Yellow Duck Framework 2.0
User's Guide

Written by Pieter Claerhout, pieter@yellowduck.be
Version 2.0.0, june 2004

About this user's guide

This user's guide brings together in one document everything needed for the day-to-day use of version 2.0.0 of the Yellow Duck Framework.

The Yellow Duck framework is web application framework created by Pieter Claerhout. More information can be found on http://www.yellowduck.be/ydf2/.

Author: Pieter Claerhout
Email: pieter@yellowduck.be
Version: 2.0.0, june 2004

Table of contents

1 Introduction

1.1 Overview

The Yellow Duck Framework takes care of all the difficult work you normally have to perform manually when developing a web application. It is based on the idea of requests that can performactions. By encapsulating all the programming in an object-oriented environment, you get a framework that is easy to use and understand, easy to extend and doesn't limit you in any way.

The Yellow Duck Framework supports the following items:

- Clean separation of code and output
- Templates for outputting HTML easily
- Automatic action dispatching using URL parameters
- Object oriented formconstruction and validation
- Object oriented handling of authentication
- Classes for creating "XML/RPC" clients and servers.
- Classes for creating syndicated XML feeds such as RSS and Atom feeds.
- Easy handling of files, directories and images. For images, there are some very straightforward functions that can create thumbnails of those images.
- An object oriented interface for creating and sending email messages.

The Yellow Duck Framework tries to be as flexible as possible so that you can tailor it in such a way that it works according to the way you want it to work. It's definitely not the framework that will solve all your needs, but for most web application related functions, you will find the Yellow Duck Framework a very handig tool to get your work done faster and more reliably.

As you can see, no database connectivity is included in the framework.We did this for some very specific reasons. The framework was tested with the native database functions from PHP and also with both the PEAR and ADOdb database libraries. All these libraries can be used in theYellow Duck Framework without any problems.We didn't want to link the framework to one specific database library in order to give you the option to choose which one suits your needs the best.

1.2 Analyzing the workflow

In the object oriented nature of the Yellow Duck Framework, each script that gets executed is based on a comman class, which is the YDRequest class. This class is smart enough to figure out which functions needs to be called using parameters given in the url and also supports some advanced functions such as authentication handling.

Let’s take a look at an example script to understand how this works:

```
<?php

  require_once( dirname( __FILE__ ) . '/YDFramework2/YDF2_init.php' );

  require_once( 'YDRequest.php' );

  class sample1Request extends YDRequest {

    function sample1Request() {
      $this->YDRequest();
    }

    function actionDefault() {
      $this->setVar( 'title', 'sample1Request::actionDefault' );
      $this->outputTemplate();
    }

    function actionEdit() {
      $this->setVar( 'title', 'sample1Request::actionEdit' );
      $this->outputTemplate();
    }

  }

  require_once( dirname( __FILE__ ) . '/YDFramework2/YDF2_process.php' );

?>
```

The template file that goes along with this script looks as follows:

```
<html>

<head>
  <title><?= $title ?></title>
</head>

<body>
  <?= $title ?>
</body>

</html>
```

When you request this script in your web browser, a number of things happen. Everything that happens is done automatically by the workflow itself and doesn’t require any manual intervention from your part. In a future chapter, you will see that this is not the only way of working with the Yellow Duck Framework. Every single part of the Yellow Duck Framework can be changed to work exactly the way you want it to work.

In the Yellow Duck Framework, all requests are always processed in the same way and order. Let’s evaluate the sample above step by step to explain how the processing is happening.

The first line you see in the sample1.php script is the include of the file called "YDF2_init.php". This file is responsible for setting up the Yellow Duck Framework, and does things such as:

- Defining a number of constants with e.g. the path to specific directories and URLs
- Starts or restores the previous session
- Reconfigures the PHP include path
- Includes the different files from the rest of the framework

After that,we define a new class, called "sample1Request" which is based on the YDRequest class. For each script, you need to have 1class which is named as the basename of the file (sample1in this case) and is appended with the string "Request". This class should have the YDRequest class as one of it ancestors.

Since we are inheriting from the YDRequest class, we initialize the parent class in the class constructor of the sample1Request class.

Then we see two functions that start with the text "action". All functions that implement actions start with "action".We will see later on how you can choose which one gets executed.

The last part of the script is the include of the "YDF2_process.php" script, which processes the actual request. It will look for a request class based on the name of the file, and will execute the process function of that class. This is where the magic happens.

The process function checks the URL to see if there was a parameter defined with the name "do". This parameter can indicate the action that needs to be executed. If the url looked as http://localhost/sample1.php?do=edit, the function called "actionEdit" will be executed.

If no action is specified in the url, the default action, called "actionDefault" will be executed. In both actions, the functions in this script don’t do a lot. They will set a template variable called "title" and they will output the template to the browser. The template contains special tags that are filled in on the fly with the right contents.

1.3 More about this user guide

Now that we discussed the basics of the Yellow Duck Framework and also explained the ideas about it, we will go over each part in more detail. In the following chapters, all the different parts of the Yellow Duck Framework will be discussed.

For more information about the Yellow Duck Framework,we like to guide you to the website of the framework which can be found on http://www.yellowduck.be/ydf2/. On this website, you will find more information as well as the latest downloads for the framework.

2 Installing the Yellow Duck Framework

This chapter will explain you how the Yellow Duck Framework can be installed. It will give you a checklist style overview of all the things you need to do to install and configure the Yellow Duck Framework.

2.1 Prerequisites

To use the Yellow Duck Framework, you need to have the following prerequisites:

- PHP version 4.2 or newer (tested with PHP 4.3)
- Webserver capable of running PHP scripts, such as Apache or Microsoft IIS

The Yellow Duck Framework was primarily tested on the Windows platformand also has undergone extensive testing on Linux based systems. Currently, the Yellow Duck Framework is only supported with PHP version 4. Support for PHP version 5 will be added in the future.

2.2 Examining the Yellow Duck Framework files

When you downloaded the latest release of the Yellow Duck Framework, you need to decompress it unzip e.g. winzip (Winzip) or the tar command (unix and linux). After decompressing, you will have the following directory structure:

```
+- YDFramework-2.0.0
   +- index.php
   +- index.tpl
   +- …
   +- YDFramework2
      +- 3rdparty
      +- docs
      +- temp
    +- YDClasses
```

The main directory of the framework, called "YDFramework-2.0.0" in this example, contains a number of sample files. In this directory, there is also a directory called "YDFramework2", which contains the actual framework files. There are a number of subdirectories in the YDFramework2 directory which each have a specific function:

- **3rdparty**: This directory contains the third party libraries that are needed for the Yellow Duck Framework to work properly. You will find a local copy of the PEAR libraries and the other third party libraries in here.
- **docs**: This directory contains the documentation at which you are currently looking. It also contains the complete API documentation.
- **temp**: This directory contains the temporary files created by the framework. In here, the cache files for the thumbnails and web downloads will be saved.
- **YDClasses**: In here, you will find all the classes that make up the framework.

This directory structure is not really important for using the Yellow Duck Framework. The only thing you might need to do from time to time is to check the temp folder and empty it if it gets too big. You can safely delete it contents without affecting the actual Framework.

***Note**: You will never have to add or alter files in the YDFramework2 directory. It's not even a good idea to put your own files in there as they might get overridden when you upgrade your framework to a newer version.*

2.3 Installation overview

There are a number of steps we need to do to get the framework installed properly. The install instructions here apply to the Apache webserver, but similar techniques are available on other servers.

These are the steps needed to install the framework:
1. Finding a place for the YDFramework2 directory
2. Assigning rights to the YDFramework2 directory
3. Configuring Apache to allow .htaccess files
4. Configuring the PHP options
5. Denying direct access to the templates
6. Using auto prepend and auto append
7. Configuring the samples
8. Testing the installation

The next sections describe these steps in detail.

2.4 Finding a place for the YDFramework2 directory

The YDFramework2 directory can be placed anywhere in the file system.For security reasons,we suggest you to put the YDFramework directory in a directory which is not viewable by the webserver. It's not a good idea to put the YDFramework2 directory into the htdocs folders from Apache.

The YDFramework2 directory can also be shared among multiple web applications. You only need one YDFramework2 directory on your system. You can if you want install a separate copy for each web application.

***Note**: If you plan to share the Yellow Duck Framework among multiple web applications, you need to think about how you will handle the temporary data from the Framework. A good idea is to write a sheduled task or a cron job that clears the temporary directory every hour to keep the size of this directory within reasonable sizes.*

2.5 Assigning rights to the YDFramework2 directory

Since the framework needs to be able to write temporary data into it's temp directory,we need to change the rights for this folder. On Windows, you normally don't need to change this. On a unix or linux based system, you can issue the following command in a shell to do this:

```
/home/pieter # chmod 777 YDFramework2/temp
```

If you are uploading the framework using your FTP client, please check the documentation of your FTP client on how to do this.

2.6 Configuring Apache to allow .htaccess files

Note*: this is an optional setting and is already done on most systems. You only need to change this if you plan to use .htaccess files to change the PHP settings or if you want to deny access to the template files.*

In the Apache configuration file, you need to change the following for the web directory of your web application:

```
<Directory "C:/Program Files/Apache/htdocs">
  AllowOverride All
</Directory>
```

With configuring the directory like this in Apache, you indicate that .htaccess files can be used to override the settings.

2.7 Configuring the PHP options

Now that we configured Apache to accept .htaccess files, create a new file called ".htaccess" and save it in the root of your web application. The settings done in the .htaccess file apply to the directory in which the file is stored and to all the directories underneath that directory. I've added the following configuration values for PHP to the .htaccess file on my system:

```
# Disable magic quotes
php_value magic_quotes_gpc 0
php_value magic_quotes_runtime 0
php_value magic_quotes_sybase 0

# Disabled registering of globals and arg*
php_value register_globals 0
php_value register_argc_argv 0

# Disallow some security holes
php_value allow_call_time_pass_reference 0
php_value allow_url_fopen 1
php_flag short_open_tag On
php_flag enable_dl Off

# Gzip compress the output
php_flag output_buffering Off
php_flag zlib.output_compression On
```

With these options turned on, you will have Gzip compressed output from the PHP scripts which helps you save bandwith. Registering of global variables and friends are also turned off, and magic quotes are disabled.

In the installation download, there is a sample .htaccess file included which is called "_default.htaccess". Copy it to the root of your web directory and rename it to ".htaccess".

***Note**: In order to have the examples working, you need to allow short open tags. this can be done by add the following to the .htaccess file:*

```
php_flag short_open_tag On
```

This will enable the use of "<?" as the open tag for PHP scripts instead of the standard "<?php" tags.

2.8 Denying direct access to the templates

Since we do not want people to access the template files directly,we need to tell Apache to deny access to these files. This can be done by adding the following code to the .htaccess file:

```
# Denying direct access to the templates
<FilesMatch "(.tpl|config.php|includes)$">
  Order allow,deny
  Deny from all
</FilesMatch>
```

In the example above, I also denied access to the config.php file, as I do not want people to access this file directly. Access to the YDFramework2 and include directories are also denied.

2.9 Using auto prepend and auto append

Instead of having to include the "YDF2_init.php" and "YDF2_process.php" files manually in each script, you can use the auto prepend and auto append options provided by the PHP interpreter.

To enable this feature, add the following lines to the .htaccess file:

```
# Auto include the Framework files
php_value auto_prepend_file "C:/YDFramework2/YDF2_init.php"
php_value auto_append_file "C:/YDFramework2/YDF2_process.php"
```

On Windows, you will have to use forward slashes instead of backslashes. Also make sure you use the complete path to the files.

2.10 Testing the installation

To test the installation, run the sample scripts provided with the downloads.

3 Writing your first application

You can consider this chapter as a small tutorial for the Yellow Duck Framework. In this chapter, we will go over some of the very basic concepts of the framework and apply these in a sample application.

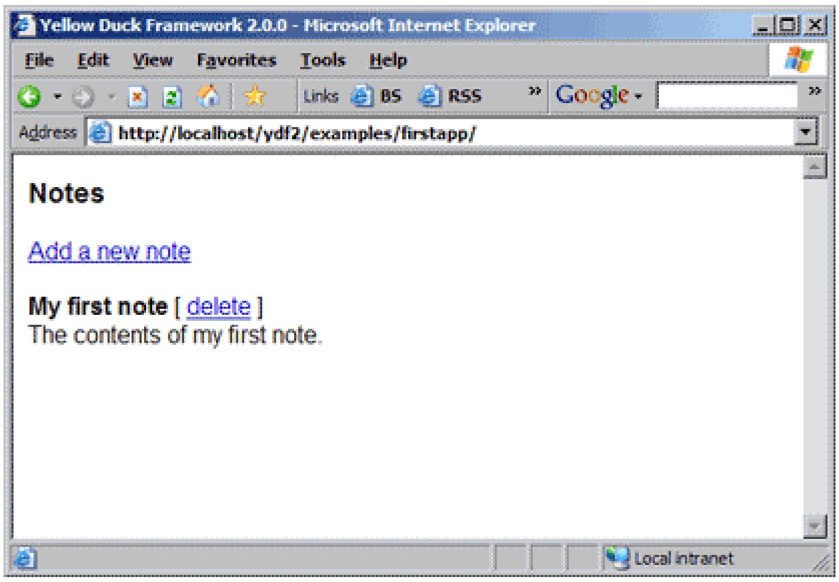
3.1 Description of the application

The sample application we are going to use is a simple notebook which has three options. The following options are supported by our notebook:

- Showing the list of notes (default)
- Adding a note
- Deleting a note

We will also use form validation to make sure that the data entered in the add note formis valid. This will prevent people from adding notes without a title or body.

After you finished this chapter, the finished example application will look like this:


Yellow Duck Framework 2.0.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links BS RSS
Google
Address http://localhost/ydf2/examples/firstapp/
Local intranet

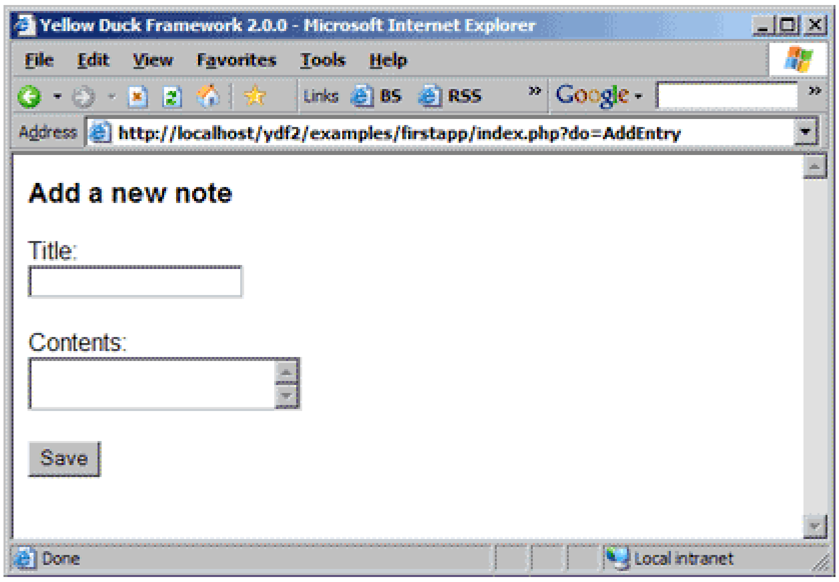
Notes

Add a new note

My first note [delete]
The contents of my first note.

The screen above is used for listing and deleting notes. This screen will also show you a link to the screen that is used for adding new notes.

The interface for adding a new note will look as follows:


Yellow Duck Framework 2.0.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links BS RSS Google
Address http://localhost/ydf2/examples/firstapp/index.php?do=AddEntry
Add a new note
Title:
Contents:
Save
Done
Local intranet

The delete screen doesn't really have a user interface. It will remove the item from the list and reload the list once that is done.

***Note**: In order no to over complicate this example, we will store the notes as files on the disk instead of using a database.*

3.2 Structuring the application

Before you start writing the actual code, you need to do a little bit of planning and determine which actions you will create to make your application work. In our case, we are going to make 3 different actions:

- **default**: This is the default action and will show the list of notes. If no notes are found, it should tell us so.
- **addnote**: This action will take care of adding a new note. It will render the formyou need to fill in, but it will also take care of handling the formcorrectly and saving the result to disk.
- **deletenote**: This function will delete a note based on it's unique ID. In this example, it will not ask you for confirmation, but it will simply delete the note.

Now that we know what actions the application will be able to perform,we now need to think about how we can store the notes on disk. We will create an associative array for each note which will contain the title, body and the unique ID for the entry.

The unique ID for an entry will be created by calculating the md5 checksum of the combination of the title and the body. This will ensure that each note is unique.

***Note**: the example will not warn about the fact that a duplicate entry was entered. It will just overwrite the existing one. I'll leave it up to you to implement that yourself after finishing this chapter.*

We will use the functions provided by the Yellow Duck Framework to load the notes from disk and save the notes to disk. In the framework, there are different objects and functions that take care of this.

After going through this example, you will have used and explored the following classes and modules in the Yellow Duck Framework:

- YDRequest (class)
- YDFSDirectory (class)
- YDFSFile (class)
- YDForm (class)
- YDError and YDFatalError (functions)
- YDObjectUtil (module)

For more detailed information of the functions and variables exposed by these classes, consult the API documentation.

3.3 Creating the directory and the files

To start, create a new folder in your webroot called "firstapp".We will store all the files related to this tutorial in that folder. Make sure that this folder is visible by your webserver.

In this folder,we will need to create two files for our application.We need 1file called "index.php", which will contain the actual script that drives the application (the logic).We will also need a file which is called "index.tpl" which contains the template for this application.The template will define how the application will be presented in the browser (the presentation).

By separating the actual script from the presentation,we will make the application a lot easier to maintain and understand. By using this structured way, it will be a lot easier to track down problems because you know immediately where to look.

3.4 Implementing the basis of the index.php file

The first thingwe will do is to implement the basic stuff of the index.php script.As you could read in the first chapter, the name of the file determines how the class should be named. Open the index.php file in a text editor and enter the following text in the file:

```
<?php

  // Initialize the Yellow Duck Framework
  require_once( dirname( __FILE__ ) . '/YDFramework2/YDF2_init.php' );

  // Includes
  require_once( 'YDRequest.php' );
  require_once( 'YDFSDirectory.php' );
  require_once( 'YDObjectUtil.php' );
  require_once( 'YDForm.php' );
  require_once( 'YDError.php' );
  // Class definition for the index request
  class indexRequest extends YDRequest {

    // Class constructor
    function indexRequest() {
      // Initialize the parent class
      $this->YDRequest();
    }

    // Default action
    function actionDefault() {
    }

    // Add Note action
    function actionAddNote() {
    }

    // Delete Note action
    function actionDeleteNote() {
    }

  }

  // Process the request
  require_once( dirname( __FILE__ ) . '/YDFramework2/YDF2_process.php' );

?>
```

There are a number of rules to follow to get the basis of the class implemented. Let's go over each one of them:

1 **Include the init file**: each script that wants to use the Yellow Duck Framework needs to include the "YDF2_init.php" file which initializes the framework.Make sure the path to this file is correct,otherwise, the script will not work.

2 **Include the needed classes**: The next step is to include all the different classes from the framework you are going to use. I used the require_once to make sure the script doesn't continue without having included these files. You don't need to specify the complete path to these files as the framework will take care of finding the right files.

3 **Define the main class**: each script that wants to use the Yellow Duck Framework needs to have a class which is named after the name of the file. In our example, the file is named index.php, which means the framework will search for a class called indexRequest. This class needs to extends the YDRequest class to allow the framework to process the request.

4 **Define the class constructor**: When the instance of our class is created, it will execute the function which has the same name as the class automatically (this function is called the class constructor). In the class constructor,we simply call the YDRequest function (which is the class constructor from the YDRequest class) to make sure the parent class is initialized as well.

5 **Define the functions for the actions**: For each action,we need to create a separate function in the class. Each function for an action has the name of the action prepended by action as it's function name. By default, these function do not require any arguments.

With this implemented, you can already surf to the index.php page, but nothing will be shown. You can try the following URLs:

- http://localhost/firstapp/index.php
- http://localhost/firstapp/index.php?do=addnote
- http://localhost/firstapp/index.php?do=deletenote
- http://localhost/firstapp/index.php?do=oops

If you typed in everything correctly, only the last URL should return an error because it's pointing to an undefined action in our class.

3.5 Improving the class constructor

We will add one thing to the class constructor, which is a reference to the data directory. Before you add the code, make a new folder called "data" in the "firstapp" folder. Also make sure that the webserver process can write into that directory. On unix or linux based systems, you can do this with the following shell command:

```
/home/pieter # chmod 777 data
```

Once you did that, add the following code to the class constructor:

```
// Set the path to the data directory
$this->dataDir = new YDFSDirectory( dirname( __FILE__ ) . '/data/' );

// Check if the data directory is writeable
if ( ! $this->dataDir->isWriteable() ) {
  new YDFatalError( 'Data directory must be writable!' );
}
```

With this code, we create a new YDFSDirectory class which represents a folder on disk.We define it specifically create it in the class constructor to ensure that all the actions are able to use this object (each action needs this).

We also check if the directory is writeable by the webserver process to ensure that we will be able to save the notes in that directory. If the directory is not writable,we will raise a fatal error which stops the execution and displays the error message.

Later on, we will see that we can use this object to get a directory listing, but we will also use it to delete and create new files.

3.6 Implementing the default action

We will now implement the default action. This is the function that is responsible for showing the list of entries.

Add the following code to the function actionDefault:

```
// Default action
function actionDefault() {

  // Start with an empty list of entries
  $entries = array();

  // Loop over the data directory contents
  Foreach ( $this->dataDir->getContents( '*.dat' ) as $entry ) {

    // Get the contents
    $entry = $entry->getContents();

    // Unserialize
    $entry = YDObjectUtil::unserialize( $entry );

    // Add it to the list of entries
    array_push( $entries, $entry );

  }

  // Add the entries to the template
  $this->setVar( 'entries', $entries );

  // Output the template
  $this->outputTemplate();

}
```

The code of this function is pretty self explanatory. Let's go over each line to see what it does:

```
$entries = array();
```

This line creates a new array which will use to store the entries in.

```
foreach( $this->dataDir->getContents( '*.dat' ) as $entry )
```

This line line will query the data directory and get a YDFSFile object for each file that ends with the extension "dat". The getContents function always returns objects.

```
$entry = $entry->getContents();
```

This line line will replace the variable $entry with the contents of our YDFSFile object.

```
$entry = YDObjectUtil::unserialize( $entry );
```

Since the entries are serialized,we need to unserialize them to get the original object back. The YDObjectUtil::unserialize function will take care of this.

```
array_push( $entries, $entry );
```

We now have the original object back, which we will just add to the list of entries.

```
$this->setVar( 'entries', $entries );
```

When a new YDRequest class is instantiated, automatically a new template object is created. You can then use the setVar function from the YDRequest class to assign variables to the template. We add a new template variable called "entries" which holds the list of entries.

```
$this->outputTemplate();
```

The last step is to parse the template and output it to the browser. Since we didn’t specify the name of the template, it will look for a file with the same name as the script, but which has the extension "tpl" instead of "php". It will parse the template and send the result to the browser.

If you run the script now in the browser, you will see an empty screen, and no errors should be shown. You don’t see anything yet since the template is still an empty file.

3.7 Implementing the template

Now that we have the default action implemented,we will change the template so that it shows the list of notes which it should do. Here’s is how the template looks like to show the list of entries:

```
<html>

<head>
  <title><?= $YD_FW_NAMEVERS ?></title>
</head>

<body>

  <?php if ( $YD_ACTION == 'default' ) { ?>

    <h3>Notes</h3>
    <p><a href="<?= $YD_SELF_SCRIPT ?>?do=AddNote">Add a new note</a></p>

    <?php if ( $entries ) { ?>
      <?php foreach ( $entries as $entry ) { ?>
        <p>
        <b><?= $entry['title'] ?></b>
        [ <a href="<?= $YD_SELF_SCRIPT ?>?do=DeleteNote&id=<?= $entry['id']
?>">delete</a> ]
        <br>
        <?= $entry['body'] ?>
        </p>
      <?php } ?>
    <?php } else { ?>
      <p>No notes were found.</p>
    <?php } ?>
  <?php } ?>

</body>

</html>
```

The first thing we do in the template is to check if we are running the default action. Since we are going to combine the templates for all the different actions,we need to make sure we only show the parts relevant for the current action.

As you can see, this is a plain PHP script which only contains the code needed to display the list. As you can see, we can reference the variables as normal PHP variables and show their contents.

We also use some special variables in the script that are automatically added to the template by the framework. We use the following ones:

- **$YD_FW_NAMEVERS:** the name and the version of the framework
- **$YD_ACTION:** the name of the current action (always in lowercase)
- **$YD_SELF_SCRIPT:** the url of the script itself without parameters

If you run the script now, it should tell you that no notes were found, as we didn't create any yet. There should also be a link that you can use to add a new entry. The next step is to create the form to add a new entry.

3.8 Implementing the addnote action

The next step is to implement the action that will take care of adding new items. Add the following code to the actionAddEntry function to do this:

```
// Add Note action
function actionAddNote() {

  // Create the add form
  $form = new YDForm( 'addEntryForm' );

  // Add the elements
  $form->addElement( 'text', 'title', 'Title:' );
  $form->addElement( 'textarea', 'body', 'Contents:' );
  $form->addElement( 'submit', 'cmdSubmit', 'Save' );

  // Apply filters
  $form->applyFilter( 'title', 'trim' );
  $form->applyFilter( 'body', 'trim' );

  // Add a rule
  $form->addRule( 'title', 'Title is required', 'required' );
  $form->addRule( 'body', 'Contents is required', 'required' );

  // Process the form
  if ( $form->validate() ) {

    // Save the entries in an array
    $entry = array(
      'id' => md5(
        $form->exportValue( 'title' ) . $form->exportValue( 'body' )
      ),
      'title' => $form->exportValue( 'title' ),
      'body' => $form->exportValue( 'body' )
    );
    // Save the serialized entry to a file
    $this->dataDir->createFile(
      $entry['id'] . '.dat', YDObjectUtil::serialize( $entry )
    );

    // Forward to the list view
    $this->forward( 'default' );

    // Return
    return;

  }

  // Add the form to the template
  $this->addForm( 'form', $form );

  // Output the template
  $this->outputTemplate();

}
```

This action does two separate things. It knows how to show the formwhich is used to add a new note, and it also knows how to save a note to a file on disk which can be retrieved later on. Let’s evaluate this action step by step:

```
$form = new YDForm( 'addEntryForm' );
```

This will create a new formobject called "addEntryForm".We will assign elements to this object to construct the whole form.

```
$form->addElement( 'text', 'title', 'Title:' );
$form->addElement( 'textarea', 'body', 'Contents:' );
$form->addElement( 'submit', 'cmdSubmit', 'Save' );
```

Now we add three elements to the form.We add a text element called "title", a textarea called "body" and a submit button called "cmdSubmit". For each of these elements, we also specify a label.

```
$form->applyFilter( 'title', 'trim' );
$form->applyFilter( 'body', 'trim' );
```

To the title and body field, we also add a filter called "trim". The trim filter will remove all spaces at the beginning and the end of the formvalues before validating the form.We do this to make sure that e.g. if the title would be just a space, it wouldn’t be considered as being valid.

```
$form->addRule( 'title', 'Title is required', 'required' );
$form->addRule( 'body', 'Contents is required', 'required' );
```

For the validation, we add two rules. With these two rules,we mark the elements title and body as required elements. We also specify the error message in case the validation fails.

```
$this->addForm( 'form', $form );
```

This function will assign the form object to the template. Please note that we didn’t use the setVar function, but used the addForm function instead. We need to use this function because the formobject needs some special treatment before it can be used in the template. Never use the setVar function to assing a form object to the template.

```
$this->outputTemplate();
```

The last step is to parse and output the template which is done by executing the outputTemplate function.

Note: *I specifically didn't explain the part which saves the note to a file, as it's not important yet. You first need to understand how this works before we can add the code for saving the note to disk.*

Before you can run the form,we need to add the code for the formto the template. Add the following stuff just before the </body> tag in the template:

```
<?php if ( $YD_ACTION == 'addnote' ) { ?>

  <h3>Add a new note</h3>

  <?php if ( $form['errors'] ) { ?>
    <p style="color: red"><b>Errors during processing:</b>
    <?php foreach ( $form['errors'] as $error ) { ?>
      <br><?= $error ?>
    <?php } ?>
    </p>
  <?php } ?>

  <form <?= $form['attributes'] ?>>
    <p>
      <?= $form['title']['label'] ?>
      <br>
      <?= $form['title']['html'] ?>
    </p>
    <p>
      <?= $form['body']['label'] ?>
      <br>
      <?= $form['body']['html'] ?>
    </p>
    <p>
      <?= $form['cmdSubmit']['html'] ?>
    </p>
  </form>

<?php } ?>
```

If we a look at that code, we see that it will only be shown if the current action is called addnote (always in lowercase!). The first part will take care of showing the errors if there are any.

The errors are always found in the $form['error'] array. this array is just a list of all the different error messages. In this example, we use a little but of stylesheets to make them appear in red.

Then, the code for the formitself is added.We first define the formtag, and use the $form['attributes'] variable to automatically add all the parameters of the formsuch as the action and method. The framework is smart enough to take care of that automatically.

Then we will add the different elements. Each element can be referenced as $form[elementname]. In this example,we use the label and html properties of each element. The label property contains the label as specified when you created the formobject. The html property contains the HTML version of the element.

This is the only code we need to add for the form. The framework will take care of remembering what was entered in each field and displays it when needed. It will also take care of the error messages.

Now that we have the basis of the form, you can surf to the index.php page and see what happens. If you submit the form, you will see that the values are remembered accross submits, and that the right errors are raised if the input was not valid.

Let’s examine the code that saves the entry to disk.We’ll go over it step by step.

```
if ( $form->validate() )
```

With this, we can check if the formwas validated succesfully. The formis only validated when all the validation rules were passed.

```
$entry = array(
  'id' => md5(
    $form->exportValue( 'title' ) . $form->exportValue( 'body' )
  ),
  'title' => $form->exportValue( 'title' ),
  'body' => $form->exportValue( 'body' )
);
```

This code will create a new associative array with the information of the entry.We can use the form’s exportValue function to get the value of a specific field of the form.We used the md5 function to create the unique ID for the entry.

```
// Save the serialized entry to a file
$this->dataDir->createFile(
  $entry['id'] . '.dat', YDObjectUtil::serialize( $entry )
);
```

The next line does two things. First, it will serialize the array of the object. This means it’s converted into code which can be saved to a file, and which can be read later on again to get the original array back. This function is part of the YDObjectUtil module. After we have the array as a serialized item,we can use the createFile function from the YDFSDirectory object to dump it to a file. The file name will be the id of the entry with the extension "dat".

```
$this->forward( 'default' );
```

Now that the note is saved to disk, we need to show the list of notes again.We have two options here. Either you do a redirect, which will redirect you to the url of the default action, but this requires two HTTP interactions. A lot faster is to forward the execution to a different action. The difference is that forwarding happens in the same request.

```
return;
```

It’s very important to add the return statement, since otherwise, the formwill be displayed again.

Note: *one could say that instead of forwarding the request to a different action, you could just call the function for that action. Unfortunately, that doesn’t work, since the framework will not know that the current action has been changed.*

If you run the script now, you will be able to add notes and display them. Also try to add a note without a title or description, and check that it is showing the right errors. Also check the contents of the data directory to see that the entries are correctly saved in there.

3.9 Implementing the deletenote action

To finish off, we will create the action that can delete a note. This action will take 1parameter from the URL, which is called ID. This entry will contain the unique ID of the entry. To implement this action, add the following code to the actiondeletenote function:

```
// Delete note action
function actionDeleteNote() {

  // Delete the file related to the entry
  $this->dataDir->deleteFile( $_GET['id'] . '.dat' );

  // Forward to the list view
  $this->forward( 'default' );

}
```

Let’s go over this action to see how it works:

```
$this->dataDir->deleteFile( $_GET['id'] . '.dat' );
```

With the deleteFile function from the YDFSDirectory object,we can delete the file for this specific note.We find the id of the note in the $_GET['id'] variable, which was passed with the URL.

```
$this->forward( 'default' );
```

After the deletion of the file, we just forward the request to the default action again to show the list of the notes. After adding this code,you can run the sample again and try to delete a note. If you delete the note, it should disappear from the list and the file should also be removed from the data directory.

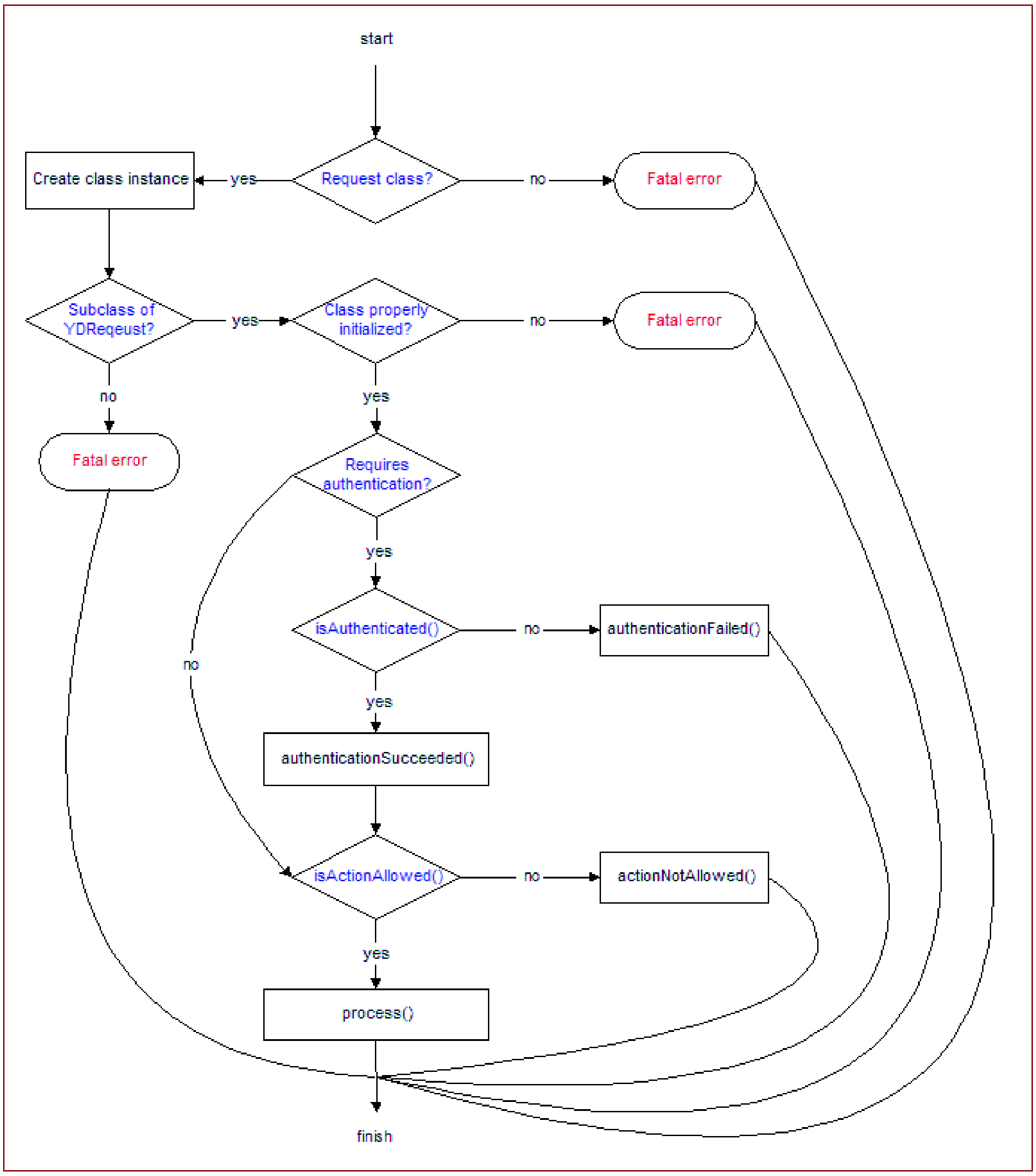
Note: *there are some functions that could be added to this action. First, it could check if there was an ID given or not. If not, you could forward the request to the default action, or you could show an error message. You could also add some checking to see if the entry exists or not before deleting it. Another option is to add a confirmation screen to prevent that someone accidently deletes a note.*

4 How requests are processed

In this chapter, we will see how requests are processed.Whenyou load a request from the browser,a lot of things are performed before the actual action is processed and the contents is displayed to the end user.

4.1 Graphical overview

The following diagram gives you a graphical overview of how requests are processed in the Yellow Duck Framework.


start
Request class?
yes
Create class instance
no
Fatal error
Subclass of YDReqeust?
yes
Class properly initialized?
no
Fatal error
no
Fatal error
yes
Requires authentication?
yes
no
isAuthenticated()
no
authenticationFailed()
yes
authenticationSucceeded()
isActionAllowed()
no
actionNotAllowed()
yes
process()
finish