# Getting Started With FIM

Benjamin Desef

## Contents

# 1 Introduction

This manual will give you an idea of how **F**ramework **Im**proved works. Therefore, a simple application will be built step-by-step. The idea of this application comes from Zend Framework's "Skeleton Application"; this document will also be oriented at Zend's "Getting Started With Zend Studio 10 & Zend Server 6". You should already know about *why* to use a Framework and be familiar with Smarty template syntax.
The third chapter contains a reference that lists every public function grouped by the file they occur in.

# 2 The demo application

We will shortly explain the directory structure that our final demo application will have. The top directory that contains all files that you will create while working through this document is called *CodeDir*. Everything will take place in this directory. Table 1 shows the top-level directory structure of CodeDir. This table is intended for a quick overview only. You do not need every directory in every installation of FIM. This guide will explain step-by-step what is important when creating an application.

## 2.1 Enabling FIM

FIM does not need many files in order to function properly.
Create an empty directory under your DocumentRoot if you do not wish to work directly in the main folder of your web server.
You will then need to provide FIM's program files. It is possible to extract these to a folder in your include path, but you may also create a new folder named `fim` and extract the files at this location.
To enable FIM, you will need to create two files (assuming you are using Apache HTTPD): one controller file and one htaccess configuration file for the server.

**Listing 1** – The controller file: `index.php`

```php
<?php
   require 'fim/fim.php';
   fimInitialize();
```

Listing 1 shows the content of the controller. At the beginning, this file will contain no more than three lines. But you may—and later on will—add some configuration parameters to the initialization function call. Note that you may have to adjust the path to `fim.php`, which lies in FIM's program folder.

| File/Directory | Description |
|---|---|
| `.htaccess` | Web server configuration file. Instructs Apache HTTPD to redirect all requests to the controller. |
| `index.php` | Controller. Starts FIM, contains configuration |
| `cache/` | Automatically-created directory that contains several files that are generated by FIM. Do not change the contents of this directory (except for deleting, if necessary). |
| `content/` | Base directory for web access. This directory contains all modules and data that is available via the browser. |
| `data/` | Database directory. This directory contains the ORM classes that map table structures to PHP objects. |
| `fim/` | Directory that contains FIM's source files. Do not change anything in this directory unless you are very certain. This directory might also be located in an include path instead of CodeDir. |
| `locales/` | Locales directory. The source and compiled ICU resource files are located in this directory. |
| `logs/` | Automatically-created directory that contains the logfiles. |
| `plugins/` | Plugin directory. A plugin is any kind of external library that is not delivered with FIM but that shall be available for use in your source code. |
| `script/` | Base directory for console access. This directory contains all modules and data that is available via command line interface. |

**Table 1** – Overview about FIM's top-level directory structure

**Listing 2** – The server configuration: `.htaccess`

```
RewriteEngine On
RewriteRule ^.*$ index.php [QSA,L]
<IfModule mod_xsendfile.c>
   XSendFile On
</IfModule>
```

In order to configure Apache properly, you also need to copy the `.htaccess` configuration file to the same directory as the controller file. This configuration shows two important points:

- `mod_rewrite` is required for using FIM. This module *has* to be activated. If you are using a server software different to Apache, there are similar settings or extensions that have to be configured in a way that every request is forwarded to the controller file.
- `mod_xsendfile` should be used and activated. XSendFile is a module that allows resource-friendly delivery of files. Similar modules exist on other server softwares. FIM is able to automatically detect which server is used and will determine which measures to take to use the feature. It can however not determine if the module is present or installed on the server. This means that if you do *not* have XSendFile (Apache, Cherokee), XSendFile2 (Lighttpd) or X-Accel-Redirect (Nginx) available although you are using one of those servers, you have to configure FIM not to use it. For this purpose, you need to modify a line in the controller `index.php`:

```
   fimInitialize(['x-sendfile' => 'none']);
```

After you have set up this two files, **F**ramework **Im**proved is ready to go.

## 2.2 The demo application

Our demo application shall be a to-do list manager. We therefore need four pages:

- **Home**
  This page will display the list of to-do items.
- **Add new item**
  This page will provide a form for adding a new item.
- **Edit item**
  This page will provide a form for editing an item.
- **Delete item**
  This page will confirm that we want to delete an item and then delete it.

As this design is quite simple, you may not yet realize the differences of FIM in contrast to many other frameworks. While it is quite common that a framework is based on modules which run actions—so we have a three-level hierarchy: controller, module, action—, FIM takes a slightly different approach. It is often necessary that actions perform quite complex operations, while belonging to a single module. It would be perfectly possible to split actions in different functions and thus to structure the jobs. However, this makes the whole module itself messy, containing lots of functions that belong to different actions. For this purpose, FIM allows a much bigger hierarchy. The controller invokes the module, but every module does only implement one single action. As every module has its own folder, structuring workflow works very effectively.
We will store information about our to-do items in a database. A single table will suffice with the following fields:

| Field name | Type | Null? | Notes |
|---|---|---|---|
| `id` | `integer` | No | Primary key, auto-increment |
| `title` | `varchar(100)` | No | Name of the file on disk |
| `completed` | `tinyint` | No | Zero if not done, one if done |
| `created` | `integer` | No | Date that the to-do item was created |

FIM works with PDO, so lots of database drivers are supported. In this demo application we will use MySQL. Create a database called `mytasklist` and run these SQL statements to create the task_item table and some sample data:

```sql
CREATE TABLE `task_item` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(100) NOT NULL,
  `completed` TINYINT NOT NULL,
  `created` INT NOT NULL,
  PRIMARY KEY (`id`));

INSERT INTO `task_item`(`title`, `completed`, `created`)
  VALUES('Purchase conference ticket', 0, UNIX_TIMESTAMP());
INSERT INTO `task_item`(`title`, `completed`, `created`)
  VALUES('Book airline ticket', 0, UNIX_TIMESTAMP());
INSERT INTO `task_item`(`title`, `completed`, `created`)
  VALUES('Book hotel', 0, UNIX_TIMESTAMP());
INSERT INTO `task_item`(`title`, `completed`, `created`)
  VALUES('Enjoy conference', 0, UNIX_TIMESTAMP());
```

Now we need to make FIM aware of this table. Create a directory under CodeDir that is named `data`. This directory will contain all table representations. So we need to create a new PHP file that is called `task_item.php`, which is exactly the table name plus the extension `.php`.
FIM organizes all files with the help of namespaces. In a properly set-up application, you should never need to include files by yourself. FIM will decide where to look for the required source files by examining the namespace of the unknown class: It will always be the directory path, relative to CodeDir. The only exception is made for the files that come with FIM itself: Classes that shall be accessed by you are in global namespace. Classes that are for internal use only are in the namespace `fim`. It should never be necessary to access a class or function in this namespace directly.
So our newly created file starts like this:

```php
<?php

namespace data;
```

Next, we need to supply FIM with the necessary information about the table. As our table has a primary key, we can inherit the class provided by FIM `\PrimaryTable`. Note the starting backslash, which will assure that we are in the right namespace.

```php
class task_item extends \PrimaryTable {

}
```

Now FIM knows about the existence of the table, but nothing about its structure yet. By delivering a static property `$columns`, we will specify the table structure:

```php
   protected static $columns = [
      'id' => '+int',
      'title' => 'string',
      'completed' => 'bool',
      'created' => 'int',
   ];
```

Now we have to decide what features should be possible with this table: We certainly want to create new tasks and we also want to delete existing ones. Then we will need to fetch all items—and we need the possibility to get a single task by its id. This leads to four methods:

```php
   /**
    * @param string $title
```

```php
     * @param bool $completed
     * @return self
     */
    public static function create($title, $completed) {
        return parent::createNew((string)$title, (bool)$completed, time());
    }

    /**
     * @return bool
     */
    public function delete() {
        return parent::delete();
    }

    /**
     * @return self[]
     */
    public static function fetchAll() {
        return parent::findBy([], '"completed" ASC, "title" ASC');
    }

    /**
     * @param int $id
     * @return self|null
     */
    public static function fetchById($id) {
        return parent::findOneBy(['id' => (int)$id]);
    }
```

All methods have in common that they are very small and only invoke methods of the parent class.

The first method, `create`, is used to create a new object. The function that is responsible for this action in the parent class is called `createNew`. It is protected by default and requires its parameters to be the fields in the same order as specified in the `$fields` array. The autoincrement key must not be passed as a parameter. The need to write a wrapper function allows to ensure that all parameters are of the right type; furthermore, one can include validity checks or default values.

The second method, `delete` is only used to make the parent-class function `delete` public. It can however be used to include checks which assure that deletion is possible.

The third method, `fetchAll`, retrieves all tasks that are stored in the database and returns them in an array. For this purpose, it uses the method `findBy`. As a first parameter, this method requires an array of `WHERE`-restrictions; as second parameter, an `ORDER BY` string may be given. Note that in MySQL, column names normally have to be enclosed in backticks (`` ` ``), whereas SQL standard defines the double quote (") as the character required for this. When using MySQL, FIM will automatically issue a command that makes MySQL aware of this

SQL standard, so using the double quote will be valid (use single quotes for strings). This simplifies changing between database systems.
Finally, the last method, `fetchById`, retrieves one single task. The arguments for `findOneBy` are the same as for `findBy`. The difference is the result type: Only the first object that fits the query is returned or `null` if there is none.
**F**ramework **Im**proved provides `find...` as well as `get...` methods. The `getBy` and `getOneBy` functions work as their `find` counterparts, but they allow to enter more complex conditions that cannot be reduced to fitting a single value. See the doccomments for further information.
As a last step—which is not required, but handsome—we create doccomments for our class:

```
/**
 * @property-read int $id
 * @property string $title
 * @property bool $completed
 * @property int $created
 */
```

Our database layer is now complete. The database object can be used in modules, which we will create next.

## 2.3 Creating modules

You already know that FIM organizes its content in modules. When you try to access our demo application at the current state, you will be presented with an error message: `Web access is not set up.`. This is because FIM can be used in web mode as well as in command line mode. At the current state, no option of these is available. To activate web access, simply create the folder `content` under CodeDir. When you now refresh our demo application, you will see a directory listing of the `content/` directory in FIM style instead. Unsurprisingly, this listing does not contain much more than the `"."` virtual folder.
Note that directory listing can be configured: By default, it is only available in development mode. You may switch to production by adding the array entry `'production' => true` to the configuration. The configuration is the array that is passed to the function `fimInitialize`. When you now reload the demo application once again, you will see a 401 (Forbidden) error message.
FIM has just created another directory for you: `logs/`. This directory already contains three files which are worth a short explanation.

- **customError.log**
  This file is a user-defined error log. FIM will not write anything into this log file unless you call `\Log::reportCustomError` in your code.

- **error.log**
  This is the general purpose error log. It should contain one line: `The error 401 was triggered while invoking the URI ␣.`, where ␣ is the URL that you just called when you got a 401 error.
  Although the 401 error was logged here, this is not the main purpose for this log file. It will contain all kinds of errors that arise because of errors in your code or configuration. This includes fatal as well as non-fatal errors: Unhandled exceptions will be logged as well as language keys that did not exist.

- **internalError.log**
  This is the log file for all errors that occur within FIM, but that are most likely not due to an error you made, but due to bugs of FIM, PHP or the operating system. In other words: It contains all errors FIM did not expect. If an error is logged here, it left FIM in an unstable state and it is possible that this state will be permanent if you do not take countermeasures. Examples are when writing a cache file failed. This problem can be fixed e.g. by verifying that FIM has the necessary access rights to the cache directory, it is not write-protected, there is enough space or whatever comes to your mind. But FIM cannot continue working without this cache file being written. However, this might only affect parts of your application and might not be noticed without viewing the log.

FIM has a built-in mailing function that allows automatic notification when errors occur. Some errors that are uncritical or duplicate very often will never trigger a mail, but others do. For this purpose, set the configuration entry `'mailErrorsTo'` to your e-mail address. If your mail server requires a certain From header, you may specific it in the entry `'mailFrom'`.
But we now want to create our modules. For this purpose, its recommended to turn the production state off again, so that error messages will be displayed directly to you.
As mentioned earlier, every page is a module. We have assembled a list of all the pages we want to create before. So let's go!
Our first page is the home. This page will display all the to-do items. We can choose: Either we could put the module directly in our `content/` directory or in a subdirectory. We will select the second option, as it is more straightforward for the developer. So let's create a directory called `tasks` in our `content/` directory. We chose this name as it describes the purpose of the application itself. We also have static resources such as CSS and JavaScript which we want to keep separate from our programming logic.
We will now need to change this directory from a "normal" directory to a module. This is done by adding a file named `fim.module.php`. Any file that starts with `fim.` is considered a special file that will not be accessible to the user via a browser. But only a small number

of filenames have a special meaning for FIM. Now we need to fill this file with content:

**Listing 3** – Our first module, CodeDir/`content/tasks/fim.module.php`

```php
<?php

namespace content\tasks;

class Module extends \Module {

   public function execute() {
      $this->title = 'My task list';
      $this->tasks = \data\task_item::fetchAll();
      $this->displayTemplate('tasks.tpl');
   }
}
```

This is very nice, as it shows lots of interesting aspects:

- Once again, our namespace represents the directory structure.
- The class name of our module is `Module`. This is required by FIM for any module class. Therefore, it is clear that only one module per namespace—i.e. per folder—can exist.
- The parent class of every module is `\Module`. While it is possible that a module might inherit from another module, `\Module` has to be the very superclass.
- The function `execute()` is triggered when a directory with a module file in it is requested. This does not mean that a module has to implement this method: If it does not, a 404 (not found) error is triggered, which may be caught in any module at a valid point of the hierarchy. But we will talk about errors in subsection 2.7.
- We can directly call our functions that were defined earlier in the table class; FIM will automatically load the class file.
- FIM uses a template system; a template is displayed by the method `displayTemplate`. This method expects one parameter which points to the template file.
- FIM introduces its own path system. This will be very convenient to you, as you never have to deal with the problem of where your code is located absolutely. Furthermore, the path system is platform-independent; FIM takes care of all necessary transformations. The path delimiter is the forward slash. While you can use backslashes, FIM will always convert them to forward slashes. Although Windows uses the backslash as path separator, it accepts the forward slash, so there is no reason to take Windows in

account.

Relative paths are used as known from the console. The meta directories `..` and `.` exist as in every file system.

Absolute paths are handled a bit differently. The "root directory" of FIM is CodeDir. You cannot go beyond this directory; so it would be nice to define this directory as `/`. But on the other hand, any kind of URL that is used cannot go beyond ResourceDir (ResourceDir is a constant that is either `content` or `script`, depending on whether FIM runs in the browser or in the console). So we would also like to safe us from typing `/content/` or `/script/` all the time. FIM's path system is built on these ideas:

Use *two* forward slashes at beginning of the path in order to indicate that the path starts at CodeDir, so at the very top.

Use *one* forward slash at the beginning to indicate that the path starts at ResourceDir. This is of course dependent on the current execution context, so you shouldn't make use of this path style in functions that are used body in CLI and in web mode if you do not really know what you do.

A note at the end: FIM does not change the behavior of PHP's internal file functions, as some third-party libraries may depend on them. So if you execute the statement `rename('/oldname', '/newname');`, this will search the files in the file system's root, not in FIM's. But FIM does provide a convenient way for this: You may either use the function `\Router::convertFIMToFilesystem()` before you pass the filename to internal functions or you can make use of the `fim://` protocol. Protocols are prepended to the filename and indicate the "way of retrieval"; the default protocol is `file://`, which may be (and normally is) omitted[1]. So the above call of the internal function `rename` can be rewritten as `rename('fim://oldname', 'fim://newname');`. You *may* use this syntax for convenience; but keep in mind that it is not as fast as first converting the filename—and it will possibly cause confusion: As every protocol *has* to use two forward slashes as separator, using *two* of those will be relative to ResourceDir, using *three* will be relative to CodeDir; real relativeness requires a starting dot after two slashes. This is not intuitive, but a design that comes out of PHP's protocol support[2]. As a summary:

| | | | | | |
|---|---|---|---|---|---|
| `fim:///path` | ↔ | `//path` | `fim://./path` | ↔ | `path` |
| `fim://path` | ↔ | `/path` | | ↔ | `./path` |

[1]There's a little difference between a path starting with the `file://` protocol and a path without protocol. The former cannot be relative while the latter of course can.

[2]Please keep in mind that Smarty has its own protocol system. Smarty protocols only use the colon as separator, without slashes. The `fim:` protocol is implemented in Smarty as well (that's actually the way FIM achieves that you can use FIM paths in Smarty templates), but it is set as *standard* here. So any path that is provided in templates automatically uses FIM style unless you prepend `file:` or a different protocol (which you generally should not do).

- The current working directory is the module's directory. By specifying the filename `tasks.tpl`, this file is looked for in the current working directory which is location of the `fim.module.php` file. *Do not rely on the include path! It will not work with FIM!*
- Assigning variables to `$this` makes them available to the template.
- Modules can be quite small...

You can be tempted to think that accessing the database in line 9 will work by now. But do not forget what we have done so far: We created the database and the table in MySQL, the database layer in FIM and accessed the database layer. But what's still missing is the connection between MySQL and FIM. Why didn't we specify host, username and password right along with the database layer?

MySQL and lots of other databases that PDO supports are multi-user databases where different users can have different privileges. Imagine the scenario that in a kind of very deep backend for super users, you wish to provide the possibility to create new tables, alter or drop existing ones and so forth. So really powerful features that are normally not needed in web applications. Or imagine there is a front-end which shall only have the SELECT privilege.

You might think that end users never connect to the database, they can only interact via the application that you provide—and if you don't provide features to do this and that, it won't be done. That's a fair argument, but you might be in a team and not everyone is as reliable as you or you provide the possibilities "just for development", to make things easier, and then... Remember that anything that can possibly go wrong, does.

So you create different database users with different privileges; therefore, you cannot specify connection information in the database layer, which is the same for your whole system. Instead, you put the information in your content directory and it takes effect for all modules that are at or below the level of the connection file in path hierarchy.

As we have quite a simple architecture, we will only create one connection file directly in the `content` directory (which should be the most common case). This file is named `fim.database.txt`—so again it is not accessible from the outside (lucky as we are...). The file is in an INI-like format which is explained in-depth in the reference. For MySQL, it will look as follows:

```
driver=mysql
host=127.0.0.1
database=mytasklist
user=<username>
password=<password>
persistent=true
```

The given user will need SELECT, INSERT, UPDATE and DELETE privileges. You may leave `persistent` to `false` if you wish to; this setting specifies whether existing connections shall be reused from request to request.

Next, we need to create the template that shall be displayed. If we called the module at the moment, we would be presented with an error message. So let's create a file named `tasks.tpl` in the module directory:

**Listing 4** – Our first template, `/tasks/tasks.tpl`

```
{extends '/base.tpl'}
{block 'body'}
   <p><a href="{url 'add'}">Add new item</a></p>
   <table class="table">
      <tr>
         <th>Task</th>
         <th>Created</th>
         <th>Completed?</th>
         <th>&nbsp;</th>
      </tr>
      {foreach $tasks as $task}
         <tr>
            <td>
               <a href="{url 'edit' task=$task->id}">{$task->title}</a>
            </td>
            <td>{L formatDate=$task->created}</td>
            <td>{if $task->completed}Yes{else}No{/if}</td>
            <td>
               <a href="{url 'delete' task=$task->id}">Delete</a>
            </td>
         </tr>
      {/foreach}
   </table>
{/block}
```

As you see, this template file will not compile on its own but instead only extend a file called `base.tpl`. We did this because HTML header and footer will be this same on every page and we do not want to repeat it. As mentioned before, every path or filename that appears within templates automatically uses FIM style. This means that `/base.tpl` is a file that is located in ResourceDir, i.e. `content/`.

Although there are once again lots of interesting things to point out, let's first create this `/base.tpl` file:

**Listing 5** – The parent template, `/base.tpl`

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <title>{$title}</title>
      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
      <link rel="stylesheet" type="text/css" href="{url '/css/bootstrap.min.css'}" />
      <link rel="stylesheet" type="text/css" href="{url '/css/bootstrap-theme.min.css'}" />
      <link rel="stylesheet" type="text/css" href="{url '/css/style.css'}" />
      <link rel="shortcut icon" type="image/vnd.microsoft.icon" href="{url '/img/favicon.ico'}" />
      <!--[if lt IE 9]>
      <script type="text/javascript" src="{url '/js/html5shiv.js'}"></script>
      <script type="text/javascript" src="{url '/js/respond.min.js'}"></script>
      <![endif]-->
      <script type="text/javascript" src="{url '/js/jquery.min.js'}"></script>
      <script type="text/javascript" src="{url '/js/bootstrap.min.js'}"></script>
   </head>
   <body>
      <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">
         <div class="container">
            <div class="navbar-header">
               <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                  <span class="icon-bar"></span>
                  <span class="icon-bar"></span>
                  <span class="icon-bar"></span>
               </button>
               <a class="navbar-brand" href="{url '/tasks'}">
                  <img src="{url 'img/fim-logo.png'}" alt="FIM" />
                  My task list
               </a>
            </div>
            <div class="collapse navbar-collapse">
               <ul class="nav navbar-nav">
                  <li class="active"><a href="{url '/tasks'}">Home</a></li>
               </ul>
            </div><!--/.nav-collapse -->
         </div>
      </nav>
      <div class="container">
         <h1>{$title}</h1>
         {block name='body'}{/block}
         <hr />
         <footer>
            <p>&copy; {date('Y')} &mdash; All rights reserved.</p>
         </footer>
      </div>
   </body>
</html>
```

So now we are done with the template files of our first page. Let's have a deeper look on what we wrote:

- FIM provides a template function that is called `url`. You call this function whenever you wish to point to another resource of FIM. The function expects at least one parameter which contains the URL that is pointed to. This first parameter doesn't have name. You may pass further parameters to the function as well, but these require a name. This is needed when linking to a module: Modules may expect parameters in their `execute` function. The name of these parameters is identical to the name you have to pass in here.
  Note that you must not make use of the `url` function when linking to a resource that lies on an external server.

- Smarty's auto-escape function is turned on by default. You should not change this feature; if you wish to output a variable directly—without `htmlspecialchars` applied—, use the `nofilter` keyword. This keyword works for the `url`, `lurl` and `L` functions as well.

- Another template function provides easy access to FIM's internationalization functions, which are encapsulated in the class `I18N`. As it is very uncomfortable to write `I18N->getLanguage()->...`, the very short-named template function `L` is a replacement (which is also faster).
  The `I18N` class is described in detail later on. It supplies several functions; the most important one should be `get()`, which is used whenever a language key is fetched. But as we did not localize our application so far, this function doesn't play a role. Even without this function, the `I18N` class is very helpful. Its function `formatDate` takes a UNIX timestamp and converts it to a human-readable string representation of the date as required by the locale that is set at the moment.
  The aforementioned function `lurl` is a combination of both template functions. It calls `I18N::translatePath` on a path that was gathered with the `url` function. But as this function is only important for multi-lingual applications, we will not talk about this at this moment.

## 2.4 A preview in the browser

You can now execute the demo application in the browser. When you call the main directory, you will still see a directory listing. This is an issue we'll soon deal with. Then navigate to the `tasks` folder. You will now see a web page that contains a list of our four sample tasks

with a proper-formatted date.
However, this web page is still very old-fashioned. When you now open the default log file `logs/error.log`, you will see lots of 404 errors. The reason of this is that although we embedded lots of style and script files in our template, we did not create these files yet.
So clear the log file and copy the directories `css`, `fonts`, `img` and `js` from the demo application archive into your `content` folder. Now refresh the page and it should look much better; your log should not report any error anymore.

## 2.5 Performing actions

There are several possibilities of what could be done next. Let's now add the modules for adding, editing and deleting a task. Creating the first module took about eight pages for you to read, but I promise that we won't need such an amount for the next three modules altogether!

### 2.5.1 The add module

The next steps should be very familiar: We create a directory called add under `content/tasks`. This directory will get the file `fim.module.php`:

**Listing 6** – The `/tasks/add` module

```php
<?php

namespace content\tasks\add;

class Module extends \Module {

   public function execute() {
      $this->title = 'Add new task';
      if(\Request::isPost()) {

         $taskTitle = trim(\Request::post('taskTitle'));
         if($taskTitle === '')
            $this->emptyTitle = true;
         else{
            \data\task_item::create($taskTitle,
               \Request::boolPost('taskCompleted'));
            $this->redirect('/tasks');
         }
      }
      $this->displayTemplate('add.tpl');
   }

}
```

This module isn't very long either, although it creates a new task, checks the input for validity and redirects back to our main page.
If the request mode was POST—so if a user submitted the form—, we retrieve the new title and remove leading and trailing whitespaces. If our title is now empty, we assign a template variable; if it isn't, we call the `create` function of our `task_item` class. Then we redirect back to the task listing with an external redirect that is executed by the user's browser. It's worth noticing that a call of the function `redirect` (as well as for the function `forward`, which will perform an *internal* redirect) aborts the execution of the module, so our template never gets displayed when we did a redirect.
We still need a template:

**Listing 7** – The `/tasks/add/add.tpl` template

```
{extends file='/base.tpl'}
{block name='body'}
   {if isset($emptyTitle)}
      <p class="alert">Please enter a task name.</p>
   {/if}
   <form action="{url '.'}" method="post">
      <table>
         <tr>
            <th><label for="title">Title</label></th>
            <td><input type="text" name="taskTitle" id="title" value="" maxlength="100" required="required" /></td>
         </tr>
         <tr>
            <th><label for="completed">Completed?</label></th>
            <td><input type="checkbox" name="taskCompleted" id="completed" value="1" /></td>
         </tr>
      </table>
      <input type="submit" value="Add" />
   </form>
{/block}
```

You may now click at the link "Add new item" and check what we have done! First, leave the form empty and submit it. This should not be possible unless you use an old browser because of the HTML5 attribute `required`. Next, set the name to a space. Now the browser allows you to submit, but our code complains about the missing task name. At last, we should add a real item. Enter any name, check "Completed" or don't and submit. Voilà—our task appears in the list.

### 2.5.2 The edit module

We create the directory `content/tasks/edit`. Our module will look like this:

**Listing 8** – The /tasks/edit module

```php
<?php

namespace content\tasks\edit;

class Module extends \Module {

   public function execute(\data\task_item $task) {
      $this->title = 'Edit task';
      if(\Request::isPost()) {
         $task->completed = \Request::post('taskCompleted');
         $taskTitle = trim(\Request::post('taskTitle'));
         if($taskTitle === '')
            $this->emptyTitle = true;
         else
            $this->redirect('/tasks');
      }
      $this->task = $task;
      $this->displayTemplate('edit.tpl');
   }

}
```

Now there is one new thing. We required the module to get one parameter which is called `$task` and which is of our table's type. So how does FIM know where to take this parameter from?
By default, the values for any parameter that is given to a module's execution function is taken from the GET values. This means that those parameter are generally to be considered unsafe! But GET parameters are mere strings, so why won't PHP complain when we set the typehint to our table? The answer is very easy: *Without* the typehint, the parameter would be a string; *with* the hint, it is an object of the class `\data\task_item`. The typehint is not there to verify the correct type, but to indicate it! This very convenient feature of FIM is called *autoboxing*. Autoboxing can be applied to every class that implements the `Autoboxable` interface. As `PrimaryTable`, the superclass of our table, does this, we can use autoboxing to convert a task's id to the task object.
PHP's capabilities of typehints are not very elaborated; you cannot for example indicate primitive types. That's why FIM also evaluates the docblocks of methods; this is discussed later.
One question remains: What happens if no task or an invalid one was given in the URL? As we did not allow `null` for our parameter, our module will never get called. Instead, a 404 Not Found error is triggered. If we had allowed `null`, our parameter would be `null`. So autoboxing can reduce the amount of checks and transformations that you need to perform on user input

drastically.
We still need to create our edit template. As it is almost identical to the one used for adding and doesn't introduce anything new, it is not printed here. You may copy it from the archive. Try to edit an item by clicking on its name!

### 2.5.3 The delete module

The last directory that we create is `content/tasks/delete`. This module's purpose is to ask for a confirmation, then to delete the item and redirect back. You should be able to create the two files on your own by now, but you can also use the files from the archive.

Now we are done with all the modules—and it took only three pages for three modules!

## 2.6 Speed

When you are calling the pages from the browser, you may experience some slowness. This has three possible reasons:

- FIM is a framework and as such slows you application down. This is a drawback that affects each and every framework, although FIM is built to be fast. You are encouraged to use a PHP opcode cacher such as APC (which comes with PHP) or XCache.
- You have just called a new page. The first call of every page requires FIM and Smarty to create several cache files. Those files will fill the folder `//cache` and will speed up execution, so this only affects the first call of a new page.
- You have XDebug enabled. While I encourage you to use XDebug in development, it affects speed badly. FIM is built to be high-performing even if XDebug is enabled[3], but it can't do magic. On production servers, XDebug shall be turned off.

## 2.7 The main module

As we have seen before, our main page will present us with a directory listing (in development mode) or a 401 Forbidden error (in production mode). This is very uncomfortable. But does it mean that we have to create a module in this folder?
Basically, yes. We will need a module for one particular reason: error handling. At the moment, when we access a resource that does not exist, we are presented a textual error

[3]This is done by omitting curly braces where possible. XDebug will not generate debug information for loops or conditions without braces, which improves speed. But follow your styleguide as you did before.

message and the failed access is written to the log. The first aspect is not nice: While the error message suffices to tell the user what has happened, it should definitely be in the style of the rest of the application. The latter aspect is a nice feature for development, as you can immediately see when you have linked to a module or file that does not exist and thus you may resolve these invalid links. But in production you do generally not want to log all those access errors—they might not have been triggered by a wrong link you supplied, but by external mislinks or bots. That's why FIM only logs those errors in development mode.
So let us create a module in `//content` that is responsible for error handling.

**Listing 9** – Error handler module, `/fim.module.php`

```php
<?php

namespace content;

class Module extends \Module {

   public function handleError($errno) {
      $this->title = "Error $errno (" . \Response::translateHTTPCode($errno) . ')';
      $this->displayTemplate('error.tpl');
   }

}
```

As it is very short, let's quickly add the corresponding template:

**Listing 10** – Error handler template, `/error.tpl`

```
{extends '/base.tpl'}
{block 'body'}
   <p>An error has occurred.</p>
   <p><a href="{url '/tasks'}">Home</a></p>
{/block}
```

When you now try to fetch a resource that does not exist, you will see a nice error message in the style of our application. While the template looks very familiar, the module file shows some interesting things:

- As mentioned at the very beginning, a module does not have to have a `execute` method.
- If an error occurs, the `handleError` function is called with the error code as first parameter. But this is not the whole story:
- An error can be triggered manually or automatically. The latter is usually the case when a routing failed (404) or rules denied access (404/401).

- The error is triggered for a certain path. Let's do an example. We tried to access the URL `/demo/tasks/super/man`, while we assume that the DocumentRoot equals CodeDir.
  Now FIM tries to apply routings. We haven't discussed this so far, so let's simply say that FIM wants to transform this URL into a local path. URLs are marked in blue while FIM paths will be green.

  1. `/`
     We start in ResourceDir, as this is the topmost folder that is accessible from the outside. There are no manual routings defined for this folder, so our current directory is `/`.
  2. `/` + `demo`
     There are no manual routings defined for the path "demo". So FIM searches if there is a folder called "demo" in our current directory (files would not be found, as the current directory now contains a module, so direct file access is prohibited). It is, so our new directory becomes `/demo/`.
  3. `/demo/` + `tasks`
     The same as before; our new directory becomes `/demo/tasks/`.
  4. `/demo/tasks/` + `super`
     There are no manual routings for the path "super" in the current directory. FIM searches for a folder of this name, but does not find one.
     Now a 404 error is triggered for the current directory, which is `/demo/tasks/`.

- FIM will now try to handle this error. It searches for modules that can do this job:

  1. `/demo/tasks/`
     This folder contains a module. FIM first checks whether this module implements the specific error handler `handleError404`. It does not, so now FIM looks for the generic error handler `handleError`. This one does not exists either, so FIM propagates to the parent directory.
  2. `/demo/`
     The same as before; we need to propagate.
  3. `/`
     This folder contains a module which provides the generic error handler. A new object of the module class is created—error handler always operate in fresh objects—and the handler is executed.

- An error handler can receive additional parameters. When the error is triggered manually via the `Module::error()` function, you may pass optional parameters which will be present in every error handler.

- An error handler can get further details about the context of the error message. The method `$this->getErrorDetails()` returns an associative array with the indexes `'code'`, `'file'`, `'line'` and `'trace'` which provide information about where the error function was called. This function will return `null` for any error that was triggered by FIM (i.e. routing or rule errors). If you wish to get the URL that caused the error, call `\Request::getURL()`. FIM makes use of the error codes 401 (Forbidden ← rules), 404 (Not found ← routing/rules) and 500 (Internal Server Error ← internal problem while passing a file to the user). You may use any code that you desire (including the ones FIM uses). It is however encouraged that you stick to HTTP error codes because of:

- The error handler may return a value:

  - `self::PROPAGATE`
    This value indicates that the handler might have processed the error or not, but FIM is required to propagate to the next handler in the chain.

  - `self::PROPAGATE_UP`
    As `self::PROPAGATE`, this return value requires FIM to look for the next handler. For this purpose, FIM will not call the generic error handler of the same module. This means that the two above return values are identical when used in a `error()` function, but behave different when used in a specific error handler.

  - `int`
    Any integer that is returned is set as the HTTP status code of the response (but only if this integer is a valid status code).

  - `null`
    If nothing is returned, the HTTP status code will become the error number with which the handler was called.

  - Anything else, e.g. `false` or `true`
    These result values will be ignored and will not change the status code.

  Note that if you require FIM to propagate but there is no further error handler, FIM's default error handler will be triggered (the one you have already seen).

- Note that the name "error" is used in a different context than in PHP. In PHP, errors are mechanisms that print a message to the user, they can have different levels of

gravity—from "harmless" hints to fatal errors—and are related to a problem in your programming code.
The term "error" as used here means "any problem that is (most likely[4]) not related to coding errors".
What does FIM with PHP's errors (assuming there are ones, of course...)?
First of all, every error will be logged in the default log file, `errors.log`. Second, when in production mode, a message will be printed for fatal errors, which informs about the error, but does not reveal any details. When in development mode, any error will be printed to the user as done by PHP's default error handler.
An exception is made for error of type `E_RECOVERABLE_ERROR`. Those errors are something crazy. While they are designed particularly for being catched, a `try-catch`-statement will not catch them, but the script will be aborted. This is because `catch` only catches exceptions, but no errors. So FIM automatically converts every recoverable error into an exception of type `ErrorException` which can then be caught.

- We did not talk about exception handling yet. As explained above, a FIM error has nothing to do with exceptions, so we need to discuss those.
  By default, exceptions are handled as errors. FIM will print a detailed analysis of the exception to the user when in development mode, and a hint when in production mode. If you know XDebug, the style of exception dump shall be very well-known for you. FIM's output is designed to be very close to the one used by XDebug (although it is not identical) so that you can get accustomed to it very quickly.
  If you wish to overwrite exception handling, you may create a function in your module with this signature: `public function handleException(\Exception $e)`. The process of propagation is exactly as described for error handlers, but without support for HTTP status codes (you may still change `\Response::$responseCode` if you wish).
  Note that we did not implement an own exception handler in our demo application. While it can be regarded as normal that exceptions are thrown, it should generally not occur that they are uncaught. So implementing an error handler is fine and recommended, as errors can be triggered by a misbehavior of the user; but for exceptions, FIM's own handler should generally suffice. But you may overwrite this default handler—just note that you are responsible for logging the exception with enough details for resolving it!

[4]A 404 error due to a wrongly typed URL is due to a error in your templates. But this kind of problem is indistinguishable to user-caused ones.

## 2.8 Routing

So let us now come to routings, i.e. the way how FIM converts URLs to paths and vice versa. If you call the top page of our application, you will not see a directory listing, but a 404 page instead. This is not very nice. There are two ways to come to grips with this:

- We could create an `execute` method in our top module and perform a redirect to the `tasks` directory. This would perhaps be the favorable approach.
- We might also create a routing that virtually "removes" the directory `tasks` from the URL. As you should really know about routings, we will choose this way.

You have already seen basic concepts of routing when we talked about error handling. The way FIM transforms a URL into a path is described in detail in the previous section—but one explanation is still missing. How can we define manual routings?
Our target will be to apply the following rules (as before, blue means URL, green is path):

- `/css/*`, `/fonts/*`, ...
  Everything that is not located in the `/tasks/` folder shall be accessible as before.
- `/` $\leftrightarrow$ `/tasks/`
  The top URL shall route to the `/tasks` module.
- `/edit/`, ... $\leftrightarrow$ `/tasks/edit/`, ...
  Anything that is located in the `/tasks/` directory shall be accessible as if it were in `/`.

You might think about this twice, then create the file `fim.router.php` in our `content` directory:

**Listing 11** – Our routing, `/fim.router.php`

```php
<?php

namespace content;

class Router extends \Router {

   protected function rewritePath(array $path, array &$parameters,
      &$stopRewriting) {
      if(!empty($path) && $path[0] === 'tasks')
         return array_slice($path, 1);
   }

   protected function rewriteURL(array $url, array &$parameters, &$stopRewriting) {
      if(empty($url) || $url[0] === 'tasks' || is_dir("fim://tasks/{$url[0]}"))
         return array_merge(['tasks'], $url);
   }

}
```

Our router file is not much bigger than the informal description of what we wanted! What is done here?
First of all, we extend the base class `\Router` for all routings. This *requires* to implement the two methods exactly as given in the listing.

- `rewritePath` is responsible for transforming a path into a URL.
- `rewriteURL` does the reverse.
- Both functions receive three parameters. The first one is an array that is filled with the path or URL segments of the requested resource, relative to the directory in which the router is located. The returned value is an array of the same structure, containing the transformed path segments. We will do another example soon.
- The second parameter contains all the GET parameters that the resource originally receives and is altered to contain the GET parameters that the routed resource will get.
- The third parameter can be set to `true` if you wish to prevent FIM from applying further routings. This interrupts the routing chain and can thus cause invalid paths! Use with care (possible use case: A router is located in a deep file structure and you don't want FIM to look for further routings, as there are no ones).
- `null` must be returned, when no routing whatsoever will be applied to the given resource, regardless of the parameters. Assuming that you don't want to rewrite the path or URL for the current parameters, but maybe for others, you *have to* return the input array instead of `null`—everything else will invalidate the cache and lead to unexpected behavior.

I think these explanations were the most abstract ones in this manual; thus, we will do another example.
We will create a second router in the `tasks` directory. It looks like this:

**Listing 12** – A second routing, `/tasks/fim.router.php`

```php
<?php

namespace content\tasks;

class Router extends \Router {

   private $useParameters = true;

   protected function rewritePath(array $path, array &$parameters,
      &$stopRewriting) {
      if(empty($path) || $path[0] === 'add')
         return;
      if(!$this->useParameters && isset($parameters['task'])) {
         array_push($path, $parameters['task']);
         unset($parameters['task']);
      }
      return $path;
   }

   protected function rewriteURL(array $url, array &$parameters, &$stopRewriting) {
      if(empty($url) || $url[0] === 'add')
         return;
      if(!($this->useParameters = isset($parameters['task'])) && isset($url[1])) {
         $parameters['task'] = $url[1];
         unset($url[1]);
      }
      return $url;
   }

}
```

You have to delete the directory `//cache/templates` (or at least any file in this directory). Note that anything that changes static routings to dynamic ones requires purging the template cache!

Please check the application in the browser. You will notice that our top directory no longer displays an error but the task list instead. Furthermore, any link pointing to add/edit/delete actions does not contain the `tasks` directory any more. But this was all achieved with the first routing. What did the second one do?

In other frameworks, it is common not to use GET parameters but to append virtual "subdirectories" to the URL. So while FIM would use a URL like `/edit/?task=1`, other frameworks would rather use the URL `/edit/1`. We won't discuss the pros and cons here. Simply open the first URL in the browser. You will see the edit page. Now view the source code and check for the `<form action="...">` tag. You will see that it equals the current URL.

Then open the second URL in your browser. Thanks to our last routing, exactly the same

page will appear! But if you view the source, you will notice that the tag's value equals the *new* URL, without the GET parameter!
So this is what the router does: When receiving a URL, it detects whether this URL uses FIM parameter style or the other one. In the latter case, the URL is rewritten so that FIM can parse it. When transforming a path back to a URL, our router still knows in which mode the request was issued and does the appropriate rewriting. So that's what the routers do—let's look at the how.

- We receive the URL `/edit/1`. FIM now searches for a router in the top-level directory. There is one; FIM creates an object of this router. *This object is never destroyed for the request; FIM will operate on this single object!*

- The function `rewriteURL` is called with the URL relative to the router directory. In other words: `$router->rewriteURL(['edit', '1'], [], $stopRewriting);`.

- The directory `/tasks/edit` exists; thus our router returns the array `['tasks', 'edit', '1']`; the parameters are unchanged.
  What does this mean?—FIM will *replace* the received URL with the one returned by the router.

- We're done with the top level. Let's go to the next directory in our URL. This is `/tasks`. It doesn't matter that this path segment was not included in the original request; only the output of the last routing is important.
  We switch our current directory to `/tasks`. Is there a router file? There is, so let's create one object of this router.

- The function `rewriteURL` is called with the URL relative to the router directory, so everything that was already routed is truncated. In other words:
  `$router->rewriteURL(['edit', '1'], [], $stopRewriting);`.

- The URL is neither empty nor starts with "add". The router checks whether there is a parameter called "task", but our parameter array is empty. So the field `$this->useParameters` is set to `false`. As we have a second entry in our URL (`'1'`), the parameter array is changed to `['task' => '1']` and the URL `['edit']` is returned.

- We're done with the second level. Let's go to the next directory: `/tasks/edit`. In this directory, we don't have a router file, so FIM simply takes the URL as path.

- We're done with the third level. There is no forth level (any more). Our URL was transformed to the path `/tasks/edit` with one GET parameter: `task` equals `'1'`.

The reverse process should also be clear by now; just keep in mind that FIM works outer-to-inner when transforming URLs to paths (so the outermost path is routed first) and inner-to-outer when transforming paths to URLs (so the innermost path is routed first)—it is a true "reverse" process.
I want to point out the difference between static and dynamic routings once again, considering the most recent router file. If our path is empty (→ list of tasks) or points to the add action, we return `null`. This tells FIM that we will never, under no circumstances change these paths in this router file. So FIM can improve caching a lot and completely removes the call of this router. On the other hand, when we have a path to the edit action and even don't rewrite the path at the moment (because of `$this->useParameters` is `true`), we *will* rewrite the path in certain circumstances (if the variable is `false`). So we return the unchanged path array; now FIM knows that although no routing has been done, it cannot remove this router from the chain. That's the difference between static and dynamic routings and the reason why you need to clear the template cache whenever a static routing becomes dynamic (and you should do as well when a dynamic routing becomes static—this improves speed a lot).

## 2.9 Let's get international!

We have already made use of FIM's `I18N` class to format dates. But this class can do much more, and that's what we will do now: We will add support for a second language to our project.
This section requires changes in our templates and the module we have written so far. This is an issue that is due to the idea of this guide and a task you won't have to do in your real projects. You are highly encouraged to always use the capabilities of `I18N` instead of hardcoding text in the templates (or even wore, as we did, in modules). If we had done this before, this section would not even be a whole page long, but this would have been too complex for the very beginnings with FIM.
There are two archives that ship with this manual. The first one (`Tour1.zip`) ends before this chapter and contains the source of everything we have done so far. The second one, `Tour2.zip` contains the final, fully localized sources.
We have to create a new directory in our CodeDir that is named `locales`. This directory will contain the ResourceBundle files used by FIM and the underlying International Component For Unicode (PHP's `intl` extension). We will then create a subfolder in the `locales` directory called `src` which will contain our language definitions.
This subfolder will get a new file: `root.txt`. This manual will quickly summarize how the file is created with one example; you may copy the final file from the archive.
We need to open a template that contains hardcoded strings. Let's start at the top level:

/base.tpl.
We will deal with the title string later. For now, we'll localize the strings "My task list", "Home" and "All rights reserved." So our language file becomes:

**Listing 13** – Localization of the base template, `//locales/src/root.txt`

```
root:table {
   base:table {
      appName:string { "My task list" }
      copyright:string { "All rights reserved." }
   }
   tasks:table {
      title:string { "Home" }
   }
}
```

The occurrences of those strings in the template are then replaced by `{L ['base', 'appName']}` an so on.
As a last step, we need to compile the language file to a ResourceBundle. For that, the tool `genrb` is needed which comes with the International Components For Unicode (ICU). Open the console and switch to the language source directory. The following command will create the compiled ResourceBundle: `genrb -d .. -e utf-8 root.txt`. We instruct `genrb` to take our source file and compile it, while the output directory shall be one level above the source. The encoding is set manually to UTF-8. This is important especially when there are characters beyond ASCII in the language file. `genrb` will recognize the correct encoding automatically, if a UTF-8 BOM is present in the language source files. While many editors support the creation of UTF-8 encoded files (you need one to create language files!), they often use UTF-8 without a BOM. In this case, the encoding has to be specified manually.
After having executed the command line, a file name `root.res` will appear in the `locales` directory. Although we are only at the very beginning of localizing the whole application, you can already open the start page and see that nothing has changed, although you now called the language function instead of typing the string directly to the template.
This didn't really justify the effort; but remember that adding new languages will become a very easy process by now. Furthermore, we can now get rid of the `$title` variable. This was a nasty thing: By using a template system, FIM encourages the distinction between code logic and output. Titles are definitely part of the output and shall not be located in modules, which do the "hard work".
We can now assign an module identifier instead of a title, which can then be used by the internationalization module to get a valid page title. But the best is that we don't have to assign this identifier, as FIM already did it for us. Every template knows of five variables that

can be used to identify the module that executed the template:

- `$__module` is a string that contains only the last part of the module folder. If the module is `/tasks/edit/fim.module.php`, this variable will be `'edit'`.
- `$__modulePath` is a string that contains the full FIM path of the module folder. The above example would be `'//content/tasks/edit'`.
- `$__modulePathArray` is an array that contains the FIM path items of the module folder: `['content', 'tasks', 'edit']`.
- `$__moduleResource` is a string that contains the FIM path relative to ResourceDir: `'/tasks/edit'`.
- `$__moduleResourceArray` is an array that contains the FIM path items of the module folder relative to ResourceDir: `['tasks', 'edit']`.

In our example, we will make use of the last variable. We will prepend one line to the base template:

`{block name='title'}{$title = {L array_merge($__moduleResourceArray, ['title'])}}{/block}`.

This allows us to remove the line that assigns a title to the template from every module; of course, we still have to add the titles to our language file. It will get a few new lines:

```
root:table {
   base:table {
      appName:string { "My task list" }
      copyright:string { "All rights reserved." }
   }
   tasks:table {
      title:string { "Home" }

      add:table {
         title:string { "Add new task" }
      }

      delete:table {
         title:string { "Delete task" }
      }

      edit:table {
         title:string { "Edit task" }
      }
   }
   error:table {
      title:string { "Error {0,number} ({1})" }
   }
}
```

We now need to recompile the file. If your operating system supports file locks, you will get an error when compiling the file with genrb: Access denied. This is due to the way ICU and intl handle ResourceBundles. A bundle that was once opened will stay open until the server shuts down. This is very convenient in production mode, as it accelerates the process of getting an entry from the bundle. However, in development, it might not please you that you have to restart your server whenever you add, change or remove language definitions. Such a restart is also necessary when you add a completely new locales, as the whole locales directory is cached, not only the files inside.
So restart your server and recompile the file. You will see that although you removed the title assignment from every module, it is still correct in the output.
We now have to further discuss the error page. You may have noticed that we put the title assignment in {block} tags, which allows us to overwrite them in other templates. This is what we need to do for the error page: First of all, our error title string does not follow the pattern of path-matching, as there is no module called /error. Second, our title changes with the error code—and we don't want to create a 64 different language strings for all possible HTTP error code (plus an unlimited number of user-defined codes). So we use ICU's capabilities of message parsing. Our error template needs a little modification:

**Listing 14** – Changing the error template /error.tpl

```
{extends '/base.tpl'}
{block 'title'}{$title = {L ['error', 'title'] $errno Response::translateHTTPCode($errno)}}{/block}
{block 'body'}
   <p>An error has occurred.</p>
   <p><a href="{url '/tasks'}">Home</a></p>
{/block}
```

Moreover, change the error module so that it assigns the value $errno to the template instead of the title.
When you now call an invalid URL, the error page will work as well. This looks very much like magic at the moment. Let's analyze, what we have done:

- Our error string contains two ICU placeholders, the first one being a number, the second one an arbitrary string. This is indicated with the braces around the parameter number. If you are using PHP 5.5 or above, this could be even nicer with named parameters: The language string might then look like "Error {errno,number} ({description})", which requires an adjustment in the language function (of which we will talk now).
- The language template function can be called in two ways. The first one is shorthand syntax: The first parameter of the function is unnamed. This means that the language

function will automatically call the method `get` of the `I18N` class. You may give an arbitrary number of additional parameters which the language function will automatically convert to an array and then pass as second parameter to the `get` function.
The second way to call the function was already introduced long before this chapter. It requires the first parameter to be named. The name of the parameter then indicates which function of the `I18N` class shall be called. All following parameters will be passed as-it to the respective function without transformation.
So the following two calls are identical:
`{L ['error', 'title'] $errno Response::translateHTTPCode($errno)}` and
`{L get=['error', 'title'] [$errno, Response::translateHTTPCode($errno)]}`.

- The `I18N->get()` function expects one or two parameter(s). The first one (string or array) indicates the language identifier that shall be requested. If it is an array, every entry but the last one of this array shall be represented as tables in the ResourceBundle, while the last can be of any value.
  When the returned value is a string and you specify a second parameter, the ICU message formatter will then be called and do the appropriate replacements.
- As in PHP 5.5 named parameters are supported, we could also change the call of the template function to
  `{L ['error', 'title'] errno=$errno description=Response::translateHTTPCode($errno)}` with an adjusted language file.

The remaining process of internationalizing the application is quite boring; you can try it for yourself or copy the changed template and language files from the archive.
I called it "internationalization" and not "localization" because in effect, we did not really change anything for the end-user so far. Our application is now ready to get a second locale. But before there is a second locale, we should create a first one—we didn't really do this so far. For details about how ICU handles ResourceBundles, it's worth reading the page `http://userguide.icu-project.org/locale/resources`. By creating a `root` resource, we assured that if everything else fails, the language keys from this file are used. But it is good style (and might prevent you from getting unexpected results[5]), to create a locale file for every supported locale. This locale file does not contain anything, as it takes its whole content from the root language:

[5]If your system's language is set to e.g. German and we will now add a locale file for German, the root resource will not be available any more, even if you explicitly tell FIM to use English. This is because the root resource is used only if neither the desired locale nor the system's default is available. As we didn't tell ICU that English is supported yet, ICU will always fall back to the system's default—i.e. German!

**Listing 15** – Our English dummy locale, `//locales/src/en.txt`

```
en {
}
```

Compile this file; you will not get a "file in use" error message here, but the changes will not be detected unless you restart the server. This doesn't matter at the moment, as there were no real changes.
Now we create the second locale by copying the `root.txt` file to `de.txt` and then translate every entry step-by-step. You might simply take the result from the archive. Restart the server and refresh our home.
As you can see, you can see nothing. Our application is now bilingual, but the default locale is still set to English. We can for a check change FIM's default locale to German: Set the configuration entry `'localeInternal' => 'de'`. When you now refresh the page, it immediately switches to German.
That's fine, but we rather want the user to decide about the locale. There are several ways to do that. The most common approach would be to take the user's browser language. But if the user wishes to change the locale, this change has to be saved. We could make use of the Session class. This class is not explained in-depth in this document, but it's designed very straightforward. In short, it provides mechanisms to store information for multiple requests. It is an advanced wrapper around PHP's session mechanisms with support for flash items (those will only last until the very next module request).
But the problem then is that one URL points to (at least) two different contents: one in English, one in German. This might not be a big deal, but it violates the concept of *Unique* Resource Locators.
Then we could add a GET parameter to the URL, containing the language. But this is nasty, we have to carry it with us with every single URL we create.
We could also introduce a virtual directory at the top level that consists of the user's language. This would is the best approach so far, but it would indicate that the resources of different languages have nothing in common, which is certainly wrong. Furthermore, FIM generates absolute URLs with the URL function; those would still need to carry the locale folder with them (although FIM would take care of this).
The recommended approach for this is to make use of subdomains. FIM provides intrinsic support for subdomains and automatically maps those to directories up to a level that is defined in your configuration. This directory can then be routed and the router sets the language. Subdomains will only appear in URLs when they change, so the language identifier will be kept away from template source.

## 2.10 Subdomains for language handling

At first, we'll play a little in the configuration. There are several keys that need to be changed. First of all, we want to activate FIM's subdomain handling. This is done by setting `'subdomainDepth'` to `1`, as we want our subdomains to be one level deep. We don't need to set `'subdomainDefault'` for our use case. This setting would allow us to define a default path that is assumed when no subdomain is given.
Now we need to specify `'subdomainBase'`. This key is required when `'subdomainDepth'` is greater than zero. FIM needs to know what our base URL looks like without any subdomains applied in order to determine the correct subdomain settings. If you run the application on a local server, this will most likely be `'localhost'` (or if you use the server configuration as in Listing 16, `'fim.localhost'`). If you want the application to be available via multiple domains, you may as well give an array of all the valid domains to this settings; FIM will automatically determine which one is currently used. `'subdomainBaseError'` allows us to specify a PHP file which will be executed when the application was called with a domain that was not given in `'subdomainBase'`. This file is included by FIM as a normal PHP script which is executed line-by-line. After the inclusion, FIM terminates. Note that neither database access nor session handling or routings work when this file is included. If this setting is not specified, a default error string is printed. There is one important thing to note: FIM handles all URL transformations properly even if it is executed in a subdirectory. However, when used in subdomain mode, FIM *has* to run as a first-level URL. This does not mean that you have to put your application in DocumentRoot, but you have to make sure that it can be accessed by a URL that does not contain a subfolder before your application. This can be achieved by setting a subdomain in your server's configuration that points to the directory your application is stored in.
We have now set up FIM; you will still have to make sure that your server knows of these subdomains. The best thing would be to create a wildcard subdomain. In Apache, the following lines have to be appended to the configuration file:

**Listing 16** – Modifying `httpd.conf`

```
<VirtualHost *:80>
   DocumentRoot "Insert the path to FIM here"
   ServerName fim.localhost
   ServerAlias *.fim.localhost
</VirtualHost>
```

If you work on localhost, your operating system will not be available to resolve the DNS `fim.localhost`. For that purpose, you either have to manually add every subdomain that is used to your hosts file (Windows), which will then look like this:

**Listing 17** – Hosts file `%windir%\system32\drivers\etc\hosts`

```
127.0.0.1 fim.localhost
127.0.0.1 en.fim.localhost
127.0.0.1 de.fim.localhost
```

However, it would be much more comfortable if you install a DNS server such as Acrylic or use the DNS server delivered with your operating system (Linux), which both support wildcard host entries. You can then add the line `127.0.0.1 *.localhost` to the hosts file (or whatever is required to configure your DNS software) and will be happy for all languages. But this is only an issue if you work locally; if you have a registered domain, simply enable wildcard subdomains for this domain (if available).
The next task is to modify our top router so that our application regains functionality:

**Listing 18** – Modified main router `/fim.router.php`

```php
<?php

namespace content;

class Router extends \Router {

   protected function rewritePath(array $path, array &$parameters,
      &$stopRewriting) {
      if(!empty($path) && $path[0] === 'tasks')
         $path[0] = \I18N::getLocale()->getLanguageId();
      else
         array_unshift($path, \I18N::getLocale()->getLanguageId());
      return $path;
   }

   protected function rewriteURL(array $url, array &$parameters, &$stopRewriting) {
      $language = array_shift($url);
      if(empty($language)) {
         \I18N::setLocale(\I18N::detectBrowserLocale());
         \Response::set('Location',
            Router::mapPathToURL('/' . implode('/', $url), $parameters));
      }else
         \I18N::setLocale($language);
      if(empty($url) || $url[0] === 'tasks' || is_dir("fim://tasks/{$url[0]}"))
         return array_merge(['tasks'], $url);
      else
         return $url;
   }

}
```

Now refresh the page in your browser (note that depending on which settings you used

above, the URL may have changed to `http://fim.localhost`). You will be immediately redirected to either `http://en.fim.localhost` or `http://de.fim.localhost`, depending on the language of your browser. If your browser has a different language, ICU and FIM will determine a best-fit to which you will be redirected. You may now manually change the URL to the other one and the whole application changes its locale.
There is one last thing missing regarding the topic of localization: The user should see which languages are supported and it should not be necessary to change the URL manually. Instead, we want to provide flags in the header of each page that allow to change the current page to a different language immediately.
As the buttons shall be visible on each page, we will only edit our main template `/base.tpl`. Within the navigation bar, after our `<ul>`, we will insert these lines:

**Listing 19** – Providing a language switcher

```
<ul class="nav navbar-nav navbar-right">
   <li class="dropdown">
      <a href="#" class="dropdown-toggle" data-toggle="dropdown">
         <img src="{url '/img/flags/'|cat:{L getLanguageId=null}:'.png'}" alt="{L getLanguageName=null}" />
         {L ['base', 'switchLanguage']}
         <span class="caret"></span>
      </a>
      <ul class="dropdown-menu" role="menu">
         {foreach i18nURL(I18N::SUPPORTED_LANGUAGES) as $key => $localized}
            <li{if $localized['active']} class="active"{/if}>
               <a href="{$localized['path']}">
                  <img src="{url "img/flags/$key.png"}" alt="{$localized['display']}" />
                  {$localized['display']}
               </a>
            </li>
         {/foreach}
      </ul>
   </li>
</ul>
```

Most of these are simple design elements. What matters is—once again—how methods of the current locale object are retrieved. This is done by using the `L` template function in its long (explicit) form and using the first key as method name. Even if the method that needs to be called doesn't expect any parameter, we will have to give one so that FIM can notice that this is not shorthand syntax; that's why we pass `null`.
The other interesting thing is the `i18nURL` function. This functions allows to iterate through all available locales, languages, regions or scripts (depending on the first parameter). For every item, the URL function will be applied to a given URL (or multiple ones), which is given as second parameter. If the second parameter is omitted, the current URL will be used. For details, see the function's documentation.—It should be noted that the results of this

function cannot be cached, so it is not terribly fast. If you build applications that expect lots of requests per second, it might be worth the effort to retrieve the menu of available languages via AJAX instead of embedding it in every template.
If you refresh the page, you will now notice the "Change language" button at the top right edge.

## 2.11 Rules

This will be the last big section in our demo application. We did not make use of FIM's rule system yet—we did not require any authentication! This will change by now.
We will design a very simple authentication mechanism without a user table or something similar. It is left as an exercise to you to create a table in the database that contains users with passwords, write a proper table class and then change the implementation below to use this class. This whole process should not require more than five minutes.
We will now create a rules file. There are basically two ways for this: First, FIM allows us to create a rules file by overriding the `\Rules` base class. This allows us to use PHP's full capabilities, but maybe we only want to create basic rules which aren't worth creating a whole class. For this purpose, FIM also allows to make use of its rules text files. Those files are written in an INI-like format that is parsed by FIM and only focuses on the rules itself, omitting all unnecessary PHP syntax.
The only rule that is needed is very simple, we will therefore use the text syntax. The same functionality can be accomplished by using the more verbose syntax which is printed at the end of this chapter for your information.
Create the file `fim.rules.txt` in the folder `/tasks`:

**Listing 20** – `/tasks/fim.rules.txt`

```
[READING]
false =E $file !== '.' && !\Session::get('authenticated')
```

This simple file will be responsible for denying access for all modules that perform actions. The first line defines the section that is controlled. This can be one of `[EXISTENCE]`, `[READING]` or `[LISTING]`. The section defines the state in which accessing a resource will fail or succeed. `[EXISTENCE]` means that FIM will proclaim that the resource does not even exist (404), `[READING]` will issue an "Unauthorized" error (401) while `[LISTING]` allows to control the availability of directory listings at a level that is far more specific than the configuration directive.
The second line assigns the value `0` to the result of the rules evaluation process, if the condition right to the equal sign fits. What does this value mean?

FIM will—once again—iterate through the different directories (from general to specific, top-to-bottom) and evaluate every rule that might have an effect on the current resource. The rule function may return one of the following values:

- `int` (or `null`, which will equal to `0`)
  Positive values indicate granted access, negative values denied access. Zero will not change the current state.
  Values that are bigger in its absolute value will overwrite absolute smaller ones. This allows to be far more precise in terms of overwriting than with the base rules "more specific rules overwrite less specific ones". As FIM cannot determine which rule really *is* specific, the absolute value will serve as hint. Only if two values are absolute-identical, rules of a deeper level will overwrite those of the more general one.

- `false`
  This value will immediately deny access to the resource. No further rules are checked.

- `true`
  This value can be interpreted as "the biggest integer". It will allow access to the resource and can only be overwritten by `false`.

If the result after the evaluation process is zero, FIM will decide whether to allow access by using the configuration setting `'defaultAccessGranted'` (for `[EXISTENCE]` and `[READING]`) or `'directoryListing'` (for `[LISTING]`).
Now we know the meaning of `0`, but what is meant with `=E`? This tells FIM that the value that follows the equal-E sign has to be evaluated by PHP. The whitespaces are only for clarification and not necessary. The variables `$fileName`, `$fullName` and `$file` will be available to get the basename or the complete FIM-normalized filename (rules will only be called for files or folders that are directly in the folder of the rule file itself, so the process of recursive iteration will only iterate through directory names). The capital `E` defines `$file` to be identical with `$fileName` while a small `e` would make it identical to `$fullName`.
The distinction between capital and small is especially important when you use the other means of comparison. Instead of `=E`, you might also write

- `=R` or `=r`, which will apply a PCRE regular expression to the `$file` variable.
  Example: `false =R /^secure.*$/i`

- `=C` or `=c`, which will do a literal comparison with the `$file` variable.
  Example: `false =C secureContent`

Furthermore, there is the possibility to write `match` on the left side of the equal sign. Only the modes `=E` and `=R` (and their small counterparts) are available. Every following line that is indented will be evaluated as PHP code, while the variable `$match` contains the matches of the regular expression if the mode was `=R` or the result of the PHP expression if the mode was `=E`. Example:

```
match =R /^(in)secure.*$/i
 1 = isset($match[1])
 -1
```

There are still two features of text rules missing: The directive `clone=...` will copy the rules of the section `...` to the current section. And finally, you may simply specify an integer or boolean in a line without an equal sign, which will make this the default return value. As evaluation stops as soon as the first expression fits, this should only be used in the last line of a section. This syntax can also be used in `match` items, but it is *required* to be the last one here.

Now we know what is done in our rules file: If the current filename is `.`—this means that the directory of the rules file is accessed directly, i.e. `//content/tasks`—or if our session tells us to do so, we will not deny access; in any other case, we will do.

When you now refresh the page, you'll see that only the listing is available; any navigation will lead to a 401 error.

We want to change this error to a login form. This is very easy; we will add error handling capabilities to the `tasks` module. So create this function:

**Listing 21** – Error handler in `/tasks/fim.module.php`

```
   public function handleError401() {
      if(\Request::isPost()) {
         if(\Request::post('password') === 'admin') {
            \Session::set('authenticated', true);
            \Request::restoreURL();
            return;
         }else
            $this->failed = true;
      }else
         \Request::saveURL();
      $this->displayTemplate('login.tpl');
   }
```

The `\Request::restoreURI()` and `\Request::saveURL()` functions are responsible for storing and retrieving the current URL so that the user is redirected to where they was before receiving an error. As we implemented the error handler without redirection to any login page (which

would be a far more convenient way and is left to your exercise), this is not really necessary in our case. You may take the required template from the archive.
As a last part of this section, I will shortly present the rules file in PHP mode:

**Listing 22** – Alternative rules file: `/tasks/fim.rules.php`

```php
<?php

namespace content\tasks;

class Rules extends \Rules {

   public function checkReading($fileName, $fullName) {
      if($fileName !== '.' && !\Session::get('authenticated'))
         return false;
   }
}
```

And for a very last part, I will only show a short excerpt to FIM's Autoboxing which relies on doccomments by providing the rules function with a third parameter that will automatically be retrieved from the session variable and thus eliminates the call of the get function:

**Listing 23** – Alternative rules file (2): `/tasks/fim.rules.php`

```php
<?php

namespace content\tasks;

class Rules extends \Rules {

   /**
    * @param string $fileName
    * @param string $fullName
    * @param bool $authenticated session
    */
   public function checkReading($fileName, $fullName, $authenticated) {
      if($fileName !== '.' && !$authenticated)
         return false;
   }
}
```

Now `$authenticated` is guaranteed to be a boolean value that comes from the session variable of this name.
Our demo application of course lacks a logout page, it should get a proper login page to which is redirected instead of changing the original page's purpose. Users might be backed in the database and so on. By the end of this chapter, you should be able to perform these tasks by yourself in a few minutes.

# 3 Reference

This section will explain the individual functions of FIM classes that are available in public namespace in an alphabetical order of the file they're located in.

## 3.1 Autoboxing

Any function or method can be autoboxed by FIM, including closures (but not functions created with `create_function`). Autoboxing will try to guess the values of the parameters that shall be given to the function. For this, a function can either indicate the type of its parameter with typehints or use PHPDoc. The doccomment for the function should have the following structure.

```
/**
 * @param <type> $<name> [<source>[:<source name>]] [<comment>]
 * <repeat>
 */
```

- `<type>`
  This will indicate of which type the parameter has to be. It can be any of PHP's primitive types; in this case, the parameter will be casted to this type. It might also be the name of a class that implements the `Autoboxable` interface. It is possible to allow `null` by specifying it as an alternative type using a pipe.
  Arrays with entries of a certain type are possible as well (but only one-dimensional arrays). The type has to be followed by an opening and a closing square bracket; if the type contains alternatives, it has to be enclosed in brackets: `(ClassName|null)[]`.

- `<name>`
  This is the name of the parameter, exactly as given in the function definition.

- `<source>`
  You might specify one of the following keys as indicator where FIM should get the value for the parameter from: `param` (requires the parameter to be given to the autoboxing function), `get` (`Request::get`), `post` (`Request::post`), `session` (`Session::get`), `file`/`filestream` (`Request::getFileResource`) or `filestring` (`Request::getFileContent`).
  It is also possible to use a chunk of PHP code here if you need complete freeness in where to get the source from (in this case, `<source name>` and `<comment>` must not be given[6]): `/** @param int $name \Session::get('user')->name */`.

[6]It is however possible to define a "comment" by using PHP's line comment delimiters `##` or `//`. Note that you have to repeat this delimiter in every new line if you wish multi-line comments.

This field might be left out; in this case, the value is taken from the parameter if available or from `Request::get` if not. This behavior can be copied by specifying `param` as an alternative.
Note that this parameter is *required* if any of the following elements are given!

- `<source name>`
  If this field is given, the name that is passed to the source function will not be the parameter's name but the one given here instead. Note that source names are always case sensitive!

- `<comment>`
  You might give a description for the parameter that contains any character until another PHPDoc element begins.

- `<repeat>`
  The above structure can be given for any parameter of the function. It is however only required for a parameter if you want to change the default behavior `mixed param|get`. But as doccomments are also (or mainly) used for documentation purposes, it is highly recommended that you specify every parameter that is used in the function.

**Calling a function** It is normally not necessary to call autoboxing functions by yourself as the `execute` function of every module is called using autoboxing (including forwarded calls). But you might wish to make use of one of those two class methods:

- `Autoboxing::callMethod($class, $name, array $arguments = [])`
  This method is intended to call a method of a class or object using autoboxing. If a static method shall be called, the first parameter must equal the class name; if an object method is to be called, set `$class` to the object.
  `$name` is a string that contains the name of the method.
  `$arguments` is an optional associative array that shall be filled with parameters that can be passed to the function if they are required.
  The return value will equal the function's return value.

- `Autoboxing::callFunction(callable $function, array $arguments = [])`
  This method can be used when a closure or function—which has to be given as first parameter—should be called. The `$arguments` parameter is as in the function above.

**Autoboxable** Classes that implement this interface can be used as typehint or PHPDoc type. The static method `unbox` will be called with exactly one parameter which is a string.

The function's return value will be used as parameter value. Note that FIM will only check whether the return value actually is of the type that it should be if you also specify a typehint!

#### Implementation remarks

- Calling a function or method with autoboxing generates a cache file in the `//cache/autoboxing` directory. Each PHP source file will create an own cache file; moreover, autoboxing for functions will also create its own cache file.
- It is not possible to apply autoboxing to PHP's or FIM's internal functions.
- An autoboxing parameter must not be of the type `callable` or be passed by reference.
- While FIM can handle autoboxing for closures, those that only comprise one single line, all start and end in the same line, have the same number of (required) parameters *and* the same doccomments cannot be distinguished by autoboxing. This should not be a problem for those who use proper code formatting.
- Autoboxing will only work for public functions. As the calling context changes, this is a drawback that has to be taken in account[7].
- Autoboxing supports inheritance. If a function is not present in the current object, it will look for its implementation in superclasses. Note that the function has to be public there as well! At the moment, PHPDocs have to be repeated if a function is overwritten, there is no support for the (yet unofficial) `/** {@inheritDoc} */` directive.
- An `AutoboxingException` will be thrown if autoboxing failed either during the function call or while preparing the cache.

### 3.2 Config

FIM provides access to its configuration via the `Config` class. The `Config::get($key)` function retrieves the value of the configuration entry `$key` while the function `Config::equals($key, ...$value)` checks if the entry `$key` is strictly equal to at least one of the values given in `$value`. The method `Config::set($key, $value)` changes FIM's configuration by setting the entry `$key` to the new value `$value`. Note that some configuration settings can only be set in the initial configuration in the controller by passing them as an entry in an associative array to `fimInitialize()`. The `Config` class provides three functions that provide further functionality:

[7]It would theoretically be possible to overcome this problem. However, determining the original calling context and switching to it would waste resources.

- `Config::registerShutdownFunction(callable $callback[, ...$parameter])`
  This function will register the function, method or closure that is given in its first parameter as a shutdown function (which will receive all the parameters that can be passed as further arguments here). These functions will be executed at the very end of the script, even if PHP halts unexpectedly. While this function is identical in what it does to PHP's internal function `register_shutdown_function`, it returns an identifier that can be used to unregister the function.

- `Config::unregisterShutdownFunction($id)`
  This function will remove the callable that was registered as a shutdown function from the chain. It expects the return value of `Config::registerShutdownFunction` as parameter.

- `Config::iniGet($key)`
  This function behaves exactly as PHP's internal function `ini_get` with the difference that numbers are recognized automatically, including the suffixes `K`, `M` and `G`. The return value will be an integer in this case.

**Configuration entries** This section will give you an overview of the different configuration entries of FIM. While all these can be passed to the initialization function `fimInitialize`, some entries cannot be changed by using `Config::set`. Possible values are given in square brackets; the first value is the default.

- `'autoEscape'` [`true`|`false`]
  Determines whether Smarty uses automatic escaping of all variables or functions.

- `'cliServer'` [`'CLI'`|`string`]
  The constant `Server` will contain the name of the server that was used to fetch the content (e.g. `example.com` if the URL was `http://en.example.com/admin/`). If FIM is used in client mode, there is no such thing as a "server", so the constant will be set to the value of this setting.
  If the default value is not changed but at least one entry in the setting `'subdomainBase'` exists, FIM will take this entry's value.

- `'defaultAccessGranted'` [`true`|`false`]
  Defines whether `READING` or `EXISTENCE` access is granted to any resource for which there are no further rules.

- `'directoryListing'` [`'development'`|`'simple'`|`'detail'`|`'none'`]
  Specifies the level of detail for directory listings. Simple directory listings will only

show the names of files and directories while detailed ones will include information about size and dates. `'none'` disables directory listing if no further rules explicitly allow it. `'development'` equals to `'detail'` in development mode and to `'none'` in production mode (when getting the value, `'development'` will never be returned but the value it was dynamically set to instead).
If directory listing is set to `'none'` and allowed in a certain folder by rules, the display mode will be simple. If you wish detailed information instead, you have to set this value to `'detail'` and generally forbid directory listing by issuing a negative value in the very top level rules file.

- `'defaultEncoding'` [`'utf-8'`|`'string'`]
  Sets the encoding that is issued by default in the `Content-Type` HTTP header. You may change this encoding for individual files by calling
  `Response::set('Content-Type', '...; charset=...');`.

- `'localeRawFallback'` [`false`|`true`]
  FIM favors the concept of language keys, i.e. a language entry in the locales file has a unique key identifier and a—in most cases—different value. If a language key was not found, FIM will report this error to the log and generate a dummy output which consists of the key with appended parameters.
  It is however possible to instruct FIM to use the key as a last fallback language entry instead by changing this setting to `true`. This means that the key itself is used and passed to the message formatter if no entry in the language was found. Note that this approach is not favorable with ICU and that you might run into problems with special characters or long key names (future versions of ICU may limit the maximum number of characters for keys).
  Note that this value cannot be changed with the `Config::set` method.

- `'localeInternal'` [`'en'`|`string`]
  Defines the locale that FIM uses for its internal messages. This includes error messages as well as e.g. headers in the directory listing. This locale will also be the default one for the application's locale. While the latter one can be changed by using `I18N::setLocale`, changing FIM's internal locale is not possible after initialization.

- `'mailErrorsTo'` [`false`|`string`]
  Any error that is reported in a log will be mailed to the address given here. Mailing can be disabled for individual messages or generally by setting this value to `false`.

- `'mailFrom'` [`'Framework <noreply@framework.log>'`|`string`]
  You may specify the value that should be passed in the `From:` mail header. Note that a specific value might be required here by your provider. This value is the default for all mails that are sent with the `PHPMailer` class that is delivered with FIM but can be overwritten for every single mail.

- `'memcachedConnection'` [`[]`|`(string|[string, int]|[string, int, int])[]`]
  The property `Session::$memcached` provides a `Memcached` object which will automatically connect to the server(s) as specified in this setting. Note that while the setting has to be an array, it can have various substructures. A string will default to the port `11211`, a two-entry array defines host and port, a three-entry array sets host, port and weight.
  Note that FIM will always provide an interface in the `Session::$memcached` property that has the same methods available as PHP's Memcached extension, even if the extension is not installed or only the older Memcache. However, real support for multiple servers is only possible when the new extension is installed (or FIM will connect to the weightiest server instead).
  Changing this value is not possible. If changes are necessary, they can be applied to the object itself.

- `'plugins'` [`[]`|`string[]`]
  A plugin is any kind of external script that is required by your application. In order to be available for autoloading, these scripts have to follow FIM's naming convention: The namespace must equal the directory while the classname is equal to the filename minus the extension `.php` (note case sensitivity on most file systems—always use the same spelling for your class names although PHP is case insensitive!). Any plugin is required to be stored in a subfolder of `//plugins`.
  If there is third-party code that does not follow these naming conventions, it is possible to tell FIM where to look for the file by specifying an entry in this configuration array. The key must equal the class name while the value is a FIM path that specifies the location of the source files. Relative paths will start in the `//plugins` directory. Note that this is the only place where it is possible to use PHP's include path with FIM. You may prepend the filename with a hash sign (#). This will instruct FIM to pass the path following this character exactly as given to PHP's `require` function.
  It is not possible to use autoloading for functions. If your third-party plugin does not provide a class but only a set of functions, it's your responsibility to include the necessary files by yourself.

- `'production'` [`false`|`true`]
  This is the main switch for changing between development and production mode. In development mode, sensitive information is directly echoed in the browser or console to the user for debugging purposes. This can greatly improve the process of development. However, in production mode, this information is only written to the log which allows e.g. to analyze errors, but does not reveal anything about the internal structure to the end user.

- `'redisConnection'` [`null`|`mixed`]
  The property `Session::$redis` provides an interface to Redis. If the PHP extension `Redis` is available, FIM will make use of this and the property will contain an instance of `RedisArray`, which is instantiated with this setting as first parameter and the configuration entry `'redisOptions'` as second. If the extension is not available though you want to use Redis, copy the Predis library files from the `src` folder of the archive to the `//plugins/Predis` folder. As this library matches FIM's naming conventions, it does not have to be registered with the `'plugins'` setting. The property `Session::$redis` will then be an instance of `Predis\Client`, instantiated as above.
  It is not possible to change this setting. If changes are necessary, they can be applied to the object itself.

- `'redisOptions'` [`null`|`mixed`]
  See `'redisConnection'`.
  It is not possible to change this setting.

- `'requireSecure'` [`false`|`true`]
  Defines whether FIM will take care that the application is only invoked with the HTTPS protocol. Any call with HTTP will be redirected to the appropriate HTTPS page.

- `'sessionLifetime'` [300|`int`]
  In order to make session hijacking harder, FIM provides this mechanism which will invalidate the current session after the given amount of seconds. A new session id will be generated and sent to the user. In order to prevent session loss, the invalidated session will still be accessible for the client for number of seconds that can be defined in the setting `'sessionTransition'`.
  Set this value to zero to disable this feature.
  Negative values and those from `1` to `9` are prohibited (but you should not use too small values anyway).

- `'sessionStorage'` [`'default'`|`'memcached'`|`'redis'`]
  By default, FIM uses PHP's integrated session storage mechanism which will save sessions to files. For speed reasons, this is not to be considered optimal. You may change the behavior to use one of the two caches that FIM intrinsically supports. Note that the option `'memcachedConnection'` or `'redisConnection'` has to be set (depending on which storage engine you wish to use).

- `'sessionTransition'` [5|`int`]
  Defines the amount of seconds for which an old session id is still valid after it has been changed with `Session::renewId` or automatically via the `'sessionLifetime'` setting. This prevents nearly simultaneous requests from invalidating each other. If your application shall be optimized for especially slow connections, higher values are recommended. Zero will disable a transition time, which is generally not favorable.
  This value's range is from zero to the value of `'sessionLifetime'` (or `PHP_INT_MAX`, if this setting is zero). Equal values mean that a whole session id renewal can be skipped by a client.

- `'subdomainBase'` [`''`|`string`|`string[]`]
  This setting allows FIM to determine which part of the URL is the constant server and which belongs to the subdomains. This setting is required when `'subdomainDepth'` unequals zero.
  If the application can be called with the URLs `http://*.super.com` and `http://*.super.net`, this setting is expected to be set to `['super.com', 'super.net']`.

- `'subdomainBaseError'` [`''`|`string`]
  Defines a valid FIM path of a PHP file that will be invoked if the application is called with a URL that fits to none of the given servers in `'subdomainBase'`. This PHP file will be included directly and shall present an error message to the user. After the execution of this file FIM will immediately halt. If no file is present when such an error occurs, FIM will throw a configuration exception.
  Note that FIM's initialization could not be completed when this file is included. Therefore, you must not make use of any database function, access the Memcached or Redis object or use methods of the class `Router` that have to do with URL conversion. You might use functions of the `Response` class instead of echoing directly, but keep in mind to call `Response::doSend` at the end of your error script.

- `'subdomainDefault'` [`''`|`'string'`]
  If FIM does not detect any subdomain although `'subdomainDepth'` is greater than zero,

this string will be taken as a default subdomain. Note that you can prevent the user to access certain folders by enforcing a default subdomain! The `//content` directory will not be available any more if a subdomain default is given. If the default contains multiple levels, even more directories will not be accessible to the user.

- `'subdomainDepth'` [0|`int`]
  This settings defines how many subdomains can be passed as a maximum. Subdomains will be implicitly rewritten to directories. The constant `CurrentSubdomain` is available to get the string that was used as a subdomain.

- `'x-sendfile'` [`'auto'`|`'apache'`|`'cherokee'`|`'lighttpd'`|`'nginx'`|`'none'`]
  X-Sendfile or X-Accel-Redirect is a module that has to be installed and/or activated in your server software. It is used to deliver raw files very resource-efficient. With `'auto'`, FIM will automatically determine the server software that is used. If you don't have this module enabled, you need to manually set this value to `'none'`, as FIM can't detect a disabled module.
  Note that you need to configure nginx specially for use with FIM. By enabling the `mod_rewrite` equivalent in nginx, the X-Accel-Redirect technology is also affected. Thus, you need to provide a virtual location that is named `fim.InternalGetContent` which will link to CodeDir. The server configuration file might look similar to this (replace `<path to FIM>` accordingly):

```
http {
   sendfile on;

   server {
      listen 80;
      server_name fim.localhost;
      root <path to FIM>;

      location ^~ /fim\.InternalGetContent/ {
         internal;
         alias <path to FIM>;
      }

      location / {
         try_files $uri /index.php;
      }

      location ~ \.php$ {
         fastcgi_pass  127.0.0.1:9000;
         fastcgi_index index.php;
         fastcgi_param SCRIPT_FILENAME <path to fim>/$fastcgi_script_name;
         include       fastcgi_params;
```

```
        }
    }
}
```

## 3.3 Database

The database class is responsible for parsing connection files. The only function that might be of interest is `Database::getActiveConnection($require = true)`, which will return the instance of the activated `DatabaseConnection` object. If the first parameter is `true`, an exception will be raised when there is no connection; else `null` will be returned. The necessary database connection will automatically be activated when a module is executed.
Connection files (`fim.database.txt`) contain the parameters that specify a database connection. They will take effect for the folder they're located in and potentially for all subfolders. Their format is an INI-like structure (`key=value`), but without sections. The following keys exist (possible values in square brackets, default underlined):

- `recursive` [<u>`true`</u>|`false`]
  If this value is `true`, the settings in this file will also apply for each subdirectory if they are not overwritten there.

- `driver` [`4d`|`cubrid`|`dblibfirebird`|`ibm`|`informix`|`mssql`|`mysql`|`oracle`|`odbc`|`pgsql`|`sqlite`|`sqlsrv`|`sybase`]
  The string that is given here will define the driver PDO uses to establish a connection. If you sometime decide to switch to a different driver and stuck to SQL standard in your queries, no changes shall be required. Some drivers provide proprietary commands; if you use them, FIM can't help you when changing the driver. Other drivers use e.g. different types of escape symbols (SQL standard defines `"` as column name delimiter while MySQL uses `` ` ``). Those specialties can normally be changed and FIM will take measures to make the engines as much standard-compliant as possible (so MySQL will also use `"` as column name delimiter).

- `host` [any string]
  Defines the host PDO will connect to. Can be used with the drivers `4d`, `cubrid`, `dblib`, `ibm`, `informix`, `mssql`, `mysql`, `oracle`, `pgsql`, `sqlsrv`, `sybase`. Note that some drivers may be slow while establishing the connection if you use hostnames (`localhost`) instead of IPs (`127.0.0.1`).

- `port` [any valid port number, <u>default port if available</u>]
  Defines the port PDO will connect to. Can be used with the drivers `4d`, `cubrid`, `dblib`,

`ibm`, `informix`, `mssql`, `mysql`, `oracle`, `pgsql`, `sqlsrv`, `sybase`.

- `database` [any string]
  Sets the database name. For SQLite a FIM path is required (but do not use relative paths!) or `:memory:`.

- `dsn` [any valid DSN string]
  PDO uses DSN strings to establish a connection. FIM uses the above settings to compute the DSN automatically. If you however need to define a DSN manually, this settings will *overwrite* all the above values of `driver`, `host`, `port` and `database`. FIM will not change the DSN in any way. For SQLite this means that you need to provide a filesystem absolute path. ODBC driver requires to use this setting.
  Note: This value will also be inherited if `recursive` is not set to `false` and will thus take precedence over the above values in any connection file in a subfolder. It has to be invalidated with `dsn=`.

- `user` [any string]
  Specifies the user name; might not be required.

- `password` [any string]
  Specifies the password; might not be required.

- `persistent` [`true`|`false`]
  Defines whether the database connection stays open after the current request has finished. Unless you are using ODBC, turning on this setting can improve speed and help to reduce overhead.

## 3.4 Database connection

A database connection is represented in FIM as an instance of the class `DatabaseConnection`. This class provides some methods for establishing and closing a connection, which should not be called by you. Those methods are not listed in the following explanation and not commented in the source.
The other methods of this class provide means to control database functions. You can access the active database connection object with `Database::getActiveConnection`.

- `DatabaseConnection->getConnection()`
  Returns the instance of the `PDO` class that lies behind this connection object. If the following helper functions do not suffice, you may use this object directly.

- `DatabaseConnection->enterTransaction()`

  Starts a transaction. Nested transactions are supported if the database driver supports the `SAVEPOINT` directive.

- `DatabaseConnection->leaveTransaction($commit = true)`

  Leaves a (nested) transaction. Depending on the parameter, the operations are either committed or reverted.

- `DatabaseConnection->execute(PDOStatement $stmt, array $values = [])`

  This is a helper function that automatically binds given values to a PDO statement that was created with the `PDO->prepare` method. The `$values` array is either associative or numeric, depending on whether named or unnamed parameters were used. FIM will automatically determine the necessary binding type by inspecting the value's type. If you wish to pass BLOB data, pass an instance of `Blob` as value (but see the remarks at the end of this subsection).

- `DatabaseConnection->select($from, $where = null, array $bind = null, array $columns = [], $order_by = null, $group_by_having = null, $limit = null, $join = '', $force_statement = false)`

  Executes a select statement. The parameters are documented in the PHPDoc.

- `DatabaseConnection->insert($into, array $columnValues)`

  Performs an insert statement and takes care of escaping the values.

- `DatabaseConnection->update($table, array $newValues, $where = null, array $bind = null)`

  Updates entries in a table that match the given conditions.

- `DatabaseConnection->delete($from, $where = null, $bind = null)`

  Removes entries from a table that match the given conditions.

- `DatabaseConnection->simpleSelect($from, array $where = [], array $columns = [], $order_by = null, $group_by_having = null, $limit = null, $join = '', $force_statement = false)`, `DatabaseConnection->simpleUpdate($table, array $newValues, array $whereColumns)` and `DatabaseConnection->simpleDelete($from, array $whereColumns)`

  Those functions are identical to their "non-simple" versions with the only difference that they expect not a `WHERE` string but an associative array with conditions.

- `DatabaseConnection->lastInsertId($table = '', $keyField = 'id')`

  Gets the last insert id. The arguments are only required when using `pgsql` driver.

**Remarks considering Binary Large Objects (BLOBs)** In PHP, there is no difference between BLOB data and a string. However, FIM needs to know which data is of type BLOB as PDO uses a special binding type for those. The usage of this type might be favorable, as some database drivers (most notably MySQL) will refuse to perform queries (and might close the connection unexpectedly) if they are bigger than a server-defined constant. But don't expect PDO to handle data properly that is larger than allowed by server constraints. Even when using the correct data type, your connection might simply die.
FIM provides the class `Blob` that can be instantiated with the string that holds the blob data as its parameter. Objects of this type can be passed as `WHERE` condition or `$bind` parameter array entry and will be recognized properly.

## 3.5 Executor

This file does not contain any class or function in public namespace. The internal class `fim\Executor` is responsible for the whole process of deciding which data should be passed to the user and takes the appropriate measures. If it is ever necessary to access this class, refer to the doccomments and the source.

## 3.6 FileUtils

FIM provides several methods that help to overcome some of PHP's problems when dealing with files.
Calculation of file sizes greater that exceed `PHP_INT_MAX` will be very unreliable when using `filesize`. For this purpose, the method `fileUtils::size($fileName)` returns the exact size of a file in bytes, regardless whether PHP is running as 32 bit or 64 bit application and which operating system is used. The return value will be an integer or double representation of the file size (while double is potentially imprecise, differences of at least one byte will not occur until a file is at least 9 PB large—if this case ever occurs, you may change FIM's implementation to return a string instead of a number).
PHP might also not recognize the existence or readability of large files; for this purpose, you can use `fileUtils::isReadable($fileName)` or `fileUtils::fileExists($fileName)`.
A problem that only occurs when PHP is running on Windows is dealing with Unicode filenames. As PHP on Windows is not compiled as a Unicode-aware application, those files are inaccessible. However, Windows provides a technique that allows to extend the range of available characters a bit. By *encoding* a filename in a proper codepage, *some* filenames might become available. For this purpose, FIM provides the methods `fileUtils::encodeFilename($utf8FileName, $mask = false)` (`$mask` determines whether invalid char-

acters will become question marks (`true`) or raise an exception) and `fileUtils::decodeFilename($codepageFileName)`. Let's take the filename `€uro.txt`. If you want to call e.g. `filesize()` on this file, you will get an exception—the Euro symbol is character `0x20AC` in the Unicode table and cannot be used in non-Unicode applications. However, assuming the current codepage is 1256 (which is common is western Europe), the character `0x80` (which has no visual representation) will be treated as the Euro symbol. By encoding the Unicode filename, you will be able to get all the statistics of this file. Codepages are nice means to extend the character range, but they are very limited on the other hand. With the codepage 1256, you will—for example!—never be able to access a file that contains a č[8].
Those functions are all intended for use with a single file or directory. If you wish to iterate through directories, you would normally make use of PHP's `DirectoryIterator`. Use the class `fimDirectoryIterator` instead which will do all the necessary conversions and assure that no exception occurs when a file with problematic characters in its name is reached.

**Implementation details** All the problems of PHP can't be solved without the help of extensions or external applications. The `fileUtils::size` function will try to get the file size with cURL, native `fopen` (up to 4 GB) or the execution of an external application. On *nix systems, a call of `stat` will do the job while on Windows, FIM delivers a special application that is invoked. FIM's directory iterator will call this application as well. For this purpose, either the module COM or the function `exec` has to be available.
In case of unexpected results you should have a look at the source. The different situations are well-commented.
Credit for the implementation of code page encoding and decoding goes to Umberto Salsi (icosaedro.it).

## 3.7 FIM

This file provides the function `fimInitialize(array $config = [])`, which has to be called by the controller. The parameter is an associative array that might contain any of the configuration entries as described above.
If you need to executing some code whenever a resource is requested, you may provide the files `fim.startup.php` or `fim.shutdown.php` in ResourceDir which will be executed before and after the executor takes over. These files won't be called when the subdomain base error script is executed.

[8]You may have quickly tried what I just wrote and were successful with accessing the Euro-named file. If you however open your test script in a HEX editor, you will see that Euro symbol was saved as character `0x80`—your source editor did the encoding for you. If the Unicode representation was used, you would have run into problems.

## 3.8 I18N

All internationalization capabilities of FIM are bundled in the class `I18N`. FIM differs between two locales: The internal locale is the one used by FIM to display its own messages while the “normal” locale is the one used by your application. Those two locales are represented as instances of the `I18N` class. You cannot create such an instance by yourself, but you have to make use of the static methods of the class:

- `I18N::getSupportedLocales($checkFor = I18N::SUPPORTED_LOCALES)`
  This method returns an array that contains all available locales (or languages, regions or scripts, depending on the parameter) as strings. Those strings can then be used to call `I18N::setLocale` or for different purposes.
  Note that in contrast to the ICU function `ResourceBundle::getLocales`, this one does not require to create a list of available locales and store this list in a resource file but instead checks the `//locales` directory. However, function results will be cached similar to the other ICU routines until the server restarts, if possible.

- `I18N::getLocale()`
  This function returns the `I18N` object of your application’s locale.

- `I18N::getInternalLocale()`
  This function returns the `I18N` object of FIM’s internal locale. Generally, there should not be any need to call this function.

- `I18N::setLocale(I18N|string $locale)`
  The application’s locale is set to the object given as parameter. If the parameter is a string, a new `I18N` object is created that represents the given locale. The return value will be the new locale object or `false` if the process failed.

- `I18N::detectBrowserLocale(array $acceptedLocales = null, $fallback = 'en')`
  By using the information available from the user’s browser, this function tries to return a locale string that fits best. Valid locale strings have to be passed as first parameter; if the parameter is omitted, all supported locales are taken into account. If no locale whatsoever fits, the fallback will be returned.

The methods of a `I18N` instance finally provide the abilities to localize:

- `I18N->get($key, array $args = [])`
  The entry `$key` is looked up in the current ResourceBundle. If `$key` is an array, FIM will handle the individual entries as keys of a table structure.

If the key does not exist, FIM will report an error and return the string representation of the key instead. By setting the configuration entry `'localeRawFallback'` to `true`, FIM will instead treat the key name as if it were the result of the lookup.
If the result is a string and `$args` is not empty, FIM will apply message formatting and return the formatted value. ICU's support for message placeholders is quite extensive. However, I have not found a complete documentation yet. `http://userguide.icu-project.org/formatparse/messages` might be a good place to start, as well as `http://icu-project.org/apiref/icu4c/classMessageFormat.html#details`.

- `I18N->getLocaleId()`, `I18N->getLanguageId()`, `I18N->getRegionId()`, `I18N->getScriptId()`
  Returns the respective identifiers of the current object.

- `I18N->getLocaleName()`, `I18N->getLanguageName()`, `I18N->getRegionName()`, `I18N->getScriptName()`, `I18N->getVariantName()`
  Returns the respective identifiers of the current object in a human-readable format.

- `I18N->formatSize($size, $digits = 0, $binary = true, $thinsp = false)`
  Returns a string that is composed by the `$size` parameter, but with appended SI suffixes. If `$size` is a double value, `$digits` defines the number of digits after the decimal dot. If `$binary` is true, the binary suffixes `kiB`, `MiB`, ... are used with $1024$ as divider instead of $1000$. If you want the number and its suffix separated by a thin space (Unicode `0x2009`) instead of a normal space, set the forth parameter to `true`.

- `I18N->formatNumber($number, $format = NumberFormatter::DECIMAL, $type = NumberFormatter::TYPE_DEFAULT)`
  Formats a number with the given settings. This function acts as a more comfortable wrapper around `NumberFormatter->formatNumber()`. The most interesting values for the second parameters should comprise `NumberFormatter::DECIMAL`, `NumberFormatter::PERCENT` and `NumberFormatter::CURRENCY`. Do not use `NumberFormatter::CURRENCY`; use the `I18N->formatCurrency` method instead.

- `I18N->formatCurrency($currency, $type = 'USD')`
  Formats a number in currency format using the given type as currency symbol. The type has to be in three-letter ISO 4217 currency code format.

- `I18N->formatDate($timestamp, $format = IntlDateFormatter::MEDIUM)`,
  `I18N->formatTime($timestamp, $format = IntlDateFormatter::MEDIUM)` and
  `I18N->formatDateTime($timestamp, $dateFormat = IntlDateFormatter::SHORT, $timeFormat = IntlDateFormatter::MEDIUM)`

Those functions are responsible for formatting UNIX timestamps to human-readable form. Examples for all possible parameter values are given in the doccomments.

- `I18N->translatePath($path)`

  Sometimes, raw data needs to be localized. The ResourceBundle format allows to include this data in its files, but it is also possible to create a directory/file structure that is localizable. If your localized images are located in `/img/<locale identifier>/...`, simply pass the name of a file to this function using the short placeholder `<L>`: `$i18n->translatePath('/img/<L>/logo.png')` will search for the file `logo.png` in a subdirectory of `/img` that matches the current locale as good as possible. `false` is returned if the file cannot be found using fallback mechanisms as ICU does (so fallback to the system's locale and then to `root`).

- `I18N->parse...`

  Those functions are the counterparts of the `format...` functions. They will do the reverse job and convert human-readable data to machine-readable.

- `I18N->getTimezone()`, `I18N->setTimezone(IntlTimeZone $timeZone)`

  These functions get or change the current time zone. Time zones will affect the output of formatting and parsing timestamp functions. Note that at least PHP 5.5 is required for a support of those functions as the `IntlTimeZone` class is not available before.

- `I18N->getCalendar()`, `I18N->setCalendar(IntlCalendar $calendar)`

  As before, these functions get or change the current calendar which also affects the output of formatting and parsing timestamp functions. Once again, PHP 5.5 is required.

## 3.9 Log

The log class allows access to FIM's three log files. The only function that should be called in your application is `Log::reportCustomError($message, $noMail = false)`. It writes the first parameter to the custom log and sends a mail notification to the address specified in the configuration, if the second parameter is not set to `true`.
The other functions are not intended to be used by your application. The reporting functions will all do the same, just writing to different logs. The handler functions are only public as PHP requires this and are called manually not even by FIM.

## 3.10 Memcached

FIM provides several ways to emulate the extension `Memcached`. The wrapper around the old extension `Memcache` will provide an interface to you that is designed exactly as the one of

the newer extension. If neither one of those is installed, FIM will provide a standalone client that directly communicates with Memcached by using socket functions and once again has the same behavior as the one of the new `Memcached` extension.
Note that those replacements are only intended for development use. You should not make use of them in a production system. Some features are only pretended: Connections to multiple servers won't work. While both replacements won't throw an error, they will simply connect to the weightiest server (if the wrapper is used and a "new" version of the old extension is installed, multiple servers work). SASL authentication will always throw an exception. Most parts of the configuration simply don't change the behavior. So these replacements are powerful in lots of ways, but not that sophisticated. Note that the standalone client requires newer versions of Memcached that support the binary protocol.

## 3.11 Module

This is the base class for all modules. Most of its methods are only available to the module object itself. Assigning a variable to the module object will make this variable available in Smarty templates.

- `public function execute(...)`

  This function has to be implemented in modules if they are executable. It will be called with autoboxing.

- `public function handleError($errno)`

  This function can be implemented in modules that perform error handling.

- `public function handleError...()`

  These functions can be implemented in modules that perform error handling of the error number ....

- `Module->getModulePath()`

  This function returns a FIM normalized path of the folder the module file is stored in.

- `$this->fetchTemplate($templateFile)`

  This function returns the output of the template that is stored in `$templateFile`.

- `$this->displayTemplate($templateFile)`

  The output of the template `$templateFile` is sent to the output buffer.

- `$this->setContentType($newMIMEType, $newCharset = null)`

  This is a shortcut that allows to set the content type easier than using `Response::set`. The

first parameter can either be a string or an array. If it is an array, content negotiation will be applied to it (see the section in the reference of the `Response` class). If it is a string, a valid MIME type or a file name with extension is expected.
If no charset is given, the default value from the configuration is taken or the previous charset if one was already passed as a content type header.

- `$this->displayFile($fileName, $checkAccess = 0, $requireSubdirectory = null)`
  The file that is referred to in the FIM path `$fileName` is outputted as if it was requested by the user.
  It is possible to check the rules before performing the task. For this purpose, the second parameter may be a bitmask composed of `Rules::CHECK_EXISTENCE` and `Rules::CHECK_READING`. If rules denied access, an error is raised.
  If the third parameter is set, a 404 error will also be triggered when the file is not located within the directory that is given there.

- `$this->error($errno)`
  Triggers an error that can be handled by error handling routines of modules. Execution is aborted after this function.

- `$this->getErrorDetails()`
  Returns information about where the last error that was triggered *manually* occurred (see the doccomment for information about the return type). If no error occurred or the last error was caused by FIM (rules etc.), the return value will be `null`.

- `$this->forward($to, array $parameters = null, $keepTemplateVars = true, $checkRules = true)`
  This function expects a valid FIM path as first parameter. The requested resource will be executed as if is was the URL (but note the difference: Do not pass a URL as first parameter but a FIM path!). If the resource is a directory containing a module, you can change the current parameters with the second argument if you wish to.
  By default, all variables that were assigned to the current module will be kept in the new one, but you may erase them with the third parameter.
  The last parameter finally determines whether FIM should check the access rights (existence/reading/listing, whatever is necessary).
  Execution will stop after this function was called.

- `$this->redirect($to, array $parameters = [])`
  This function will perform an *external* redirect (so it's the user's browser that does it) to another resource which has to be given as valid FIM path as first parameter. GET parameters can be passed as second argument.

Execution will stop after this function was called.

If the FIM path could not be converted to a URL, an exception is raised.

## 3.12 PrimaryTable

This file provides the base class `PrimaryTable` as well as data helper utilities. The class `PrimaryTable` should be used as a superclass for every table that contains at least one primary key.

The extending class is required to define the protected static property `$columns`, which has to be initialized as an associative array. The keys correspond to the column names, the values indicate the data type. Valid types are `'string'`, `'int'`, `'double'`, `'bool'`, `'blob'` (which will be represented as a string, but has different internal handling) or a class name. This class has to implement either the `DataHelper` or the `fimSerializable` interface. Any of these types have to be prepended either with + if they are auto-increment, * if they are primary keys or 0 if `null` is allowed (not for data helpers).

Data helpers are classes that implement the `DataHelper` interface or extend the base class `DataHelperPrimitive`. The column will be represented as an object of this class, although it is stored in a different format in the database (thus e.g. allowing an easier way to handle bitmasks or similar data). The data helper will be "bound" to a certain object with the function `DataHelper->bind(PrimaryTable $row, $fieldName, $value, callable $change)`. It must report any change to its value by invoking the callable `$change` (which does not expect any parameter). The connection between data helper and table entry has to be released when `DataHelper->unbind()` is called. Finally, the function `DataHelper->getStorageValue()` has to return the database representation of the current value.

The base class `DataHelperPrimitive` implements all those methods. It thus simplifies the process of creating a data helper. When extending this class, you don't need to deal with the communication aspects except from calling the protected function `$this->setValue($value)` whenever the (already provided) property `$this->value` changes.

A primary table itself provides lots of methods (but note that a `PrimaryTable extends Table`, so also see the reference of `Table`):

- The necessary methods for being FIM-serialized and unboxed are supplied. Unboxing requires the existence of one auto-increment or only one primary key which is used as identifier. The `PrimaryTable::unbox($key)` method can however be overwritten, if more specific features are required.

- `$this->makeDBRepresentation()`

  Table entries can be marked as "virtual", i.e. they do not exist in the database but only in the application. This function will add the current, virtual table entry to the database and change its state accordingly.

- `$this->delete()`

  Removes the current object from the database and unbinds all existing data helpers.

- `self::createNew(...$columns)`

  This function has to be called from a subclass. It expects the values for all columns in the order as given in the `self::$columns` property, excluding a possible auto-increment value.
  As last argument, `true` might be given. This will make the entry "virtual", so that it is not added to the database.

- `self::translateStatement(PDOStatement $result)`

  This function converts the results from a database statement into objects of the current class.

- `self::findBy(array $where = [], $orderBy = null)`

  Returns an array of objects of this class that fulfill given conditions. The first parameter expects an associative array that assigns an exact value to a column. The second parameter can be any valid SQL string that shall follow the `ORDER BY` keyword.

- `self::getBy($where = null, array $bind = null, $order_by = null, $group_by_having = null, $limit = null, $join = '')`

  For more complex queries, use this function. Here you can specify the `WHERE` condition as a string. You may make use of placeholders which can then be bound by using the second argument. The remaining parameters are string as well that will be inserted directly into the SQL string.

- `self::findOneBy(array $where, $orderBy = null)` and
  `self::getOneBy($where = null, array $bind = null, $order_by = null, $group_by_having = null, $limit = null, $join = '')`

  These functions behave like their more general counterparts, but return only the first object or `null` if there were no results.

## 3.13 ReflectionFile

The class `ReflectionFile` is required by FIM to analyze your application's code and generate appropriate autoboxing cache files. But as the functionality of this might be of some use for you, you may use the class `ReflectionFile` to apply PHP's reflection to a whole file. Credit for the implementation goes to Zend Framework.

## 3.14 Request

This class provides several methods that unify the access of request data.

- `Request::get($param, $default = null)`
  Returns the entry `$param` from the GET parameters.

- `Request::post($param, $default = null)`
  Returns the entry `$param` from the POST parameters.

- `Request::boolGet($param, $default = false)`
  Returns the entry `$param` from the GET parameters as a boolean; lots of different ways of submitting a boolean value are recognized.

- `Request::boolPost($param, $default = false)`
  Returns the entry `$param` from the POST parameters as a boolean; lots of different ways of submitting a boolean value are recognized.

- `Request::hasGet($param)`
  Returns whether a certain GET parameter exists.

- `Request::hasPost($param)`
  Returns whether a certain POST parameter exists.

- `Request::hasFile($param, $checkSuccess = false)`
  Returns whether a certain file was present; if the second parameter is true, FIM additionally checks whether the upload was successful.

- `Request::itemize($filter = null, $post = true, $iterator = false, $matchFlag = RegexIterator::MATCH)`
  Returns an array or iterator (set the third parameter appropriately) of all the data that was submitted via POST or GET (depending on the second parameter). Additionally, a regular expression can be applied to the data, so that only certain keys are returned by using the first parameter.

- `Request::getFileError($param)`
  Returns the error that occurred when the file identified by `$param` was tried to be uploaded. The return value will be one of PHP's `UPLOAD_ERR_*` constants.

- `Request::saveFileUpload($param, $to)`

  Stores a file upload to a given location, `$to` being a valid FIM filepath.

- `Request::getFileContent($param)`

  Retrieves the content of a file as a string value.

- `Request::getFileResource($param)`

  Retrieves a valid `fopen` handle to a given file. As this is a temporary file it will be opened with `'rb'` privileges.

- `Request::getFileName($param, $default = null, $encodeName = true)`

  Retrieves the user-defined name of a file upload; with the third parameter, it is possible to automatically encode the file name into a proper code page name.

- `Request::getFileSize($param, $default = null)`

  Retrieves the file size of a file upload in bytes

- `Request::getFileType($param, $default = null)`

  Retrieves the MIME type of a file upload. Warning: This value is not reliable as it is set by the client and not validated internally!

- `Request::getFileTemporaryName($param, $default = null)`

  Retrieves the temporary name of a file upload.

- `Request::getFileDetails($param, $encodeName = true)`

  Retrieves an associative array holding all details regarding a specific file upload. Use this function instead of calling `getFileError`, `getFileName`, `getFileSize` or `getFileTypes` one after the other.

- `Request::getURL($parameters = false)`

  Gets the current URL that was requested. This does not include subdomains or the server name and no parameters as well. BaseDir will be truncated if necessary. The URL will start with a slash.
  If the first parameter is set to `true`, GET parameters will be included as well.

- `Request::getParameterString()`

  Gets the current parameter string that is appended to the URL, starting with the delimiter character. An empty string if there were no parameters.

- `Request::isHTTPS()`

  Determines whether the request was sent via HTTPS.

- `Request::isPost()`

  Determines whether the request method was POST.

- `Request::isGet()`

  Determines whether the request method was GET.

- `Request::postSizeExceeded()`

  Determines whether a POST request was started but the maximum POST size was exceeded. This can happen when there were too large file uploads and it means that no POST variable is available.

- `Request::saveURL()`

  Saves the current url so that it can be restored in further processing. As this will be saved for new requests until it is restored, this function can e.g. be used to redirect to an inaccessible page after login.

- `Request::restoreURL()`

  Performs a redirect to the url that was saved with ::saveURL(). Execution will not be halted (unlike module forward or redirect functions)!

- `Request::getBrowser()`

  Determines the browser the user used according to the user agent. For details see the doccomment.

- `Request::getPort($insertable = false)`

  Returns the port that is used for the current request. The parameter specifies whether a string, starting with a colon (or an empty string, if the default port is used), or an integer shall be returned.

- `Request::getFullURL($includePath = true)`

  Returns the full URL with which the request was started, including protocol, host and path.
  Set the first parameter to false in order only to get the host. The URL will not contain a trailing slash.

- `Request::getMaxUploadSize()`

  Gets the maximum number of bytes that can be uploaded. This is the minimum value of `php.ini`'s `upload_max_filesize`, `post_max_size` and `memory_limit`.

## 3.15 Response

The `Response` class provides several means to control the output to the user.

- `Response::$responseText`
  This variable contains the whole output. FIM activates output buffering; after your application finished, anything that was `echo`ed before will be appended to this variable. It is therefore possible to *pre*pend something to the output at any stage of code execution by setting this variable directly instead of using `echo` or `print`.

- `Response::$responseCode = 200`
  Set this to change the http response code header. Default is 200 (OK).

- `Response::getMIMEType($filename)`
  Determines the MIME type based on the filename's extension. FIM has built-in support for about one thousand extensions. If you notice any problems with this function, please contribute your MIME type!

- `Response::translateHTTPCode($code)`
  Transforms the given HTTP status code to its textual meaning. Hint: This function is not localized and will return the official string representations of the codes!

- `Response::headers(array $new = null)`
  Lists all headers that will be sent or replaces them altogether. By specifying an associative array as parameter, *all* the current headers will be removed and replaced by the new ones!

- `Response::get($name, $default = null)`
  Gets a specific header

- `Response::has($name)`
  Checks whether a specific header exists

- `Response::set($name, $value, $overwrite = true)`
  Sets a header field. `$name` and `$value` will be normalized.
  In order to set a `Content-Disposition` file name, set the content disposition field and do not use any quotes or escaping for the filename. FIM will escape it properly for any browser that is used.

- `Response::delete($name)`
  Deletes a specific header field.

- `Response::expire($time)`
  Sets an `Expires` header in the request. If this function is not called, the default expiring time will be one month.
  Module pages and parsed files will expire at once by default. Response codes that differ from 200 (OK) will expire also at once.

- `Response::cache($expires, $mustRevalidate)`
  Deals with the caching mechanism. Sends an `Expires` header and `must-revalidate` and its HTTP/1.0 equivalents.

- `Response::contentNegotiation(array $contentTypes, $charset = null)`
  Sets a content type based on the `Accept` header of the client.
  The first parameter has to contain acceptable content types. Preferred ones come first. You *have to* give one value that has the key `'*'`; this content type will be set if no other type matches the `Accept` header. If you forget to do so, an exception will be thrown.

- `Response::doSend()`
  This function sends all headers that were specified before and then outputs the response text.
  Do not call this function manually unless you use this class in the subdomain base error script.

- `Response::mail($to, $subject, $message, $from = null, $html = false, array $attachments = [])`
  This functions sends an e-mail. It uses the PHPMailer script that is delivered with FIM for this purpose. It is possible to use PHPMailer on your own if you need to make use of its more advanced capabilities than the ones offered in this function.
  See the documentation of PHPMailer for an insight in its structure. FIM will automatically make the class `PHPMailer` available; the default `From` header will be set to the configuration setting `'mailFrom'`

## 3.16 Router

The `Router` class contains lots of static methods that are necessary to deal with path mappings (URL $\leftrightarrow$ FIM path $\leftrightarrow$ file system path). It can however be extended and subclasses will be instantiated in order to perform routings, as they are explained in subsection 2.8.

- `Router::normalizeFIM($path, $returnArray = false)`
  Normalizes a path in FIM style. The returned path will start with two slashes and will not end with a slash (if it is a string, as can be controlled with the second parameter). Note that the result of this function might depend on the current working directory.

- `Router::normalizeFilesystem($path, $returnArray = false)`
  Normalizes a path in filesystem style. The returned path will start with a slash or drive letter and will not end with a slash. This function understands the two protocols `fim://` and `file://`, if given.

- `Router::convertFIMToFilesystem($fimPath, $normalize = true)`
  Converts a FIM path to an absolute path.

- `Router::convertFilesystemToFIM($filesystemPath, $normalize = true)`
  Converts a filesystem path to a FIM path, if possible (else `false` is returned).

- `Router::mapURLToPath($url, &$parameters, &$noRouting = null, &$failureAt = null)`
  Converts a given URL to a normalized path respecting given routings. The third parameter will be set to `true` if no routing will ever occur to the given URL, regardless of its parameters. The forth parameter will be set to the last directory that could be mapped successfully and thus allows to see at which level the routing process failed.

- `Router::mapPathToURL($path, array $parameters = [], &$noRouting = null, $allServers = false)`
  Converts a given path with parameters to a URL. The last parameter controls whether an array is returned with the first entry being the part before the server string, the second one the part after the server part. A string is returned when the last parameter is `false` or if the URL would be the same, regardless of server settings. This is only important if you allowed multiple entries in the `'subdomainBase'` settings.

- `protected abstract rewriteURL(array $url, array &$parameters, &$stopRewriting)` and
  `protected abstract rewritePath(array $path, array &$parameters, &$stopRewriting)`
  These functions have to be implemented by subclasses. They are explained in subsection 2.8 and in the doccomments.

## 3.17 Rules

The `Rules` class is base class for all rules that control resource access. Setting any property to the rules object will assign it to Smarty templates. The class contains static methods that apply the rules:

- `Rules::check($fileName, $checkFor)`
  Checks whether a bitmask of rights can be applied to a certain file or folder. The result will be cached. The second parameter is composed of the class constants `Rules::CHECK_EXISTENCE`, `Rules::CHECK_READING` and `Rules::CHECK_LISTING`. The shortcut value `Rules::CHECK_ACCESS` is a composition of all those three.

- `Rules::checkFilterExistence($fileName)`

  Checks whether for a specific location a rule exists.

  This function only checks for the existence of `fim.rules` files, not what these files will do.

- `Rules::clearCache($checkCache = Rules::CHECK_ACCESS)`

  Clears the cache for checked rules. Calling this might be necessary after an update of login credentials or similar operators.

- `$this->checkSimpleAccess($fileName, $checkFor)`

  Checks the recommendation that the simple rules file in the same folder as the rules script gives for a certain file name.

- `public function checkExistence($fileName, $fullName)`,

  `public function checkReading($fileName, $fullName)` and

  `public function checkListing($fileName, $fullName)`

  Those functions can be overwritten in subclasses and shall return values as described in subsection 2.11. The default implementation will do nothing and return `null`.

## 3.18 Semaphore

FIM provides a class that allows synchronization. This class can be used in two modes: The first one will act as a process-only semaphore, locks will only apply to the current process. The second one can span multiple servers on multiple computers and is thus able to synchronize a whole data center. While the first option will work best with one of the extensions `Semaphore`, `Wincache`, `APC`, `XCache`, `Redis` or `Memcached`, the unavailability of all these will not break its functionality. The second option however requires Redis or Memcached to be set up properly.

- `Semaphore::__construct($name, $multiServer = false)`

  Creates a new object. Semaphores are identified by their name.

- `Semaphore->__destruct()`

  A semaphore is unlocked automatically when its object goes out of scope.

- `Semaphore->lock()`

  Acquires an exclusive lock. This function will return `true` if the lock was successfully. Calling this function will make any other Semaphore object of this name that tries to acquire the lock wait until it is released.

- `Semaphore->unlock()`

  Releases the lock. If the result is `false`, a problem occurred.

## 3.19 Serialization

PHP's method `unserialize` has a serious drawback: it automatically creates the object that shall be unserialized. This instance will then be refilled with the data. FIM needs the capability to unserialize singletons, so the functions in this file work as a replacement for PHP's serialization methods.
The interface `fimSerializable` has to be implemented by every class that can be serialized. Its `fimSerialize()` method has to return a string that contains the serialized content of the object; the *static* method `fimUnserialize($serialized)` returns the unserialized object and is responsible for its creation as well.
The global methods `fimSerialize($arg)` and `fimUnserialize($serialized)` then work exactly as their PHP counterparts.

## 3.20 Session

The `Session` class provides several static methods that allow to preserve data between different requests. Note that in command line mode, none of these methods will work.

- `Session::$memcached` and `Session::$redis`

  Holds instances of `Memcached` and `Redis` or `Predis`, if you made the appropriate settings.

- `Session::get($name, $default = null)`

  Gets a session variable

- `Session::has($name)`

  Checks whether a specific value is stored within a session.

- `Session::set($name, $value, $overwrite = true)`

  Sets a session variable.

- `Session::delete($name)`

  Removes a session variable.

- `Session::clear($onlyStatic = true)`

  Removes all the session variables. The parameter indicates whether flash variables shall also be removed.

- `Session::getFlash($name, $default = null, $extend = false)`

  Gets a temporary session variable. Those are only available for the very next module invocation (excluding internal forwards) and deleted afterwards.

- `Session::hasFlash($name)`

  Checks whether a specific temporary value is stored within a session.

- `Session::setFlash($name, $value, $overwrite = true)`

  Sets a temporary session variable that will only be accessible within the very next request.

- `Session::deleteFlash($name)`

  Removes a temporary session variable.

- `Session::extendFlash($name)`

  Extends the lifetime of a temporary session variable so it will stay one more request alive.

- `Session::extendFlashs()`

  Extends the lifetime of all temporary session variables.

- `Session::getConst($name, $default = null)`

  Gets a FIM constant. Those are values that won't be stored for the next request, but they are stored for the current execution and can be thought of an alternative to PHP's constants. The advantage of FIM's "constants" is that they are variable.

- `Session::hasConst($name)`

  Checks whether a specific constant is known to FIM.

- `Session::setConst($name, $value, $overwrite = true)`

  Sets a FIM constant.

- `Session::deleteConst($name)`

  Removes a FIM constant.

- `Session::getId()`

  Returns the current session id.

- `Session::renewId()`

  Renews the session id with respect to the session transition. Renewal is not possible when the current session is a transition session which was already replaced by another one (see configuration reference).

- `Session::replaceId($newId)`

  Replaces the current session id with a new one. The session's lifetime will be set appropriately. It will not be possible to access the session under its old id.

- `Session::setLifetime($lifetime = 0)`

  Sets the lifetime of the session cookie used by the framework. A lifetime of zero means that the cookie expires when the browser window closes (default). All other values are in seconds, calculated from now. This does not have anything to do with the session configuration.

## 3.21 Smarty

This file provides the template functions `{url}`, `{lurl}` and `{L}` which are described in detail in the explanations of Listing 2.3 and Listing 2.9.

## 3.22 Table

The base class for all table-like structures that do not necessarily contain primary keys is `Table`. By extending this class, subclasses will have a protected variable `$fields` available that contains the data of the table entry. By setting or getting properties of the object, the contents of this variable are changed.
A setter for the individual fields may be defined by creating a method names `protected function set...($value)` where `...` is the fields name. The setter has to return `false` if FIM shall not set the field automatically to the new value.
The protected function `singleton($key, $action = 0)` can be used to manage singletons: Tables shall generally only be allowed to have one instance per row in a table; otherwise, data gets inconsistent. Therefore, this function can be called with the first parameter as a unique identifier of the current row. The second parameter may be `0` (return the instance connected to this key), `null` (remove this key's instance) or an object to store the instance.
The function `count()` will return the number of rows of the current table.
Any table class needs quick access to its details which are the same for each object of a particular table class. For this purpose, a static storage is created. This storage is only filled once with all the necessary data (the `Table` superclass will automatically determine the table name by using the class name) and can then be retrieved by calling `$this->getStaticStorage($singleValue = null)`. The function will return the whole static storage array if `$singleValue` is `null` or the concrete entry with the key given as first parameter.

# 4 Final remarks

## 4.1 Requirements

FIM requires at least PHP 5.4, 5.5 is recommended for full functionality. As PHP 5.6 has just left development state, FIM does not make use of its new features yet, although this is planned to be integrated.
The server software has to support URL rewriting so that any request is forwarded to the front controller. Configuration examples for Apache HTTPD or nginx can be found on page 5 or page 50.
PHP has to provide the modules PDO (for database access, not needed if not desired), Intl (for internationalization, required). On Windows, either cURL, COM or `exec` have to be available in order to use `fileUtils`.
In development, usage of XDebug is recommend while one of the PHP caches APC, XCache, Wincache, Zend cache, Opcache or similar engines should be used in production systems. Those caches must be required not to remove PHPDoc from the source!
When Memcached shall be used, the extension Memcached is highly recommended (though FIM will also work with Memcache and without any of these two).

## 4.2 Acknowledgments

FIM makes use of several third-party libraries or parts of them. Alphabetically ordered:

- array_column
  FIM will automatically make `array_column` available for PHP 5.4 by using this library.

- BigFileTools
  The idea of `fileUtils::size($fileName)` comes from BigFileTools, though the implementation differes a bit.

- PHPLint IO Library
  Important parts of the encoding functions in `fileUtils` come from Umberto Salsi, icosaedro.it

- PHP Mailer
  FIM uses PHPMailer as internal mail delivery system. PHPMailer is bundled with FIM and can be accessed directly.

- Smarty
  FIM uses Smarty 3 as template engine.

- Zend Framework 2
  The class `ReflectionFile` comes from ZF 2 with only subtle changes.
- Other smaller snippets are marked directly in the source.

## 4.3 Contribute

You may contribute to **F**ramework **Im**proved on GitHub by reporting—or fixing—bug or localizing FIM in other languages. And you can of course donate.