST be plural. For any other task the view name MUST be singular. For example: `onItemsBrowse` and `onItemAdd`
on*Viewname*	for example `onItems`. The name consists of the word on in lowercase, followed by camel cased view name.
on*Taskname*	for example `onBrowse`. The name consists of the word on in lowercase, followed by camel cased task name.

The method to be called is selected from the most to the least specific. For example, if you have a component named `com_todo` and a view named `items`, with the task browse being called FOF will search for the following method names, in this order: `onItemsBrowse`, `onItems`, `onBrowse`

Please note that any of these methods should only modify the toolbar and not perform any other kind of data processing.

Customising the link bar

The link bar is the area normally displayed right below the toolbar in the back-end of the site. It is usually rendered as flat links (Joomla! 2.5), a left-hand sidebar (Joomla! 3.0 and later) or tabs (when using Akeeba Strapper). The exact rendering depends on the template. The interesting thing is how these links are populated, described below.

Automatically populated link bar

FOF will normally look inside your component's views directory and look for plural views. These views are automatically added to the link bar in alphabetical order. Exception: a view called `cpanel` will always be added to the link bar.

If you want a view to not be included in the link bar, please create a file named `skip.xml` and put it inside its directory. FOF will see that and refrain from adding this view to the link bar.

If you want to modify the ordering of a view you have to create or modify the `metadata.xml` file inside your view's directory. The `<foflib>` section inside the `metadata.xml` file is read by FOF. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<metadata>
    <foflib>
        <ordering>12</ordering>
    </foflib>
    <view title="COM_FOOBAR_VIEW_ITEMS_TITLE">
        <message><![CDATA[COM_FOOBAR_VIEW_ITEMS_DESC]]></message>
    </view>
</metadata>
```

tells FOF that this view should be the 12th link in the link bar.

If you're not using a `metadata.xml` file and have a view called `cpanel` or `cpanels` then it will always be reordered to the top of the link bar list.

Fully customised link bar

The automatically generated link bar is usually enough, but sometimes you want a more complex presentation. For example, you want to show different link bars depending on a configuration setting (e.g. a "Power user" switch in your component's options), or a drop-down menu. To this end, `FOFToolbar` provides the following methods.

```
public function clearLinks()
```

Removes all links from the link bar, allowing you to start from a clean slate.

```
public function &getLinks()
```

Returns the raw data for the links in the link bar. We recommend against using it as the internal data structure may change in the future.

```
public function appendLink($name, $link = null, $active = false, $icon = null, $parent = '
```

Appends a link to the link bar. If you use the last option (`$parent`) you are creating a submenu item whose parent is the `$parent` item. You reference the parent item by its name (i.e. the `$name` parameter you used in the parent element). Drop-downs only work in a. Joomla! 3.0 and later without any additional requirements; or b. Joomla! 2.5 but only when using the optional Akeeba Strapper package which back ports jQuery and Bootstrap to Joomla! 2.5 sites.

In order to use these methods you will have to override the `renderSubmenu` method in `FOFToolbar`.

When the link bar is rendered

The link bar is rendered in all HTML views, unless you are have an input variable named *`tmpl`* with a value of `component`. Typically, this means that you are passing a query string parameter `&tmpl=component` to the URL of your component.

You can force the entire toolbar (and, by extent, the link bar) to be displayed or hidden using the *`render_toolbar`* input variable. When you set it to 0 the toolbar and link bar will not be displayed. When you set it to 1 the toolbar and link bar will be displayed (even when you use `tmpl=component`).

7. HMVC

Before we say anything else, let's define what HMVC means in the context of FOF. The H stands for "Hierarchical". That is to say there's a hierarchy of MVC calls. In very simple terms, HMVC allows you to call an MVC triad from anywhere else.

Practical uses:

- Showing a component's view inside a module, without having to rewrite the model and view logic inside the module.
- Allowing a plugin (e.g. a system or content plugin) to use the rendered output of a component and inject it to the output of the page or send it as an email.
- Displaying a view of the same or a different component within a component.

The possibilities are endless.

How to use it?

You already know it, you just didn't realise it. Here's the secret sauce:

```
FOFDispatcher::getTmpInstance('com_foobar', 'items', array('layout' => 'fancy'))->dispatch
```

You are simply creating an instance of the dispatcher of the component you want, telling it which view to render and giving it an option configuration array (the last argument in the method call). Then you just call the dispatch() method and let it render.

If you want to get the output in a variable you have to do something like this:

```
@ob_start();
FOFDispatcher::getTmpInstance('com_foobar', 'items', array('layout' => 'fancy'))->dispatch
$result = ob_end_clean();
```

If you need to pass input variables to the dispatcher you can do something like this:

```
FOFDispatcher::getTmpInstance('com_foobar', 'items', array('input' => $input))->dispatch()
```

where `$input` can be an indexed array, a stdClass object or –preferred– a `FOFInput` or `JInput` instance. For example:

```
$inputvars = array(
    'limit'            => 10,
    'limitstart'    => 0,
    'foobar'        => 'baz'
);
$input = new FOFInput($inputvars);
FOFDispatcher::getTmpInstance('com_foobar', 'items', array('input' => $input))->dispatch()
```

And, of course, you can mix and match all of the above ideas to something like:

```
$inputvars = array(
    'limit'            => 10,
    'limitstart'    => 0,
    'format'        => 'json'
);
$input = new FOFInput($inputvars);
@ob_start();
FOFDispatcher::getTmpInstance('com_foobar', 'items', array('input' => $input))->dispatch()
$json = ob_end_clean();
```

See the awesome thing we just did? We got the first 10 items of com_foobar in JSON format in the $json variable. Just a side note. This example also screws up the document MIME type if you use it in an HTML view. Be warned.

Chapter 3. Features reference

1. Configuring MVC

All MVC and associated classes in FOF (Dispatcher, Controller, Model, View, Table, Toolbar) come with a default behavior, for example where to look for model files, how to handle request data and so on. While this is fine most of the times –as long as you follow FOF's conventions– this is not always desirable.

For example, if you are building a CCK (something like K2) you may want to look for view templates in a non-standard directory in order to support alternative "themes". Or, maybe, if you're building a contact component you only want to expose the add view to your front-end users so that they can file a contact request but not view other people's contact requests. You get the idea.

The traditional approach to development prescribes overriding classes, even to the extent of copying and pasting code. If you've ever attended one of my presentations you've probably figured that I consider copying and pasting code a mortal sin. You may have also figured that, like all developers, I am lazy and dislike writing lots of code. Naturally, FOF being a RAD tool it provides an elegant solution to this problem. The `$config` array and its sibling, the `fof.xml` file.

1.1. The $config array

You may have observed that FOF's MVC classes can be passed an optional array parameter `$config`. This is a hash array with configuration options. It is being passed from the Dispatcher to the Controller and from there to the Model, View and Table classes. Essentially, this is your view (MVC triad) configuration. Setting its options allows you to modify FOF's internal workings without writing code.

The various possible settings are explained in The configuration settings section below.

1.2. The fof.xml file

The `$config` array is a great idea but has a major drawback: you have to create one or several .php files with specialized classes to use it. Remember the FOF promise about not having to write code unless absolutely necessary? Yep, this doesn't stick very well with that promise. So the `fof.xml` file was born in FOF 2.1.

The `fof.xml` file is a simple XML file placed inside your component's back-end directory, e.g. `administrator/com_example/fof.xml`. It contains configuration overrides for the front-end, back-end and CLI parts of your FOF component.

A sample fof.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<fof>
    <!-- Common settings -->
    <common>
        <!-- Table options common to all tables -->
        <table name="*">
            <field name="locked_by">checked_out</field>
            <field name="locked_on">checked_out_time</field>
        </table>
        <!-- Table options for a specific table -->
```

```
        <table name="item">
            <field name="enabled">published</field>
        </table>
    </common>

    <!-- Component back-end options -->
    <backend>
        <!-- Dispatcher options -->
        <dispatcher>
            <option name="default_view">items</option>
        </dispatcher>
    </backend>

    <!-- Component front-end options -->
    <frontend>
        <!-- Dispatcher options -->
        <dispatcher>
            <option name="default_view">item</option>
        </dispatcher>
        <!-- Options common for all views -->
        <view name="*">
            <!-- Per-task ACL settings. The star task sets the default ACL privileges for
            <acl>
                <task name="*">false</task>
            </acl>
        </view>
        <view name="item">
            <!-- Task mapping -->
            <taskmap>
                <task name="list">browse</task>
            </taskmap>
            <!-- Per-task ACL settings. An empty string removes ACL checks. -->
            <acl>
                <!-- Everyone, including guests, can access dosomething -->
                <task name="dosomething"></task>
                <!-- Only people with the core.manage privilege can access the somethingel
                <task name="somethingelse">core.manage</task>
            </acl>
            <!-- Configuration options for the model and view -->
            <config>
                <option name="behaviors">filter,access</option>
            </config>
        </view>
    </frontend>
</fof>
```

The `fof.xml` file has an `<fof>` root element. Inside it you can have zero or one tags called `<frontend>`, `<backend>` and `<cli>` which configure FOF for front-end, back-end and CLI access respectively. You may also have a tag named `<common>` which defines settings applicable for any mode of access. These common settings will be overridden by the corresponding settings defined in the `<frontend>`, `<backend>` and `<cli>`. Please note that the CLI is yet another special case: it will mix the common, back-end and CLI settings to derive the final configuration. In the other two cases (front- or back-end access) only the common and the configuration for this specific mode of access will be used.

1.2.1. Dispatcher settings

You can configure the way the Dispatcher works using the `<dispatcher>` tag. Inside it you can have one or more `<option>` tags. The name attribute defines the name of the configuration variable to set, while the tag's content defines the value of this configuration variable.

The available variables are:

default_view — Defines the default view to show if none is defined in the input data. By default this is cpanel. In the example above we set it to items in the back-end and item in the front-end.

1.2.2. Table settings

The table settings allow you to set up table options, in case you do not wish to use the default conventions of FOF. You define the table the options apply to using the name attribute of the view tag. Please note that this is the name used in the class, not in the database. So, if you have a database table named #__example_items and your class is named `ExampleTableItem` you must use `name="item"` in the `fof.xml` file.

The settings of each table are isolated from the settings of every other table, with one notable exception: the star table, i.e. `name="*"`. This is a placeholder table that defines the default settings. These settings are applied to all tables. If you also have a table tag for a view, the default settings (from the star table) and the settings for the particular table are merged together. This applies to all settings described below.

Field map settings

The field map settings allow you to map specific magic field names to your table's fields, in case you do not use FOF's contentions. It works the same way as adding setting the `_columnAlias` array in your specialized Table class.

The field map is enclosed inside the `<table>` tag itself. It consists of one or more `<field>` tags. The name attribute defines the name of the magic (FOF convention) field to map, whereas the content of the tag defines the name of the field in your database table.

1.2.3. View settings

There are several options that are applied per view. In the context of the `fof.xml` file, a “view” actually refers to an MVC triad, not just the View part of the triad. In so many words, the options affect the Controller, Model and View used to render this particular component view. You define the view the options apply to using the name attribute of the view tag.

The settings of each view are isolated from the settings of every other view, with one notable exception: the star view, i.e. `name="*"`. This is a placeholder view that defines the default settings. These settings are applied to all views. If you also have a view tag for a view, the default settings (from the star view) and the settings for the particular view are merged together. This applies to all settings described below.

Task map settings

The task map settings allow you to map specific tasks to specific Controller methods. Other frameworks would call this the “routing” feature. It works the same way as adding running registerTask in your specialized Controller class.

The task map is enclosed inside a single `<taskmap>` tag. You can have exactly zero or one `<taskmap>` tags inside each view tag.

Inside the `<taskmap>` tag you can have one or more `<task>` tags. The name attribute defines the name of the task to map, whereas the content of the tag defines the controller’s method which will be called for this task.

ACL settings

The ACL settings can be used to override or fine-tune the access control for each task of the particular view. Even though FOF comes with default ACL mappings for its basic tasks, these are not always sufficient or appropriate for all situations. Normally this is achieved by overriding the onBefore methods in the Controller, e.g. `onBeforeSave` to set up the ACL checks for the save task. You can use the ACL mappings in the `fof.xml` instead of such checks. You can even use the ACL mapping in `fof.xml` for custom tasks for which no onBefore method exists.

The ACL settings are enclosed inside a single `<acl>` tag. You can have exactly zero or one `<acl>` tags inside each view tag.

Inside the `<acl>` tag you can have one or more `<task>` tags. The `name` attribute defines the name of the task to apply the access control, whereas the content of the tag defines the Joomla! ACL privilege required to access this task. You can use any core ACL privilege or any custom ACL privilege defined in your component's `access.xml` file. If you leave the content blank then no ACL check is performed (the task is always accessible by all users). If you use the special value `false` then the ACL privilege is always going to fail, i.e. the task will not be accessible by any user.

Option settings

The configuration options of views and models can be modified directly from the view definition of `fof.xml`. The configuration settings are enclosed inside a single `<config>` tag. Inside it you can have one or more `<option>` tags. Each tag is equivalent to passing a value in the `$config` array. The `name` attribute defines the name of the configuration setting you want to modify. The content of the tag is the value of this setting. See the Configuration settings section below for more information on what each setting is supposed to do.

1.3. Configuration settings

The following settings can be used either in the $config array passed to a Dispatcher, Controller, Model or View class or in the `fof.xml` file's `<option>` tags inside the `<view>` tags.

autoRouting — A bit mask which defines the automatic URL routing of redirections.

A value of 1 means that front-end redirections will be put through Joomla!'s `JRoute::_()`.

A value of 2 means that back-end redirections will be put through Joomla!'s `JRoute::_()`.

You can combine multiple values by adding them together.

asset_key — The key to be used for ACL assets. This is typically in the form *`component.view`*, e.g. `com_example.item` This is only used for per-item ACL privileges. If you do not specify an asset key, the default *`component.view`* convention will be used instead.

base_path — The base path of the component.

- In `$config`: Specify the absolute path.
- In `fof.xml`: Specify a path relative to the site's root.

behaviors — Add model behaviours. See the FOFModel documentation for more information on behaviours.

- In `$config`: An array containing the names of model behaviours to add
- In `fof.xml`: A comma separated list with the names of model behaviours to add

cacheableTasks — A comma separated list of tasks which support Joomla!'s caching.

cid — Define a comma separated list of item IDs to limit the view on. Normally this is empty. Only use when you want to limit a view to very specific items. Only valid in the `fof.xml` file.

csrf_protection — Should we be doing a token check for the tasks of this view? The possible values are:

- 0 - no token checks are performed
- 1 - token checks are always performed
- 2 - token checks are always performed in the back-end and in the front-end, but only when the request format is `html` (default setting)
- 3 - token checks are performer only in the back-end

default_task — The task to execute if none is defined. The default value is `display`.

event_after_delete — The content plugin event to trigger after deleting the data. Default: `onContentAfterDelete`

event_after_save — The content plugin event to trigger after saving the data. Default: `onContentAfterSave`

event_before_delete — The content plugin event to trigger before deleting the data. Default: `onContentBeforeDelete`

event_before_save — The content plugin event to trigger before saving the data. Default: `onContentBeforeSave`

event_change_state — The content plugin event to trigger after changing the published state of the data. Default: `onContentChangeState`

event_clean_cache — The content plugin event to trigger when cleaning cache. There is no default value.

helper_path — The path where the View will be looking for helper classes. By default it's the helpers directory inside your component's directory.

- In `$config`: Specify an absolute path.
- In `fof.xml`: Specify a path relative to the component's directory.

id — Define an item ID to limit the view on. Normally this is empty. Only use when you want to limit a view to one single item. Only valid in the `fof.xml` file.

ignore_request — Set to 1 to prevent the Model's `populateState()` method from running. By default the method is empty and does nothing, as the Model is supposed to be decoupled from the request information, having the Controller push state variables to it.

layout — The default layout to use for this view. This is normally determined automatically based on the task currently being executed.

model_path — The path where the Controller will be looking for Model class files. By default it's the models directory of the component's directory.

- In `$config`: Specify an absolute path.
- In `fof.xml`: Specify a path relative to the component's directory.

model_prefix — The naming prefix for the Model to be loaded by the Controller. The default option is *`Componentname`*`Model` where *`Componentname`* is the name of the component without the `com_` prefix.

modelName — The name of the Model class to load. Automatically defined based on the component and view names.

searchpath — The path where Controller classes will be searched for. By default it's the `controllers` directory inside your component's directory.

- In `$config`: Specify the absolute path.
- In `fof.xml`: Specify a path relative to the component's root directory.

table_path — The path where the Model will be looking for table classes. By default it's the tables directory inside your component's directory.

- In $config: Specify an absolute path.
- In fof.xml: Specify a path relative to the component's directory.

table — Set the name of the table class the Model will use. Please note that the the component name is added to this name automatically. For example, given a component `com_example` and a table setting of `foobar` the actual table class which will be used will be `ExampleTableFoobar`.

tbl — The name of the database table to use in the table class of this view. It is in the format of #__tablename, e.g. #__example_items

tbl_key — The name of the key field of the database table to use in the table class of this view. It is in the format of component_view_id, e.g. example_item_id.

template_path — The path where the View will be looking for view template (.php) or form (.xml) files. By default it's the `tmpl` directory inside the current view's directory.

- In `$config`: Specify an absolute path.
- In `fof.xml`: Specify a path relative to the component's directory.

use_table_cache — By default FOF caches the names of the tables in the database and their field definitions in the file *`JPATH_CACHE`*`/fof/cache.php`, where JPATH_CACHE is usually the cache directory in the front- or back-end of your site respectively. If you've set *`Debug System`* to `Yes` in your site's Global Configuration then by default the cache is not used.

You can override this behaviour per view / for all views of a component using this parameter. Set to 0 to force the cache to never be used or set it to 1 to force the cache to always be used (even when your site is in debug mode).

view_path — The path where the Controller will be looking for View class files. By default it's the `views` directory inside your component's directory.

- In `$config`: Specify an absolute path.
- In `fof.xml`: Specify a path relative to the component's directory.

viewName — The name of the View class to load. Automatically defined based on the component and view names.

2. XML Forms

Traditionally, creating view templates involves a .php file where PHP and HTML code are intermixed to create the appropriate representation of the data to be served to a web browser. While this gives maximum flexibility to the developer it is also a drag, requiring you to write a lot of repetitive code.

Joomla! 1.6 and later is providing a solution to this problem, at least for edit views: JForm. With it it's possible to create an XML file which defines the controls of the form and have JForm render it as HTML.

Pros:

- The view templates are easier to read
- The HTML generation is abstracted, making it easier to upgrade to newer versions of Joomla! using a different HTML structure

Cons:

- You need to change your Controllers, Models and Views to cater for and display the forms
- They only apply to edit views

FOF takes this concept further with the FOFForm package. Not only can you create edit views, but you can also create browse (records listing) and read (single record display) views out of XML forms. Moreover, the forms are handled automatically by the FOF base MVC classes without requiring you to write any additional code. If you want you can always combine a traditional .php view template with a form file for maximum customisation of your view.

2.1. Form types

2.1.1. The different form types

As implied above, there are three types of XML forms available in FOF: Browse, Read and Edit. Each one follows slightly different conventions and is used in different tasks of each MVC triad. In this section we are going to present what each of those types does and what is its structure.

There a few things you should know before we go into more details.

All form files are placed in your view's `tmpl` directory, e.g. `components/com_example/views/items/tmpl`.

All form files' names begin with `form.` and end with `.xml`. This is required for Joomla! to distinguish them from view metadata XML files. The middle part of their name follows the same convention as the regular view template files, i.e. "default" for browse tasks, "form" for edit tasks and "item" for read tasks.

For example, the browse form for `com_example`'s items view is located in `components/com_example/views/items/tmpl/form.default.xml` whereas the form for editing a single item is located in `components/com_example/views/item/tmpl/form.form.xml`

2.1.2. Browse forms

Browse forms are used to create a records list view. They are typically used in the back-end to allow the user to view and manipulate a list of records. A typical browse form looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<form
    lessfiles="media://com_todo/css/backend.less||media://com_todo/css/backend.css"
    type="browse"
    show_header="1"
    show_filters="1"
    show_pagination="1"
```

```
    norows_placeholder="COM_TODO_COMMON_NORECORDS"
>
    <headerset>
        <header name="ordering" type="ordering" sortable="true" tdwidth="1%" />
        <header name="todo_item_id" type="rowselect" tdwidth="20" />
        <header name="title" type="fieldsearchable" sortable="true"
            buttons="yes" buttonclass="btn"
        />
        <header name="due" type="field" sortable="true" tdwidth="12%" />
        <header name="enabled" type="published" sortable="true" tdwidth="8%" />
    </headerset>

    <fieldset name="items">
        <field name="ordering" type="ordering" labelclass="order"/>

        <field name="todo_item_id" type="selectrow"/>

        <field name="title" type="text"
            show_link="true"
            url="index.php?option=com_todo&amp;view=item&amp;id=[ITEM:ID]"
            class="todoitem"
            empty_replacement="(no title)"
         />

        <field name="due" type="duedate" />

        <field name="enabled" type="published"/>
    </fieldset>
</form>
```

You MUST have exactly one `<headerset>` and one `<fieldset>` tag. The `name` attribute of the `<fieldset>` MUST always be `items`. Extra tags and/or `<fieldset>` tags with different name attributes (or no name attributes) will be ignored.

2.1.2.1. Form attributes

The enclosing `<form>` tag MUST have the following attributes:

type It must be always set to `browse` for FOF to recognise this as a Browse form

The enclosing <form> tag MAY have one or more of the following attributes:

lessfiles FOF allows you to include LESS files to customise the styling of your components. You can give a comma separated list of LESS files' identifiers (see the "Media files identifiers" section below) to be loaded by FOF. For example `media://com_example/less/backend.less`

Compiled LESS files are cached in the `media/lib_fof/compiled directory` for efficiency reasons, using a mangled filename for privacy/security reasons. They are not written in your site's cache or adminstrator/cache directory as these directories are not supposed to be web-accessible, whereas the compiled CSS files, by definition, need to be web-accessible.

Since LESS files require a lot of memory and time to compile you can also provide an alternative pre-compiled CSS file, separated from your LESS file with two bars. For example: `media://com_example/less/backend.less||media://com_example/css/backend.css`

cssfiles — This works in the same manner as the lessfiles directive, but you are only supposed to specify standard CSS files. The CSS files are defined using identifiers, too. `For example: media://com_example/css/backend.css`

Please note that media file overrides rules are in effect for these CSS files.

jsfiles — Works the same way as cssfiles, but it's used to load Javascript files. The Javascript files are defined using identifiers, too. For example: `media://com_example/js/backend.js`

Please note that media file overrides rules are in effect for these Javascript files.

show_header — Should we display the header section of the browse form? This is the place where the field titles are displayed.

show_filters — Should we show the filter section of the browse form? On Joomla! 2.5 this is the area below the header where the user can filter the display based on his own criteria. On Joomla! 3.0 and later this area is rendered in the sidebar, at the left hand side of the records list.

show_pagination — Should we show the pagination results? That's the links to the first, second, third, …, last page and the drop-down for the number of items per page. It is displayed below the list of records.

norows_placeholder — A translation key displayed instead of a records list when the current view contains no records, e.g. the table is empty or the filters limit display to zero records.

2.1.3. Read forms

While browse views display a list of records, read forms will display just a single record. These are nowhere near as powerful as hand-coded PHP-based view templates but can be used to get a quick single item output in a snatch when prototyping a component or when your data is really simple. A typical read form looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<form
    lessfiles="media://com_todo/css/frontend.less||media://com_todo/css/frontend.css"
    type="read"
>
    <fieldset name="a_single_item" class="todo-item-container form-horizontal">
        <field name="title" type="text"
            label=""
            class="todo-title-field"
            size="50"
         />

        <field name="due" type="duedate"
            label="COM_TODO_ITEMS_FIELD_DUE"
            labelclass="todo-field"
            size="20"
            default="NOW"
         />
        <field name="description" type="editor"
            label=""
        />
    </fieldset>
</form>
```

You MUST have at least one `<fieldset>` tag. The name attribute of the `<fieldset>` is indifferent.

2.1.3.1. Form attributes

The enclosing `<form>` tag MUST have the following attributes:

type It must be always set to `read` for FOF to recognise this as a Read form

The enclosing <form> tag MAY have one or more of the following attributes:

lessfiles FOF allows you to include LESS files to customise the styling of your components. You can give a comma separated list of LESS files' identifiers (see the "Media files identifiers" section below) to be loaded by FOF. For example: `media://com_example/less/backend.less`

Compiled LESS files are cached in the `media/lib_fof/compiled` directory for efficiency reasons, using a mangled filename for privacy/security reasons. They are not written in your site's `cache` or `adminstrator/cache` directory as these directories are not supposed to be web-accessible, whereas the compiled CSS files, by definition, need to be web-accessible.

Since LESS files require a lot of memory and time to compile you can also provide an alternative pre-compiled CSS file, separated from your LESS file with two bars. For example: `media://com_example/less/backend.less||media://com_example/css/backend.css`

cssfiles This works in the same manner as the `lessfiles` directive, but you are only supposed to specify standard CSS files. The CSS files are defined using identifiers, too. For example: `media://com_example/css/backend.css`

Please note that media file overrides rules are in effect for these CSS files.

jsfiles Works the same way as `cssfiles`, but it's used to load Javascript files. The Javascript files are defined using identifiers, too. For example: `media://com_example/js/backend.js`

Please note that media file overrides rules are in effect for these Javascript files.

2.1.4. Edit forms

Edit forms are used to edit a single record. They are typically used in the back-end. If you want to use an Edit form in the front-end you will need to specialise your Toolbar class to render a front-end toolbar in the edit task of this specific view, otherwise the form will not be able to be submitted (unless you do other tricks, outside the scope of this developers' documentation).

An edit form looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<form
    lessfiles="media://com_todo/css/backend.less||media://com_todo/css/backend.css"
    validate="true"
>
    <fieldset name="basic_configuration"
        label="COM_TODO_ITEMS_GROUP_BASIC"
        description="COM_TODO_ITEMS_GROUP_BASIC_DESC"
        class="span6"
    >
        <field name="title" type="text"
            class="inputbox"
            label="COM_TODO_ITEMS_FIELD_TITLE"
            labelclass="todo-label todo-label-main"
            required="true"
```

```
            size="50"
         />

        <field name="due" type="calendar"
            class="inputbox"
            label="COM_TODO_ITEMS_FIELD_DUE"
            labelclass="todo-label"
            required="true"
            size="20"
            default="NOW"
         />
        <field name="enabled" type="list" label="JSTATUS"
            labelclass="todo-label"
            description="JFIELD_PUBLISHED_DESC" class="inputbox"
            filter="intval" size="1" default="1"
        >
            <option value="1">JPUBLISHED</option>
            <option value="0">JUNPUBLISHED</option>
        </field>
    </fieldset>
    <fieldset name="description_group"
        label="COM_TODO_ITEMS_GROUP_DESCRIPTION"
        description="COM_TODO_ITEMS_GROUP_DESCRIPTION_DESC"
        class="span6"
    >
        <field name="description" type="editor"
            label=""
            class="inputbox"
            required="false"
            filter="JComponentHelper::filterText" buttons="true"
        />
    </fieldset>
</form>
```

2.1.4.1. Form attributes

The enclosing `<form>` tag MAY have one or more of the following attributes:

lessfiles
: FOF allows you to include LESS files to customise the styling of your components. You can give a comma separated list of LESS files' identifiers (see the "Media files identifiers" section below) to be loaded by FOF. For example: `media://com_example/less/backend.less`

  Compiled LESS files are cached in the `media/lib_fof/compiled` directory for efficiency reasons, using a mangled filename for privacy/security reasons. They are not written in your site's `cache` or `adminstrator/cache` directory as these directories are not supposed to be web-accessible, whereas the compiled CSS files, by definition, need to be web-accessible.

  Since LESS files require a lot of memory and time to compile you can also provide an alternative pre-compiled CSS file, separated from your LESS file with two bars. For example: `media://com_example/less/backend.less||media://com_example/css/backend.css`

cssfiles
: This works in the same manner as the `lessfiles` directive, but you are only supposed to specify standard CSS files. The CSS files are defined using identifiers, too. For example: `media://com_example/css/backend.css`

Please note that media file overrides rules are in effect for these CSS files.

jsfiles — Works the same way as `cssfiles`, but it's used to load Javascript files. The Javascript files are defined using identifiers, too. For example: `media://com_example/js/backend.js`

Please note that media file overrides rules are in effect for these Javascript files.

validation — Set it to `true` to have Joomla! load its unobtrusive Javascript validation script. Please note that FOF does not perform automatic server-side validation checks. This is the responsibility of your specialised Table class and its check() method.

2.1.5. Formatting your forms

OK, granted, the automatically rendered forms are a timesaver but, by default, they look terrible. This is quite expected. It's like comparing a rug churned out by a mechanised production line (the automatically rendered form) and a hand-stiched persian rug (the hand-coded PHP-based view template). The good news is that, unlike rugs, there's some room of improvement with XML forms.

For starters, the `<fieldset>`s of Edit and Read forms, as well as the fields themselves, can be assigned CSS classes and IDs which can help you provide a custom style. Moreover, you can mix XML forms and PHP-based view templates to further customise the display of your forms.

In this section we will cover both customisation methods. If this doesn't sound enough for your project you can always use hand-coded PHP-based view templates, much like how you did since Joomla! 1.5.0. It's up to you to decide which method is best for your project!

2.1.5.1. Assigning classes and IDs to <fieldset>s

Each fieldset of a Read and Edit form can have the following optional attributes:

class — One or more CSS classes to be applied to the generated `<div>` element.

name — The value of this attribute is applied to the id attribute of the generated `<div>` element.

label — The value of this attribute is rendered as a level 3 heading (`<h3>`) element at the top of the generated `<div>` element.

If you are using Joomla! 3 (which has Bootstrap by default) or Joomla! 2.5 together with the optional Akeeba Strapper package (which back ports Bootstrap to Joomla! 2.5) you can use Bootstrap's classes to create visually interesting interfaces. For example, using `class="span6 pull-left"` will create a half-page-wide left floating sidebar out of your field set.

2.1.5.2. Mixing XML forms with PHP-based view templates

Inside your .php view template file you can use `$this->getRenderedForm()` to return the XML form file rendered as HTML. This allows you to customise the layout (e.g. adding information before/after the form) while still using the XML file to render the actual form.

To use this approach, simply insert this code in your custom .php template file:

```
<?php
$viewTemplate = $this->getRenderedForm();
echo $viewTemplate;
?>
```

2.2. Header fields type reference

2.2.1. How header fields work

A header field has two distinct functions:

- It is used to render headers in list views which are used to label the columns of the display and optionally allow you to sort the table by a specific field
- It is used to render filtering widgets (drop-down lists and search boxes). In Joomla! 2.5 you can only render filtering widgets *directly below* a header field in a list table and you can only have up to as many filtering widgets as your fields. In Joomla! 3.x and above the filtering widgets are rendered either *above* the header fields (search boxes) or in the left-hand column (drop-down lists). As in Joomla! 3.x and above the filters are detached from the header fields you can have as many filters as you want, even more than the number of fields you are displaying in the filter list.

A leader field can render only a header, only a filter or both. Most of the header field types render both. Those whose name starts with `filter` will only render a filtering widget, but not a header field. As a result these header fields will only work on Joomla! 3.x and later.

2.2.2. Common fields for all types

For all following fields you can set the following attributes:

- **name** The name of the header field. This has to match the table field name in the model.

  If you want to create a header for a calculated field or for a column that doesn't correspond to a table field please use a name that doesn't overlap with the name of a column in the table. If you want to list a field many times (e.g. display a row selection checkbox and the record ID at the same time) you will have to use the same `name` in both headers, but use a different `id` attribute.

- **type** The header type. See below for the available field types, as well as the options which can be specified in each one of them.
- **label** The language string which will be used for the label of the header; this is a language string that will be fed to JText::_() for translation.
- **id** The `id` attribute for this header. Skip it to have FOF create one based on the field name.

  If none is provided FOF will automatically create one using the convention component_modelname_fieldname_LABEL where component is the name of your component, modelname is the name of your model (usually equals to the view name) and fieldname is the name of the field. For example, for a component com_foobar, a view named items and a field named baz we get the language string COM_FOOBAR_ITEMS_BAZ_LABEL.

- **tdwidth** The width of this column in the list table. You can use percentile or pixel units, i.e. `tdwidth="10%"` or `tdwidth="120px"`
- **sortable** Set to "true" if you want to be able to sort the table by this field.
- **filterclass** The CSS class for the filtering widget
- **onchange** The Javascript code to be executed when the filtering widget's value is modified

2.2.2.1. Additional attributes for search box filtering widgets

The following attributes apply to all header fields rendering a search box filtering widget:

- **searchfieldname** The name of the field that will be searchable. If omitted it will be the same as the `name` attribute.
- **placeholder** The placeholder text when the field is empty. Useful to explain what kind of information this search field is supposed to be searching in.
- **size** The size (in characters) of the search box
- **maxlength** The maximum length in characters which is allowed to be entered in the field
- **buttons** Set to true (or skip) to show Go and Reset buttons next to the text field. Set to "false" to hide those buttons. The user can still press Enter to submit the form.
- **buttonclass** The CSS class of the Go and Reset buttons

2.2.2.2. Additional attributes for drop-down list filtering widgets

This element has `<option>` sub-elements defining the available options. Please consult Joomla!'s own `list` field type for more information.

Since FOF 2.1.0 we allow you to use a programmatically generated data source instead of the hard-coded `<option>` tags. This can be used when you need your code to generate options based on some configuration data, data from the database and so on. You do that by supplying the name of a PHP class and a static method on that class which returns the data. The data must be returned in an indexed array where the key is the key of the drop-down list item and the value is the description (translation key or string). You may also use a simple array containing indexed arrays by using the `source_key` and `source_value` attributes.

The following additional attributes apply to all header fields rendering a drop-down list filtering widget.

- **source_file** (optional) The PHP file which provides the class and method. It is given in the pseudo-URL format e.g. `admin://components/com_foobar/helpers/mydata.php` or `site://components/com_foobar/helpers/mydata.php` for a file relative to the administrator or site root directory respectively.
- **source_class** (required) The name of the PHP class to use, e.g. `FoobarHelperMydata`
- **source_method** (required) The static method of the PHP class to use, e.g. `getSomeFoobarData`
- **source_key** (optional) If you are using an array of indexed arrays, this is the key of the indexed array that contains the key of the drop-down option.
- **source_key** (optional) If you are using an array of indexed arrays, this is the key of the indexed array that contains the value (description) of the drop-down option.
- **source_translate** (optional) By default all values are being translated, i.e. fed through JText::_(). If you don't want that, set this attribute to "false".

2.2.3. Field Types

2.2.3.1. accesslevel

Displays a header field with a viewing access level drop-down filtering widget.

There are no additional attributes to set.

2.2.3.2. field

Displays a header field, without any filtering widget.

There are no additional attributes to set.

2.2.3.3. fielddate

Displays a header field with a date selection search box filtering widget.

The additional attributes you can set are:

- **readonly** Set to true to make the search box read only
- **disabled** Set to true to disable the search box (it displays but you can't click on it)
- **filter** Skip to show the date/time as entered. Set to SERVER_UTC to convert a date to UTC based on the server timezone. Set to USER_UTC to convert a date to UTC based on the user timezone.

2.2.3.4. fieldsearchable

Displays a header field with a search box filtering widget.

There are no additional attributes to set.

2.2.3.5. fieldselectable

Displays a header field with a drop-down list filtering widget.

There are no additional attributes to set.

2.2.3.6. fieldsql

Displays a header field with a drop-down list filtering widget. The source of the filter values comes from an SQL query.

The additional attributes are:

- **key_field** the table field to use as key
- **value_field** the table field to display as text
- **query** the actual SQL query to run

We recommend avoiding this field type as the query is specific to a particular database server technology. Using the `model` or `fieldselectable` type with a programmatic data source is strongly encouraged.

2.2.3.7. filtersearchable

This is the same as fieldsearchable but no header is rendered. Only the filtering widget is rendered. This header field type only works on Joomla! 3.x and later.

2.2.3.8. filterselectable

This is the same as fieldselectable but no header is rendered. Only the filtering widget is rendered. This header field type only works on Joomla! 3.x and later.

2.2.3.9. filtersql

This is the same as fieldsql but no header is rendered. Only the filtering widget is rendered. This header field type only works on Joomla! 3.x and later.

The same warning applies to using this field type.

2.2.3.10. language

Displays a header field with a drop-down list containing the languages installed on your site.

The additional attributes are:

- **client** If set to "site" displays a list of installed front-end languages. If set to "administrator" displays a list of installed back-end languages. Default: site.

2.2.3.11. model

Similar to the fieldselectable header, but gets the options from a FOFModel descendant.

You can set the following attributes on top of those of the 'fieldselectable' field type's:

- **model** The name of the model to use, e.g. FoobarModelItems
- **key_field** The name of the field in the model's results which is used as the key value of the drop-down
- **value_field** The name of the field in the model's results which is used as the label of the drop-down
- **translate** Should the value field's value be passed through JText::_() before being displayed?
- **apply_access** Should we respect the view access level, if an access field is present in the model
- **none** The placeholder to be shown if the value is not found in the data returned by the model. This placeholder goes through JText, so you can use a language string if you like.

In order to filter the model you can specify `<state>` sub-elements in the format:

```
<state key="state_key">value</state>
```

Where state_key is the key of a state variable and value is its value. For instance, you could have something like:

```
<state key="foobar_category_id">123</state>
```

2.2.3.12. ordering

Displays a header field which allows reordering of your data.

On Joomla! 2.5 it displays the name of the field followed by a disk icon which saves the ordering.

On Joomla! 3.x and later it displays an "up and down triangle" icon. When clicked the AJAX-powered reordering handles in the list view become enabled.

There are no additional attributes.

2.2.3.13. published

Displays a header field and a drop-down filtering field for Published / Unpublished and related publishing options.

The additional attributes are:

- **show_published** Should we show the Published status in the filter? Default: true
- **show_unpublished** Should we show the Unpublished status in the filter? Default: true

- **show_archived** Should we show the Archived status in the filter? Default: false
- **show_trash** Should we show the Trashed status in the filter? Default: false
- **show_all** Should we show the All status in the filter? Default: false. You actually don't need this as no selection results in all records, irrespective of their publish state, to be displayed.

2.2.3.14. rowselect

Displays a checkbox which, when clicked, automatically selects all the row selection checkboxes in the list.

There are no additional attributes.

2.3. Form fields type reference

2.3.1. Common fields for all types

For all following fields you can set the following attributes:

- **name** The name of the field. This has to match the table field name in the model.

  If you want to create a header for a calculated field or for a column that doesn't correspond to a table field please use a name that doesn't overlap with the name of a column in the table. If you want to list a field many times (e.g. display a row selection checkbox and the record ID at the same time) you will have to use the same `name` in both fields, but use a different `id` attribute.

- **type** The field type. See below for the available field types, as well as the options which can be specified in each one of them.
- **label** The language string which will be used for the label of the field; this is a language string that will be fed to JText::_() for translation. If you leave it empty FOF will automatically generate a language string using the convention *`COMPONENTNAME_VIEWNAME_FIELDNAME`*_LABEL.
- **id** The `id` attribute for this field. Skip it to have FOF create one based on the field name.

  If none is provided FOF will automatically create one using the convention component_modelname_fieldname_LABEL where component is the name of your component, modelname is the name of your model (usually equals to the view name) and fieldname is the name of the field. For example, for a component com_foobar, a view named items and a field named baz we get the language string COM_FOOBAR_ITEMS_BAZ_LABEL.

- **emptylabel** Set this to 1 if you intend to have a field without a label. In this case you must NOT define the `label` attribute.
- **description** The language string which will be used for the label of the field; this is a language string that will be fed to JText::_() for translation.
- **required** Set it to 1, yes or true to make this a required field. If you use the form validation then the form cannot be submitted unless this value is filled in.

Important

The automatic label and description only apply if you are using Akeeba Strapper or if you are using Joomla! 3.0 and later. If you are using FOF on plain old Joomla! 2.5 you must provide the `label` and `description` attributes manually.

2.3.2. Field types

2.3.2.1. accesslevel

This will display a select list with existing Joomla! Access Levels.

You can set the following attributes:

- **class** CSS class (default '')

2.3.2.2. button

This will display an input button.

You can set the following attributes:

- **class** CSS class (default '')
- **icon** Bootstrap icon to add to the button (default '')
- **onclick** "onclick" attribute to add to the button (default '')
- **text** Button text value; this is a language string that will be fed to JText::_() for translation

2.3.2.3. cachehandler

This will display a select list with available Joomla! cache handlers

You can set the following attributes:

- **class** CSS class (default '')

2.3.2.4. calendar

This will display a calendar/date field.

You can set the following attributes:

- **class** CSS class (default '')
- **format** (defaults '%Y-%m-%d')
- **filter** can be one the following:
  - SERVER_UTC convert a date to UTC based on the server timezone
  - USER_UTC convert a date to UTC based on the user timezone

2.3.2.5. captcha

This will display a captcha input.

You can set the following attributes:

- **plugin** The name of the CAPTCHA plugin to use. Leave empty to use whatever is the default on in the Global Configuration of the Joomla! site

2.3.2.6. checkbox

This will display a single checkbox input.

You can set the following attributes:

- **class** CSS class (default '')
- **value** the input value
- **checked** the default status for input
- **disabled** Is this a disabled form element?

2.3.2.7. components

This will display a select with a list of installed Joomla! components

You can set the following attributes:

- **class** CSS class (default '')
- **client_ids** comma separated list of applicable client ids (note: 0 = admin, 1 = site)
- **readonly** is this a read only field?
- **disabled** Is this a disabled form element?
- **multiple** Should we allow multiple selections?
- **onchange** onchange JavaScript event

2.3.2.8. editor

This will display a WYSIWYG edit area field for content creation and formatted HTML display.

You can set the following attributes:

- **class** CSS class (default '')
- **rows** How many rows the generated `<textarea>` will have, typically used when Javascript is disabled on the browser
- **cols** How many columns the generated `<textarea>` will have, typically used when Javascript is disabled on the browser
- **height** The height of the editor (default: 250)
- **width** The width of the editor (default: 100%)
- **asset_field** The name of the asset_id field in the form (default: asset_id)
- **created_by_field** The name of the created_by field in the form
- **asset_id** The Joomla! asset ID for this record. Leave empty to let FOF use the value of the asset field defined by asset_field.

- **buttons** Which buttons should we show (rendered by editor-xtd plugins)? Use 0, false or no to show now buttons, otherwise provide a comma separated list of button plugin names
- **hide** Which buttons should we hide? Similar to above.

2.3.2.9. email

This will display a text input which expects a valid e-mail address.

You can set the following attributes:

- **class** CSS class (default '')
- **show_link** if true put a mailto: link around the address (default false)
- **size** Size of the text input in characters
- **maxlength** Maximum length of the input in characters
- **readonly** Is this a read only field?
- **disabled** Is this a disabled form element?
- **onchange** onchange Javascript event

2.3.2.10. groupedlist

This will display a grouped drop down list.

You can set the following attributes:

- **class** CSS class (default '')

This element supports sub-elements organised in `<group>` and `<option>` tags. For more information please consult the documentation of Joomla!'s JFormFieldGroupedList element.

2.3.2.11. hidden

This will display a hidden input.

You can set the common attributes. Moreover, to make sure this field is rendered properly, you MUST define the attribute `emptylabel="1"` and NOT assign a label attribute at all.

2.3.2.12. image

This is an alias for the "media" field type (see below).

2.3.2.13. imagelist

This will display a media selection field showing images from a specified folder.

You can set the following attributes:

- **class** CSS class
- **directory** folder to search the images in
- **style** inline style

- **width** HTML width attribute
- **height** HTML height attribute
- **align** HTML align attribute
- **rel** HTML rel attribute
- **title** image title
- **filter** The filtering string for filenames to show. Default: `\.png$|\.gif$|\.jpg$|\.bmp$|\.ico$|\.jpeg$|\.psd$|\.eps$`

2.3.2.14. integer

This will display a text input which expects a valid integer value.

You can set the following attributes:

- **class** CSS class (default '')
- **first** Starting number
- **last** Last number to show
- **step** Step for increasing the numbers

For example, when using first=10, last=20 and step=2 you get a list of 10, 12, 14, 16, 18, 20.

2.3.2.15. language

This will display a select input of all available Joomla! languages

You can set the following attributes:

- **class** CSS class (default '')
- **client** Can take the values of 'site' or 'administrator' to show the available languages for the front- and back-end respectively.

2.3.2.16. list

This will display a select input of generic options.

IMPORTANT The following attributes apply to all field types that present a drop-down list; they all descend from this field type.

You can set the following attributes:

- **class** CSS class (default '')
- **readonly** Is this a read-only field?
- **disabled** Is this a disabled form element?
- **multiple** Should we allow multiple selections?
- **onchange** The onChange Javascript event

- **url** URL template for each element (use [ITEM:ID] as a placeholder for the item id)
- **show_link** if true, adds a link around each item based on the "url" attribute (default false)

This element has `<option>` sub-elements defining the available options. Please consult Joomla!'s own element of the same type for more information.

Since FOF 2.1.0 we allow you to use a programmatically generated data source instead of the hard-coded `<option>` tags. This can be used when you need your code to generate options based on some configuration data, data from the database and so on. You do that by supplying the name of a PHP class and a static method on that class which returns the data. The data must be returned in an indexed array where the key is the key of the drop-down list item and the value is the description (translation key or string). You may also use a simple array containing indexed arrays by using the `source_key` and `source_value` attributes.

The relevant attributes are:

- **source_file** (optional) The PHP file which provides the class and method. It is given in the pseudo-URL format e.g. `admin://components/com_foobar/helpers/mydata.php` or `site://components/com_foobar/helpers/mydata.php` for a file relative to the administrator or site root directory respectively.
- **source_class** (required) The name of the PHP class to use, e.g. `FoobarHelperMydata`
- **source_method** (required) The static method of the PHP class to use, e.g. `getSomeFoobarData`
- **source_key** (optional) If you are using an array of indexed arrays, this is the key of the indexed array that contains the key of the drop-down option.
- **source_key** (optional) If you are using an array of indexed arrays, this is the key of the indexed array that contains the value (description) of the drop-down option.
- **source_translate** (optional) By default all values are being translated, i.e. fed through JText::_(). If you don't want that, set this attribute to "false".

2.3.2.17. media

This will display a media selection field.

You can set the following attributes:

- **class** CSS class
- **style** inline style
- **width** HTML width attribute
- **height** HTML height attribute
- **align** HTML align attribute
- **rel** HTML rel attribute
- **title** image title
- **asset_field** The name of the asset_id field in the form (default: asset_id)
- **created_by_field** The name of the created_by field in the form

- **asset_id** The Joomla! asset ID for this record. Leave empty to let FOF use the value of the asset field defined by asset_field.
- **link** The link to a media management component to use. Skip this to use Joomla!'s own com_media (strongly recommended!)
- **size** Field size in characters
- **onchange** The onChange Javascript event
- **preview** Should we show a preview of the selected media file?
- **preview_width** Maximum width of preview in pixels
- **preview_height** Maximum height of preview in pixels
- **directory** Directory to scan for images relative to site's root. Skip to use the site's images directory.

2.3.2.18. model

Similar to the list field, but gets the options from a FOFModel descendant.

You can set the following attributes on top of those of the 'list' field type's:

- **model** The name of the model to use, e.g. FoobarModelItems
- **key_field** The name of the field in the model's results which is used as the key value of the drop-down
- **value_field** The name of the field in the model's results which is used as the label of the drop-down
- **translate** Should the value field's value be passed through JText::_() before being displayed?
- **apply_access** Should we respect the view access level, if an access field is present in the model
- **none** The placeholder to be shown if the value is not found in the data returned by the model. This placeholder goes through JText, so you can use a language string if you like.
- **format** See the text field type
- **show_link** See the text field type
- **url** See the text field type

In order to filter the model you can specify `<state>` sub-elements in the format:

```
<state key="state_key">value</state>
```

Where state_key is the key of a state variable and value is its value. For instance, you could have something like:

```
<state key="foobar_category_id">123</state>
```

2.3.2.19. ordering

This will display an ordering field for your list, both in traditional Joomla! method and with a new ajax drag'n'drop method. We recommend placing this field first on your form, to respect Joomla! 3.0 and later's JUI (Joomla! User Interface) guidelines.

2.3.2.20. password

This will display a password input field.

You can set the following attributes:

- **class** CSS class
- **size** Size of the field in characters
- **maxlength** Maximum length of the input in characters
- **autocomplete** Should we allow browser autocomplete of the password field?
- **readonly** Is this a read only field?
- **disabled** Is this a disabled form element?
- **strengthmeter** Should we show a password strength meter?
- **threshold** What is the minimum password strength we are supposed to accept in order to validate the field (default: 66)?

2.3.2.21. plugins

This will display a select input with a list of all installed Joomla! package.

You can set the following attributes:

- **class** CSS class
- **folder** The plugin type to load, e.g. "system", "content" and so on.

The list field type's attributes apply as well.

2.3.2.22. published

This will display a status toggle input field (each time you click on it it changes the status).

You can set the following attributes:

- **show_published** if true, the "published" status will be included in the toggle cycle (default true)
- **show_unpublished** if true, the "unpublished" status will be included in the toggle cycle (default true)
- **show_archived** if true, the "archived" status will be included in the toggle cycle (default false)
- **show_trash** if true, the "trash" status will be included in the toggle cycle (default false)
- **show_all** if true, all the available status will be included in the toggle cycle (default false)

The list field type's attributes apply as well.

2.3.2.23. radio

This will display a radio selection input.

You can set the following attributes:

- **class** CSS class

2.3.2.24. rules

Displays the ACL privileges setup user interface.

Please consult the documentation of JFormFieldRules for more information.

2.3.2.25. selectrow

Displays a checkbox to select the entire row for toolbar button operations such as edit, delete, copy etc.

2.3.2.26. sessionhandler

This will display a Joomla! session handler selection input.

You can set the following attributes:

- **class** CSS class

Please refer to Joomla!'s JFormFieldSessionHandler for more information.

2.3.2.27. spacer

This will display a spacer (static element) between form elements.

You can set no attributes.

2.3.2.28. sql

This will display a select input based on a custom SQL query

You can set the following attributes:

- **class** CSS class
- **key_field** the table field to use as key
- **value_field** the table field to display as text
- **query** the actual SQL query to run

We recommend avoiding this field type as the query is specific to a particular database server technology. Using the `model` or `list` type with a programmatic data source is strongly encouraged.

2.3.2.29. tel

This will display a text input which expects a valid telephone value.

You can set the following attributes:

- **class** CSS class (default '')
- **show_link** if true, a "tel:" link will be appended around the field value (default false)
- **empty_replacement** a string to show in place of the field when it's empty

The text field type's attributes apply as well.

2.3.2.30. text

This will display a single line text input.

You can set the following attributes:

- **class** CSS class (default '')
- **url** URL template for each element (use [ITEM:ID] as a placeholder for the item id). This goes through the field tag replacement (see below)
- **show_link** if true, a "tel:" link will be appended around the field value (default false)
- **empty_replacement** a string to show in place of the field when it's empty
- **size** The size of the input in characters
- **maxlength** The maximum acceptable input length in characters
- **readonly** Is this a read-only field?
- **disabled** Is this a disabled form field element?
- **format_string** A string or translation key used to format the text data before it is displayed. Uses the format() PHP function's syntax.
- **format_if_not_empty** Should we apply the format string even when the field is empty? Default: true
- **parse_value** If set to true, the value of the field will go through the field tag replacement (see below) Default: false

2.3.2.30.1. Field tag replacement for text fields

You can reference values from other fields inside your text. You can do that using the square bracket tag syntax, i.e. `[ITEM:fieldname]` is replaced with the value of the field `fieldname`. The tag must open with a square bracket, followed by the uppercase word ITEM, followed by a colon, the field name and closing with a square bracket. You must not use spaces in the tag.

FOF also recognises the special tag `[ITEM:ID]`, replacing it with the value of the key field of the table.

2.3.2.31. textarea

This will display a textarea input.

You can set the following attributes:

- **class** CSS class (default '')
- **disabled** Is this disabled form element?
- **cols** Number of columns
- **rows** Number of rows
- **onchange** The onChange Javascript event

2.3.2.32. title

This is like a text field. On list views it will display a second line containing secondary information, e.g. the alias (slug) of the record.

The following attributes are used on top of the text field's attributes:

- **slug_field** The name of the field containing the slug or other secondary information to display. Default: slug
- **slug_format** The format string (string or translation key) for the secondary information line. Default: (%s)
- **slug_class** The CSS class of the secondary information line. Default: small

2.3.2.33. timezone

This will display a select list with all available timezones.

You can set the following attributes:

- **class** CSS class (default '')

2.3.2.34. url

This will display a text input which expects a valid URL.

You can set the following attributes:

- **class** CSS class (default '')
- **show_link** if true, an <a> link will be added around the field value (default false)
- **empty_replacement** a string to show in place of the field when it's empty

The text field type's attributes apply as well.

2.3.2.35. user

This will display a select list with all available Joomla! users.

You can set the following attributes:

- **class** CSS class (default '')
- **show_username** if true, show the username (default true)
- **show_email** if true, show the username (default true)
- **show_name** if true, show the full name (default true)
- **show_id** if true, show the id (default true)
- **show_link** if true, add a link around the field value (default false)
- **show_avatar** if true, show the avatar (user picture). Default false.
- **avatar_size** size of the image in the avatar (avatars are square, so this is both the width and height of the avatar)
- **avatar_method** if set to "plugin" use FOF plugins, else fall back to a Gravatar based on the user's email address

Appendix A. Definitions

1. Media file identifiers

FOF expects you to give an abstracted path to your media (CSS, Javascript, image, …) files, also called an "identifier". It allows it to perform media file overrides very easily, in a fashion similar to how Joomla! performs template overrides for view files. This section will help you understand how they are used and how media file overrides work.

Media file identifiers are in the form:

```
area://path
```

Where the `area` can be one of:

media : The file is searched inside your site's `media` directory. FOF will also try to locate it in the media overrides directory of your site, e.g. `templates/your_template/media` where your_template is the name of the currently active template on your site.

In this case the `path` is the rest of the path relative to the media or media override directory. The first part of your path SHOULD be your extension's name, e.g. com_example.

Example: `media://com_example/css/style.css` will look for the file `templates/your_template/media/com_example/css/style.css` or, if it doesn't exist, `media/com_example/css/style.css`

admin : The file is searched for in the administration section of your extension. The first part of the path MUST be your extension's name. The file is first searched for in your template override directory.

Example: `admin://com_example/assets/style.css` will look for the file `administrator/templates/your_template/com_example/assets/style.css` or, if it doesn't exist, `administrator/components/com_example/assets/style.css`

site : The file is searched for in the front-end section of your extension. The first part of the path MUST be your extension's name. The file is first searched for in your template override directory.

Example: `site://com_example/assets/style.css` will look for the file `templates/your_template/com_example/assets/style.css` or, if it doesn't exist, `components/com_example/assets/style.css`

Important

FOF cannot know what is the other side's template. Let's put it simply. If you are in the front-end, your template is called "foobar123" and you use the identifier `admin://com_example/assets/style.css`, FOF will look for the template override in `administrator/templates/foobar123/com_example/assets/style.css`. Of course this is incorrect, but there is no viable way to know what the back-end template in use is from the site's front-end and vice versa. As a result, we strongly recommend only using `media://` identifiers for media files.

On top of that there is a security aspect as well. The front-end of your component should never try to load media files from the back-end of the component. Many web masters choose to conceal the fact that they are using Joomla! by means of password protection or redirection of the `administrator` directory.