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 preprend 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 preprend 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 preprend 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.

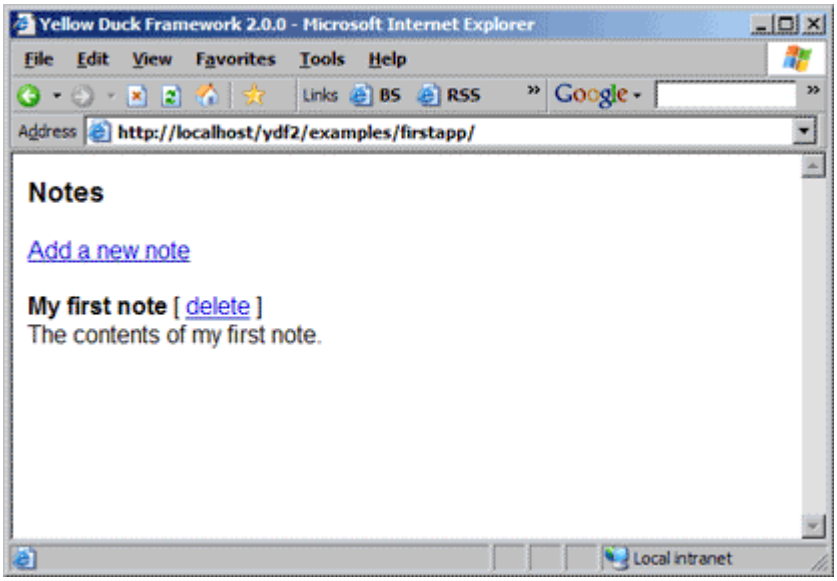
3.1 Description of the application

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

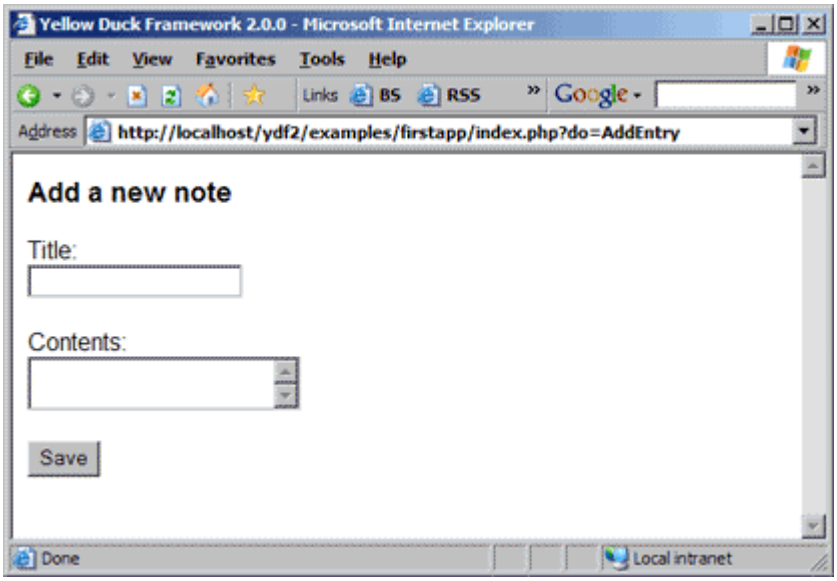
We will also use form validation to make sure that the data entered in the add note formis valid. This will prevent people from adding notes without a title or body.

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



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:



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.

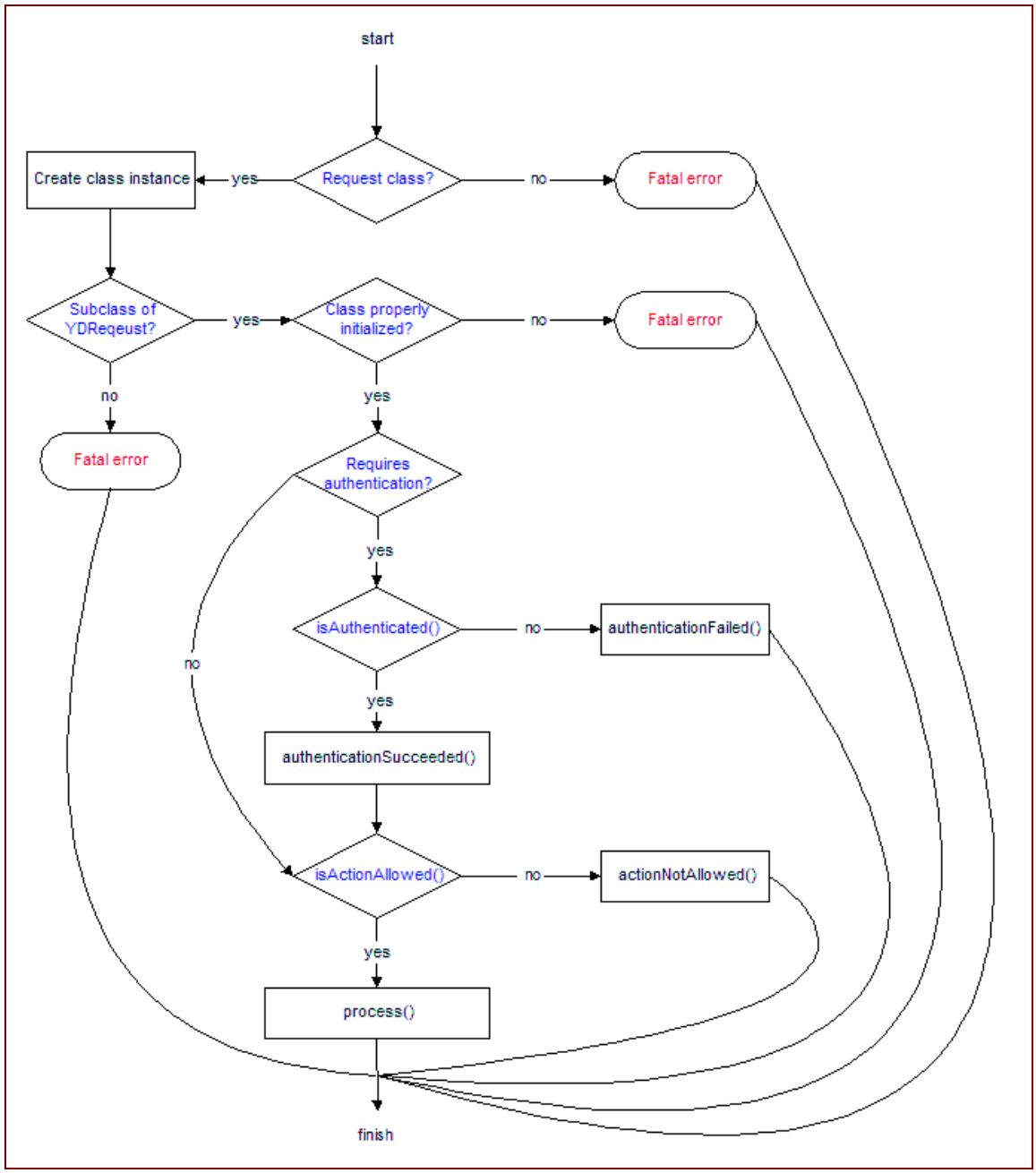
Note: *there are some functions that could be added to this action. First, it could check if there was an ID given or not. If not, you could forward the request to the default action, or you could show an error message. You could also add some checking to see if the entry exists or not before deleting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



This whole process of handling a request is automatically done by the framework. There is no need to code this yourself. Of course, if you want, there are a number of things you can override in this workflow to make the framework do what you want it to do.

4.2 Where is this processing happening

When you load a file that includes the Yellow Duck Framework using the following code, the file will be able to use the functionality of the framework. Including the Yellow Duck Framework goes as follows:

```
require_once( 'YDFramework2/YDF2_init.php' );
```

To process the actual file using the framework, we need to add another include file, but we need to include this at the end of the script. The last line of every script processed with the Yellow Duck Framework should look as follows:

```
require_once( 'YDFramework2/YDF2_process.php' );
```

The process script is the one that does what is displayed in the image above. Without including this file, nothing will be executed from the script and you will probably end up with an empty page as the result.

Let's go over each step in the process to see how it works and what influence it has on the processing of the request.

4.3 Step 1 - Check for a request class

When you load a script, the first thing the framework will do is to check for a class which is named after the script file. The class name always has the same structure. It's basically the basename of the script file with the text "Request" appended to it. Let's take a look at some examples:

- index.php -> indexRequest
- processing.php -> processingRequest
- MyOwnScriptForProcessing.php -> MyOwnScriptForProcessingRequest

If no class with the given name is declared, the execution will stop with a fatal error indicating this. The fatal error will also stop the execution of the script.

Note: *Class names are case insensitive in the Yellow Duck Framework. However, we strongly suggest you to maintain the same case as in the filename.*

Note: *Also note that some characters are not allowed as a class name, and as such are not allowed in the filename. Especially characters like a dash, a dollar sign, ad sign and an ampersand are not allowed.*

4.4 Step 2 - Is the request class derived from YDRequest?

The Yellow Duck Framework always expects that the request class has the YDRequest class as one of it's parent classes. This is needed because there are a number of standard functions from the YDRequest class that are always expected to be there.

You don't have to inherit from the YDRequest class directly. It's allowed to define your own request class based on the YDRequest class, and inherit from that class instead. You can do this to provide basic functionality to your request classes which is not available in the standard YDRequest class. You can also use this to change the way requests are processed so that you can tailor the framework to suit your needs.

If the YDRequest is not one of the parent classes from the request class, a fatal error is raised and the execution of the script will be stopped.

4.5 Step 3 - Is the request class properly initialized?

Inheriting from the YDRequest class is one thing, but it also needs to be initialized properly. In the class constructor, you need to make sure you call the function with the same name as the class name of the parent class to make sure the parent class is initialized properly. If you inherit from the YDRequest directly, you code will look as follows:

```
// Inherit from the YDRequest class
class indexRequest extends YDRequest {

  // This is the class constructor for our class.
  function indexRequest() {
    // Initialize the parent class.
    $this->YDRequest();
  }

}
```

If you would leave out the function call to YDRequest in the above class, a fatal error will be raised indicating that the YDRequest class was not properly initialized. The execution will stop at that moment.

If the YDRequest class was properly initialized, the framework will now create an instance of the request class. When instantiating, no parameters are passed to the class constructor of the request class.

4.6 Step 4 - Does this request requires authentication?

In every request class, you can enable authentication. If you enable authentication for a request class, the framework will execute certain functions to check if the authentication before the request is processed.

You can enable authentication in a request class by using the following code:

```
$this->setRequiresAuthentication( true );
```

It will then use the function "isAuthenticated" from the request class to find out if the user is already authenticated. If not, the function "authenticationFailed" is executed which can e.g. redirect to the login. page which takes care of the rest. After executing the "authenticationFailed" function, the execution of the script is stopped.

If the "isAuthenticated" function returns true, the class function "authenticationSucceeded" is executed, after which the execution of the request continues. This function can for example set a session variable indicating that we are logged in or can add an entry to the database.

***Note**: for an in depth look at how authentication is handled in the framework, please go through the chapter How to handle authentication.*

4.7 Step 5 – Is the specified action allowed?

After the authentication tokens are checked, the framework will check if the specified action is allowed by calling the "isActionAllowed" function which returns a boolean indicating this. This function can for example limit certain actions to specific conditions such as the username.

If the "isActionAllowed" function returns false, the function called "actionNotAllowed" will be executed. By default, this function returns a fatal error indicating that the action is not allowed. You can override this function to make this work the way you want. After this function is executed, the processing of the request will stop.

In the "isActionAllowed" function, you can use the class function "getActionName" to get the name of the current action in lowercase. The action name is the same as what is specified with the do function in the URL. If no action is specified, the text "default" will be returned which always points to the default action.

4.8 Step 6 – Process the actual request

Now that all the different checks are performed, the process function of the request class is executed. This will figure out the right function name for the function for the specified action, and it will execute this function.

4.9 What if?

What if I want a different way to specify the actions?
If you want to specify a different way for the framework to determine the actions, you will need to override the "getActionName" function to make it determine the correct action name

What if I want a different way of processing the actions?
You will have to override the "process" function of the request class for this. This will not change the way the different checks are happening though (which is not advised by any means).

5 Using templates

In this chapter, we will have a closer look at how the template engine in the Yellow Duck Framework is working and how it's implemented. The template engine in the framework is based on PHP and follows the same syntax as any normal PHP file.

5.1 The YDTemplate class

For managing templates, there is a specific class class YDTemplate. If you instantiate this class, you can assign variables to it, and then, using these variables, you can output a named template.

The following code sample gives you a small example on how this works:

```
$array = array(
  array(
    'author' => 'Stephen King',
    'title' => 'The Stand'
  ),
  array(
    'author' => 'Neal Stephenson',
    'title' => 'Cryptonomicon'
  ),
  array(
    'author' => 'Milton Friedman',
    'title' => 'Free to Choose'
  )
);

$template = new YDTemplate();
$template->setVar( 'title', 'This is the title' );
$template->setVar( 'array', $array );
echo( $template->getOutput( 'mytemplate' ) );
```

The template for this code could look as follows:

```
<html>

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

<body>
  <?php if ( is_array( $book ) ): ?>
    <table>
      <tr>
        <th>Author</th>
        <th>Title</th>
      </tr>
      <?php foreach ( $book as $key => $val ): ?>
        <tr>
          <td><?= $val['author'] ?></td>
          <td><?= $val['title'] ?></td>
        </tr>
      <?php endforeach; ?>
    </table>
  <?php else: ?>
```

```
    <p>There are no books to display.</p>
  <?php endif; ?>
</body>

</html>
```

As you can see, we assigned two variables to our template instance. Each variable you want to use in the template engine needs to be assigned to the template instance before you can use it. Assigning a variable to a template implies that we attach a template variable name to a PHP object. These objects can be anything from an array, a simple string to even a complete object.

Once we assigned all the variables, we can issue the "getOutput" function of the template class to get the result of the parsed template. This function takes one argument, which is the name of the template.

The name of the template you need to specify for the "getOutput" function is the basename of the template without the "tpl" extension. In our example, you would need the following files to make the example work:

- **mytemplate.php:** the actual PHP script
- **mytemplate.tpl:** the template attached to the PHP script

By convention, all templates have the extension "tpl". This is also hardcoded in the framework itself. Please, stick to this standard.

5.2 Template search paths

There are different ways on where the template engine will search for templates. By default, there are two possible ways of specifying the path for the templates:

- **Template specified by base name:** this will cause the template engine to look in the same directory as the current script.
- **Template specified by a full path:** this will cause the template engine to look for the template at the specified path.

If you want, you can override the default search path for the templates. This is done by the class function "setTemplateDir" of the template class. With this function, you can add an extra path to the search path of the template class. After you added a new path, you can specify templates in that directory by their basename instead of having to specify the full path for the template.

5.3 Standard template variables

For each template, there are a number of standard variables available which you can always reference. These variables all start with "YD_" and are all uppercase. Here's the list of the standard variables:

Variable	Explanantion
YD_FW_NAME	name of the framework
YD_FW_VERSION	version of the framework
YD_FW_NAMEVERS	the combination of the two items above
YD_FW_HOMEPAGE	the homepage of the Yellow Duck Framework
YD_SELF_SCRIPT	the current script's path, e.g. "/myapp/index.php"
YD_SELF_URI	the URI which was given in order to access this page, e.g. "/myapp/index.php?do=edit"
YD_ACTION_PARAM	the name of the $_GET parameter that specifies which action needs to be executed. This is "do" by convention.
YD_ENV	These variables are imported into PHP's global namespace from the environment under which the PHP parser is running.
YD_COOKIE	An associative array of variables passed to the current script via HTTP cookies.
YD_GET	An associative array of variables passed to the current script via the HTTP GET method.
YD_POST	An associative array of variables passed to the current script via the HTTP POST method.
YD_FILES	An associative array of items uploaded to the current script via the HTTP POST method.
YD_REQUEST	An associative array consisting of the contents of YD_GET, YD_POST and YD_COOKIE.
YD_SESSION	An associative array containing session variables available to the current script.
YD_GLOBALS	An associative array containing references to all variables which are currently defined in the global scope of the script.

Please do not use the "YD_" prefix for your own variables. This prefix is uniquely reserverd for the variables automatically assigned by the Yellow Duck Framework.

5.4 PHP configuration for short open tags

Depending on how you want to write the PHP constructs, you might need to enable the short open tags function in PHP. This is done by changing the PHP configuration in php.ini or using a .htaccess file. You will need to add the following line to enable short open tags:

Using a .htaccess file

```
php_flag short_open_tag On
```

Using the php.ini file

```
short_open_tag = On
```

All the examples for the Yellow Duck Framework have been written using short open tags. Make sure you have them enabled before running the examples.

6 Using and validating forms

Processing and validating forms is probably one of the most time-consuming and most difficult things to handle in a web application. Forms can vary from very simple forms to really difficult forms, which makes having a unified system to handle all these forms a real time-saver.

By applying an object oriented approach to forms, and by providing different methods and objects to handle forms with ease, processing forms has become really easy and fast.

The form module in the Yellow Duck Framework is based on a package from the PEAR (PHP Extension and Application Repository) libraries. The package we used is called HTML_QuickForm and provides most of the functionality. A number of additions were made to this package to make it more straightforward for using this in the Yellow Duck Framework.

6.1 The YDForm class

All te functionality for form handling in the Yellow Duck Framework is handled by a class called "YDForm". The YDForm class extends the HTML_QuickForm class to provide some additional features only found in the Yellow Duck Framework.

In addition to the standard functions provided by the HTML_QuickForm class, the YDForm class provides one additional function, called "toArray", which converts the form object into an array suitable for using with the YDTemplate class.

For an overview of the standard functions for the HTML_QuickForm class, please have a look on the documentation page which you can find on: http://pear.php.net/manual/en/package.html.html-quickform.php.

6.2 Creating a new form

Creating a new form is done in two steps. First, you need to include the class definition, and then you need to create an instance of the YDForm class. The following code sample will illustrate this:

```
require_once( 'YDForm.php' );
$form = new YDForm( 'myForm' );
```

The first line will include the correct class definition, while the second one will create an instance of the form class. The class constructor takes a number of arguments, but only one is required, the rest is optional. Here's the list of parameters that you can pass when instantiating the form object.

Argument	Description
$name	The name of the form.
$method (optional)	Method used for submitting the form. Most of the times, this is either POST or GET.
$action (optional)	Action used for submitting the form. If not specified, it will default to

	the current URI.
$target (optional)	HTML target for the form.
$attributes (optional)	Attributes for the form.

Make sure that all the different form instances you want to use all have separate names. If not, the results might not be as you expected.

By default, the form's action – the URI that will be called when the form is submitted – is the same URI that displayed the form in the first place. This may seem strange at first, but it is actually very useful, as we shall see.

6.3 Adding form elements

Adding form elements is done using the addElement function from the YDForm class. Depending on the element you want to add, you need to specify a different number of parameters.

The syntax for the addElement function looks as follows:

```
object HTML_QuickForm::addElement( mixed $element )
```

If $element is a string representing an element type, then this method accepts variable number of parameters, their meaning and count depending on element type.

Parameters starting from second will be passed to the element's constructor, consult the docs for the appropriate element to find out which parameters to pass.

You can add as many elements to the form as you like. The YDForm class is smart enough to preserve the values of the different form elements between requests, so you don't have to worry about this. The Yellow Duck Framework takes care of this for you.

The next section will give you an overview of the different element types you can add to the YDForm class.

6.4 The different form elements

There are a whole number of standard form elements (sometimes called widgets) that can be added to a form. Here's an overview of the different elements together with the HTML element they represent.

First, let's have a look at the standard HTML elements.

- **button**: Class for <input type="button" /> elements.
- **checkbox**: Class for <input type="checkbox" /> elements.
- **file**: Class for <input type="file" /> elements.
- **hidden**: Class for <input type="hidden" /> elements.
- **image**: Class for <input type="image" /> elements.

- **password**: Class for <input type="password" /> elements. docs
- **radio**: Class for <input type="radio" /> elements.
- **reset**: Class for <input type="reset" /> elements.
- **select**: Class for <select /> elements. The class allows loading of <option> elements from array or database.
- **submit**: Class for <input type="submit" /> elements.
- **text**: Class for <input type="text" /> elements.
- **textarea**: Class for <textarea /> elements.

There are also some custom element types. These are generally a bit more complex but are able to do things which would otherwise require a lot of manual work.

- **advcheckbox**: Class for an advanced checkbox type field. Basically this fixes a problem that HTML has had where checkboxes can only pass a single value (the value of the checkbox when checked).
- **autocomplete**: Class for a text field with autocompletion feature. The element looks like a normal HTML input text element that at every keypressed javascript event, searches the array of options for a match and autocompletes the text in case of match.
- **date**: Class for a group of elements used to input dates (and times).
- **group**: Class for a form element group. QuickForm (and also YDForm) allows grouping of several elements into one entity and using this entity as a new element.
- **header**: Class for adding headers to the form.
- **hiddenselect**: This class behaves as a select element, but instead of creating a <select> it creates hidden elements for all values already selected with setDefaults() or setConstants().
- **hierselect**: Class to dynamically create two HTML <select> elements. The first select changes the content of the second select.
- **link**: Class for a link type field.
- **static**: Class for static data.
- **bbtextarea**: this is a special type of textarea field which has a toolbar for adding BBCode style tags to markup the text.

***Note**: it is possible to make your own widgets as well. Please refer to the documentation of the PEAR project.*

6.5 Displaying the form

For displaying the form, there are several options. The way you display the form is largely influenced by how much control you want to have over the form rendering and whether you want to use the YDTemplate class for templates or not.

The easiest way of displaying your form is by calling it's class function called display. This will render the form and display the resulting HTML. This is definitely the easiest way of working, but you have very little to almost no control on how the form will be displayed.

If you want to use a template engine, there are two different ways of rendering the form. If you still want to stick with the default layout, you can use the toHtml class function of the YDForm class to get the HTML representation of your form. You can then assign this as a normal

template variable to the template class. This again is a very quick way of working but only gives you limited possibilities to determine the exact look and feel of the rendered form.

If you want full control and you will be using templates (which is the advised way of working), you can use the class function toArray to get an array representation of your form which is very easy to use in your code. Let's take a look at how the following form code is converted to an array:

```
<?php

$form = new YDForm( 'firstForm' );
$form->setDefaults( array( 'name' => 'Joe User' ) );
$form->addElement(
  'text', 'name', 'Enter your name:', array( 'size' => 50 )
);
$form->addElement(
  'textarea', 'desc', 'Enter the description:'
);
$form->addElement( 'submit', 'cmdSubmit', 'Send' );0

?>
```

The code above results in the following array structure:

```
array (
  'frozen' => false,
  'javascript' => '',
  'attributes' => 'action="/form.php" method="post" name="firstForm" id="firstForm"',
  'requirednote' => '* denotes required field',
  'errors' =>
  array (
  ),
  'hidden' => '',
  'name' =>
  array (
    'name' => 'name',
    'value' => 'Joe User',
    'type' => 'text',
    'frozen' => false,
    'label' => 'Enter your name:',
    'required' => true,
    'error' => NULL,
    'html' => '<input size="50" name="name" type="text" value="Joe User" />',
  ),
  'desc' =>
  array (
    'name' => 'desc',
    'value' => NULL,
    'type' => 'textarea',
    'frozen' => false,
    'label' => 'Enter the description:',
    'required' => false,
    'error' => NULL,
    'html' => '<textarea name="desc"></textarea>',
  ),
  'cmdSubmit' =>
  array (
    'name' => 'cmdSubmit',
    'value' => 'Send',
    'type' => 'submit',
```

```
    'frozen' => false,
    'label' => '',
    'required' => false,
    'error' => NULL,
    'html' => '<input name="cmdSubmit" value="Send" type="submit" />',
  ),
)
```

The structure of this array is really easy. Every form array always contains a number of standard elements of which only a few are important for us right now. Let's go over them:

- **attributes**: This variable contains the attributes for the actual form HTML tag. It contains parameters such as the action, method, name and ID of the form.
- **javascript**: The javascript for the form in case you are using client-side validation (see later on). This is the part that should go in the head tag of the HTML page.
- **errors**: The errors collection is an array which contains a list of all the errors that were encountered during the validation of the form. We will discuss this in depth later on.

Then it contains a subarray for each element in the form. Each of these subforms have the same structure. Let's have a look at the different parameters for each form element:

- **name**: The name of the form element.
- **value**: The value of the form element (which is generally the contents of the form element).
- **type**: The type of the element.
- **label**: The label specified for the form element.
- **required**: Indicating if this is a required form element or or not. This is always a boolean value.
- **error**: The error message for this form element after validating the form. If not error is attached to the form, this will contain a NULL value.
- **html**: The HTML representation of the form element.

Instead of first having to convert the form object to an array for assigning it to the template, there is a shortcut function called addForm in the YDTemplate class which automatically converts the form object to an array and assigns it to the template object.

By using the elements of this array in the template, you can pretty much layout the field exactly as you wish. It looks like a lot of work, but it's a lot easier than handcoding every single part of the form.

6.6 Validating forms

The concept of validating forms is to evaluate the input of the form and checking to see if it matches the specified rules. A form is considered to be validated when the URI called to display it has one or more parameters (either GET or POST according to the method used by the form), and all the rules (which we will discuss in a second) have been satisfied.

For validating forms, we will use three specific functions of the YDForm class. Here is an overview of the ones we are going to use:

- **applyFilter:** We will use this function to massage the data of the form before validating it. We can use this function to e.g. convert a field value to uppercase or we can use it to remove leading and trailing spaces from the actual value.
- **addRule:** This function attaches a rule to a specific form element and also defines the error message that needs to be shown if this rule is not honoured.
- **validate:** This function will check the input of the form against all rules after applying the specified filters, and will return either true or false to indicate if the form was valid or not.

The applyFilter function takes two arguments. The first argument is either the name of an element or the string "__ALL__", meaning that this filter applies to all elements on the form. The second argument is either a function, user-defined or built-in function.

There are two ways of having the rules checked an applied. You can choose between server side and client side evaluation of the rules. By default, the rule are server side evaluated. By changing the way you call the addRule function, you can change the rule to client side evaluation. The following code sample illustrates adding a client side rule:

```
$form->addRule( 'txt1', 'name required', 'required', '', 'client' );
```

If you omit the parameter called 'client', the rule will be a server side evaluated rule again. Be aware that client side evaluation is not as clear as server side evaluation and is less secure. Since the client side evaluation is based on JavaScript, it's easy to get around the validation by disabling JavaScript in your browser. The only advantage the client side evaluation has is that it saves you the roundtrip to the server.

Up till now, we only discussed one specific rule, namely the required rule. There are of course a lot more rules which you can use. Here's the complete list:

- **required**: value is not empty
- **maxlength**: value must not exceed n characters
- **minlength**: value must have more than n characters
- **rangelength**: value must have between m and n characters
- **regex**: value must pass the regular expression
- **email**: value is a correctly formatted email
- **lettersonly**: value must contain only letters
- **alphanumeric**: value must contain only letters and numbers
- **numeric**: value must be a number
- **nopunctuation**: value must not contain punctuation characters
- **nonzero**: value must be a number not starting with 0
- **callback**: This rule allows to use an external function/method for validation, either by registering it or by passing a callback as a format parameter.
- **compare**: The rule allows to compare the values of two form fields. This can be used for e.g. 'Password repeat must match password' kind of rule.

There are also some rules that apply specifically to file uploads:

- **uploadedfile:** Required file upload
- **maxfilesize:** The file size must not exceed the given number of bytes
- **mimetype**: The file must have a correct mimetype
- **filename:** The filename must match the given regex

Note: *Some rules (for example the rangelength rule) take an additional argument indicating the length. Please refer to the PEAR documentation for an explanation of each rule.*

Now that we assigned all the rules and filters to the form, we can use the validate function to check if the input from our form was validated succesfully. Only if this condition is met, we will do something with the data of the form. This is also called "processing the form" which is discussed in the next section.

6.7 Processing forms

For the actual processing of the forms, there are only two functions you will really use a lot. This is the class function exportValue from the YDForm class and it's companion function called exportValues.

The exportValue function will export the value of a named form element. Remember that this function will export the cleaned-up value of the form element. This means that the return value of this function will also have all the possible form filters applied before it's returned.

The exportValues will do the same as the exportValue function but will return the values for all the elements in the form. Unfortunately, this function does not work with file uploads. Again, the values returned by this function will also have all the necessary filters applied to them.

6.8 More information

For more information about using the HTML_QuickForm package from the PEAR project, you can visit the following websites:

http://www.thelinuxconsultancy.co.uk/quickform.html
http://pear.php.net/manual/en/package.html.html-quickform.php

7 Debugging tools

Debugging is an integral part of every web application. Debugging is the art of finding problems in your web application and fixing them. In the Yellow Duck Framework, there are a number of tools in addition to the standard PHP tools that will help you debugging your applications based on the Yellow Duck Framework.

7.1 Showing debug information

When the Yellow Duck Framework is initialized, it automatically defines a constant called YD_DEBUG which is a boolean telling us if we are running in debug mode or not. Depending on this setting, the framework will either show or hide debugging information.

There are a number of ways of turning on debugging in the Yellow Duck Framework. The first and easiest one is to add the parameter YD_DEBUG=1 to the URL of the request. This will enable showing of the debug messages.

Take a look at the following URLs which demonstrate this:
http://localhost/test.php?YD_DEBUG=1
http://localhost/test.php?id=01&YD_DEBUG=1

This option will turn on the debug messages for the current request only. If you are developing a new application, it might be easier to constantly turn on debugging. This is done by defining a constant called YD_DEBUG before you include the YDF2_init.php file. Having this constant defined before you initialize the framework will override the default debugging settings, and will force the debugging to be turned on or off.

***Note**: for release applications, you want to prevent users from seeing debug information. In order to prevent people from using the YD_DEBUG URL parameter to show debug information, which can reveal confidential information, make sure you define the YD_DEBUG constant with a value of 0 before you initialize the framework.*

Now that we know how debugging information can be enabled, let's have a look at how the Yellow Duck Framework displays this information. We specifically choose not to use any popup windows for this or display the information in the visible HTML code, since both methods are not easy to handle. In the Yellow Duck Framework, debugging information is added to the source code in the form of HTML comments. This makes it easy to find them without screwing up the layout of your pages. The following code snippet shows you an example of HMTL code with debugging information added:

```
Supported languages: en
<pre>'en'</pre>
Supported languages: nl, fr, en
<pre>'en'</pre>
Browser languages:
<pre>array (
  0 =&gt; 'en',
  1 =&gt; 'nl',
  2 =&gt; 'fr',
)</pre>
<!--
```

```
[ YD_DEBUG ]

Processing time: 11 ms

Total size include files: 66 KB

Included files:
  C:\program files\apache\htdocs\ydf2\examples\language1.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDF2_init.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDBase.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDError.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDPhpUtil.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDTimer.php
  c:\program files\apache\htdocs\ydf2\YDFramework2\YDF2_init.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDRequest.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDTemplate.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDObjectUtil.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDLanguage.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDDebugUtil.php
  c:\program files\apache\htdocs\ydf2\YDFramework2\YDF2_process.php
  C:\program files\apache\htdocs\ydf2\YDFramework2\YDClasses\YDExecutor.php
-->
```

As you can see, the debug comments from the framework are always prepended but the text "[YD_DEBUG]" to make them easily recognizable. Each debug comment will also start on a new line to make it even easier.

7.2 Standard debug information

By default, the Yellow Duck Framework shows a whole bunch of standard extra information if you turn on debugging. The following information is always shown in debugging mode:

- The total processing time in seconds for this request.
- The total filesize of the include files.
- The list of the full paths of the included files.

In the next section, we will see that it's very easy to add your own debugging information.

7.3 Adding debugging information

To add your own debugging information, you can use the YDDebugUtil module. This module has three static functions specifically designed for debugging purposes.

By using the static function call YDDebugUtil::debug, we can show our own debugging information. This function takes a variable number of arguments and will glue all these together before showing the debug message.

The static function call YDDebugUtil::dump will dump the contents of any variable and display it on the screen. It basically does the same as the native var_dump function from PHP, but the output is much more readable.

The static function call YDDebugUtil::r_dump does the same as the YDDebugUtil::dump function, but it will return the information instead of displaying it. If you combine this function together with the function YDDebugUtil::debug, you can dump the contents of a variable as a debug message.

8 Handling authentication

Needs to be written.

8.1 Example 1 - IP based authentication

Needs to be written.

8.2 Example 2 - Basic HTTP authentication

Needs to be written.

8.3 Example 3 - Form based authentication

Needs to be written.

9 Using files, images and directories

Needs to be written.

9.1 Why this abstraction?

Needs to be written.

9.2 Using files

Needs to be written.

9.3 Using images

Needs to be written.

9.4 Using directories

Needs to be written.

10 Using URLs and downloading data

Needs to be written.

10.1 Raw HTTP connections

Needs to be written.

10.2 The YDUrl class

Needs to be written.

10.3 Caching of downloaded data

Needs to be written.

11 Using XML/RPC clients and servers

11.1 What is XML/RPC?

It's a spec and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

It's remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

To invoke an XML/RPC call, an XML/RPC client might send the following XML data to a server:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>tide.getTideForDay</methodName>
  <params>
    <param>
      <value>
        <string>2004-05-11</string>
      </value>
    </param>
  </params>
</methodCall>
```

This server will then interpret the XML, execute the right function and send back the following XML data:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>tideDate</name>
            <value><string>2004-05-11</string></value>
          </member>
          <member>
            <name>tideDay</name>
            <value><string>di</string></value>
          </member>
          <member>
            <name>tideLow1</name>
            <value><string>01:07:00</string></value>
          </member>
          <member>
            <name>tideLow2</name>
            <value><string>13:39:00</string></value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodResponse>
```

The XML/RPC client will interpret this XML response and will convert it back to objects and variables in the native programming language.

The beauty of XML/RPC is that it's really easy to use and still high performant if implemented correctly. The client and the server can be speaking two totally different programming languages and still are able to exchange data between the two with preserving the native data types.

The Yellow Duck Framework has classes for creating both XML/RPC clients and servers. Both use the very latest technology and are one of the few implementations that support GZip compression for the HTTP data streams, resulting in better performance.

The YDXmlRpcClient class implements an XML/RPC client while the YDXmlRpcServer class implements a full XML/RPC server which supports introspection so that the client can easily determine what functions are supported by the server.

Let's have a look at how these classes can be used to perform XML/RPC based communication.

11.2 Using YDXmlRpcClient

To create an XML/RPC client, we will use the YDXmlRpcClient class which does all the work for us. When you instantiate this class, you need to specify the URL of the XML/RPC server.

Once you made an instance of the XML/RPC client, you can execute functions on that server by using it's execute function. This function takes two arguments which is the name of the XML/RPC method and an optional array with parameters.

The request will then be executed and the result will be returned as real PHP values and objects.

Let's have a look at the following code sample:

```
$client = new YDXmlRpcClient(
  'http://www.grijzeblubber.be/bba/xmlrpc.php'
);

$result = $client->execute(
  'tide.getTideForDay', array( '2004-05-11' )
);
```

In this example, we first connect to the XML/RPC server using the specified URL. Then we execute the 'tide.getTideForDay' function on that server which returns an array.

We had to specify a string with the date. After executing this, the variable $result will have the following contents:

```
array (
  'tideDate' => '2004-05-11',
  'tideDay' => 'di',
  'tideLow1' => '01:07:00',
  'tideLow2' => '13:39:00',
)
```

If something went wrong or the XML/RPC server returned an error, a fatal error will be raised with the right error message. The error message will be the same as the error message returned by the XML/RPC server.

What you can not see in this example is that if the XML/RPC server supports GZip compression, this will be used to do the communication. This not only saves bandwidth but also improves the performance of the XML/RPC calls quite a lot. Since a new HTTP request has to be made for each XML/RPC call, reducing the total size of information send over the network can really improve the performance and scalability of your XML/RPC based services.

11.3 Using YDXmlRpcServer

Needs to be written.

11.4 More information

More information on XML/RPC can be found on the official XML/RPC website. You can find this website on the following URL:

http://www.xmlrpc.com/

12 Sending emails

In the Yellow Duck Framework, we provided a special class to easily send emails with all the possible options. It is possible with the framework to create emails that have both text and HTML contents, inline images are supported for HTML emails and also file attachments are fully supported.

The Yellow Duck Framework supports the following options for sending emails:

- Specifying the from address
- Specifying the reply-to address
- Specifying one or more to addresses
- Specifying one or more cc addresses
- Specifying one or more bcc addresses
- Specifying the plain text version of the email
- Specifying the HTML version of the email
- File attachments
- Inline HTML images

Note: *The emails outputted with the Yellow Duck Framework have been tested with a variety of email clients and servers to make sure they were all correctly supported.*

12.1 The YDEmail class

The class YDEmail in the Yellow Duck Framework is the class that defines an email. You can use the different class methods to set the recipients, subject, contents and so on for your email.

Once the email object is set up, you can use the function send to send the actual email to the correct recipients. This function takes no arguments and is smart enough to figure out the list of recipients to send the email to.

Before sending the email, a number of conditions is checked which would prevent the email from being send properly. An error will be raised if you forgot to specify at least one recipient for your email. You will also be forced to specify the sender from the email.

Note: For the sending of the emails, the internal mail function is used to do the hard work. Depending on your computer platform, this might requires some setup. Please refer to the PHP Online Documentation for instructions on how to set this up.

Note: If you specify HTML text as the text only contents of your email, all HTML tags will be stripped from the text. We cannot guarantee that the layout of the text will be properly preserved.

12.2 An example

The following sample demonstrates you how you can setup a YDEmail object, fill it and send it to the specified recipients.

```
$eml = new YDEmail();
$eml->setFrom( 'pieter@yellowduck.be', YD_FW_NAME );
$eml->addTo( 'pieter@yellowduck.be' );
$eml->setSubject( 'Hello from Pieter & Fiona!' );
$eml->setTxtBody( $body );
$eml->setHtmlBody( $body );
$eml->addAttachment( 'email1.tpl' );
$eml->addHtmlImage( 'fsimage1.jpg', 'image/jpeg' );
$eml->send();
```

In this code sample, we first created a new instance of the YDEmail class to represent our email message.

After that, we set the correct from address. This is normally your own email address, and unless otherwise specified, it will be used as the address to reply to. Always make this a valid and existing address unless you do not want people to reply to your email.

Then we added a recipient using the addTo function. In this example, we only specified the email address. You can optionally specify a name as well which will be the name shown in the email client.

Then we defined the subject for the email. Remember that the subject of an email is not supposed to contain HTML data. If you happened to have HTML in there, the framework will automatically remove the HTML tags.

The following two lines specify the plain text and HTML body of our email message. You don't have to specify both. If you included HTML tags in the plain text version of your email, the HTML tags will be automatically removed.

Using the addAttachment function, we added a file attachment to the email. The only argument we specified was the path to the file we want to attach.

Then we used the addHtmlImage function to add an image which we can use as an inline image in the HTML version of our email. We specified both the path of the image as well as the image MIME type.

Then we invoked the send function to actually send the email to the different recipients.

12.3 Combining YDTemplate and YDEmail

To make it easier to construct the contents of your email, you can combine the YDTemplate class with the YDEmail class to create email messages based on templates.

The way this works is by first creating a YDTemplate instance which will be used to create the body of your email. After you setup the YDTemplate instance, you can use the getOutput

function to get the parsed result of your template object. You will probably have to specify the name of a template as you will probably use a different template than the one from your script.

After you got the body of your email, you can assign this to the YDEmail class using the functions setTxtBody and setHtmlBody. The following example demonstrates this concept:

```
// Parse the template for the email
$emlTpl = new YDTemplate();
$emlTpl->setVar( 'email', $form->exportValue( 'email' ) );
$body = $emlTpl->getOutput( 'email1_template' );

// Send the email
$eml = new YDEmail();
$eml->setFrom( 'pieter@yellowduck.be', YD_FW_NAME );
$eml->addTo( $form->exportValue( 'email' ) );
$eml->setSubject( 'Hello from Pieter & Fiona!' );
$eml->setTxtBody( $body );
$eml->setHtmlBody( $body );
$eml->addAttachment( 'email1.tpl' );
$eml->addHtmlImage( 'fsimage1.jpg', 'image/jpeg' );
$eml->send();
```

The template for this email will look like:

```
<html>

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

<body>

  <p>Hello from Pieter & Fiona</p>

  <p><img src="fsimage1.jpg" border="1" alt="Pieter & Fiona"
  width="320" height="213"></p>

  <p>This email was sent to
  <a href="mailto:<?= $email ?>"><?= $email ?></a>.</p>

  <p><?= $YD_FW_NAMEVERS ?></p>

</body>

</html>
```

***Note**: if you look carefully at the example above, you will see that we referenced the image in the template by it's name given when you added the file to the YDEmail object. This src attribute should only contain the name of the image, and not a local path. An internet url is supported as well.*

13 Creating RSS and ATOM feeds

RSS and ATOM are XML-based file formats intended to allow lists of information, known as "feeds", to be synchronised between publishers and consumers. Feeds are composed of a number of items, known as "entries", each with an extensible set of attached metadata. For example, each entry has a title.

13.1 What is RSS and ATOM?

The primary use case that Atom and RSS address is for syndicating Web content such as Weblogs and news headlines to other Web sites and directly to consumers. However, nothing precludes it from being used for other purposes and types of content.

RSS is the oldest of the two, and has undergone different revisisions. Currently, there are 3 major versions of RSS, which is version 0.91, version 1.0 and version 2.0. RSS 1.0 is also referenced as RDF Site Summary. Let's have a look at the XML contents for each one of them:

This is an example of an RSS 0.91 XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rss version="0.91">
  <channel>
    <title>Yellow Duck Framework</title>
    <description>News about the Yellow Duck Framework</description>
    <link>http://localhost/ydf2/?do=rss091</link>
    <item>
      <title>Title 1</title>
      <link>http://localhost/ydf2/?do=rss091#1</link>
      <description>Description 1</description>
      <guid>3a31c01d5c9023115e2433cdb1b6515e</guid>
    </item>
    <item>
      <title>Title 2</title>
      <link>http://localhost/ydf2/?do=rss091#2</link>
      <description>Description 2</description>
      <guid>be3e15a372932faea5b235d1965c36bf</guid>
    </item>
  </channel>
</rss>
```

This code sample shows you the source of a RSS 1.0 XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns="http://purl.org/rss/1.0/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel rdf:about="">
    <title>Yellow Duck Framework</title>
    <description>News about the Yellow Duck Framework</description>
    <link>http://localhost/ydf2/?do=rss10</link>
    <dc:date>2004-05-22T13:36:11+01:00</dc:date>
    <items>
      <rdf:Seq>
        <rdf:li rdf:resource="http://localhost/ydf2/?do=rss10#1"/>
        <rdf:li rdf:resource="http://localhost/ydf2/?do=rss10#2"/>
```

```
      </rdf:Seq>
    </items>
  </channel>
  <item rdf:about="http://PClaerho1/ydf2/examples/?do=rss10#1">
    <dc:format>text/html</dc:format>
    <title>Title 1</title>
    <link>http://localhost/ydf2/feedcreator1.php?do=rss10#1</link>
    <description>Description 1</description>
  </item>
  <item rdf:about="http://PClaerho1/ydf2/examples/?do=rss10#2">
    <dc:format>text/html</dc:format>
    <title>Title 2</title>
    <link>http://localhost/ydf2/feedcreator1.php?do=rss10#2</link>
    <description>Description 2</description>
  </item>
</rdf:RDF>
```

The code for an RSS 2.0 feed looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rss version="2.0">
  <channel>
    <title>Yellow Duck Framework</title>
    <description>News about the Yellow Duck Framework</description>
    <link>http://localhost/ydf2/?do=rss20</link>
    <lastBuildDate>Sat, 22 May 2004 13:36:15+0100</lastBuildDate>
    <item>
      <title>Title 1</title>
      <link>http://localhost/ydf2/?do=rss20#1</link>
      <description>Description 1</description>
      <guid>d1c9daf41c19e53ed46f8081d4ab99b7</guid>
    </item>
    <item>
      <title>Title 2</title>
      <link>http://localhost/ydf2/?do=rss20#2</link>
      <description>Description 2</description>
      <guid>7dbd540a343ecc960efa3c1d9f715e74</guid>
    </item>
  </channel>
</rss>
```

Atom looks similar, but has again some very subtle differences:

```
<?xml version="1.0" encoding="utf-8"?>
<feed version="0.1" xmlns="http://example.com/newformat#">
  <title>Yellow Duck Framework</title>
  <subtitle>News about the Yellow Duck Framework</subtitle>
  <link>http://localhost/ydf2/?do=atom</link>
  <entry>
    <title>Title 1</title>
    <link>http://localhost/ydf2/?do=atom#1</link>
    <id>2e3c209c100c00c83a791fdbc0c6af1b</id>
    <content type="text/html" xml:lang="en-us">
      <div xmlns="http://www.w3.org/1999/xhtml">
        Description 1
      </div>
    </content>
  </entry>
  <entry>
    <title>Title 2</title>
    <link>http://localhost/ydf2/?do=atom#2</link>
```

```
    <id>5d7ed2c293a6e24c9be308e81a2fb8f5</id>
    <content type="text/html" xml:lang="en-us">
      <div xmlns="http://www.w3.org/1999/xhtml">
        Description 2
      </div>
    </content>
  </entry>
</feed>
```

As you can see with the different examples, the idea is always the same. First, you have the properties of the feed itself, which contains a title, a link, the actual URL of the XML file and so. Then you have one or more entries which again have their own set of properties.

The YDFeedCreator class in the Yellow Duck Framework provides you with an object oriented way of generating these XML files. Note: The YDFeedCreator class will try to make all the data you put in the feed XML compliant, but sometimes, this is not possible. You might end up with feed that are not 100% valid in that sense.

Note: Atom feeds have the most strict specification and expects you to make all the HTML contents you put in e.g. the item descriptions valid XML.

13.2 Setting up the feed

When we want to make a new feed, we first have to setup the feed itself. The following code sample illustrates this:

```
$fc = new YDFeedCreator();
$fc->setTitle( 'Yellow Duck Framework' );
$fc->setDescription( 'News about the Yellow Duck Framework' );
$fc->setLink( 'http://localhost/ydf2/' );
```

In the above code sample, we first created a new instance of the YDFeedCreator class. Please note that we didn't specify the format of the feed yet, as we will do this only when the actual feed get's created.

The second and third line define the title and the description of the feed. The description of the feed is the only one that supports embedded HTML. If you specified HTML in the title of the feed, this will be removed automatically. The description is an optional item for the feed and can be omitted.

The last line defines the source of the feed. This is normally a a link to the website where the XML data came from. This is not the same as the actual URL of the XML data.

13.3 Adding items

Now that the feed is setup correctly, we can start adding items to it. When you add an item, we can use the following code to do this:

```
$fc->addItem(
    'Title 1 & Co.',
    'http://localhost/ydf2/#1',
    'Description 1 & Co.'
```

```
);
$fc->addItem(
     'Title 2 & Co.',
     'http://localhost/ydf2/#2',
     'Description <b>2</b> & Co.'
);
```

By using the addItem function of the YDFeedCreator object, we can add new items to the feed. Each item requires a number of parameters.

For each feed item, a title and a link are required. The link is the actual URL pointing to the entry itself on the original website. Additionally, you can add a description as well which can contain HTML data, although, it's not required.

As the last argument, you can specify a GUID (Global Unique Identifier) for your item. This is the identifier that uniquely identifies this item. If you omit this, the Yellow Duck Framework will create one automatically for you.

The automatic GUID is created by making the checksum of the link of the original website, the link of the item and the title of the item. If you don't want to use the automatic GUID creation, you can create your own.

13.4 Outputting the feed

After setting up the feed, and adding the items, we can create the XML version of the feed. This is done by using either the toXml or outputXml class functions.

The toXml will convert the feed to XML and will return as string representation of the feed. The outputXml function will also convert the feed to XML, but will output it directly to the browser using the correct HTTP headers to indicate that the contents being send is XML and not HTML.

Both functions take one argument which defines which kind of XML needs to be outputted. If no format is specified, it will assume that you want to output RSS 2.0. The following formats are supported:

- RSS0.91
- RSS1.0
- RSS2.0
- ATOM

There is also a little helper function called getColoredXml which will return a HTML version of the XML version of the feed with colored XML tags. This is mainly used for debugging purposes. This function takes the same arguments as the toXml and outputXml functions.

14 Error handling

This chapter will explain how errors can be reported and handled in the Yellow Duck Framework.

Needs to be written.

15 Other classes and modules

In this chapter, we will discuss some of the smaller classes and modules. There are a whole bunch of small extra things in the Yellow Duck Framework that take care of some very specific functions. Most of these classes are not rocket science, but they make that you don't have to reinvent the wheel every time you need them.

15.1 YDArrayUtil

The YDArrayUtil module houses all the different function related to handling of arrays. Currently, there is only one function in this module which converts a single-dimension array to a two-dimension array (table). Let's take a look at an example.

The original array looked as follows:

```
array (
  0 => 1,
  1 => 2,
  2 => 3,
  3 => 4,
  4 => 5,
  5 => 6,
  6 => 7,
)
```

Let's say we want to convert it to a table with 3 columns, we can do the following:

```
YDArrayUtil::convertToTable( $array, 3 );
```

The new array will look as follows:

```
array (
  0 =>
  array (
    0 => 1,
    1 => 2,
    2 => 3,
  ),
  1 =>
  array (
    0 => 4,
    1 => 5,
    2 => 6,
  ),
  2 =>
  array (
    0 => 7,
  ),
)
```

There is a switch for the convertToTable that can fill the last row with null values so that it matches the correct number of columns.

This function was primarly made for use in things such as image galleries that need to display their contents in columns. It can of course be used for a lot more than just image galleries.

15.2 YDBrowserInfo

The YDBrowserInfo object returns information about the browser that performed the request. The YDBrowserInfo will check the headers and extract information such as the version of the browser, the name of the browser and the platform on which the browser is running.

The following code sample demonstrates how to use this class:

```
$browser = new YDBrowserInfo()
echo( $browser->getAgent() );
echo( $browser->getBrowser() );
echo( $browser->getVersion() );
echo( $browser->getPlatform() );
echo( $browser->getDotNetRuntimes() );
echo( $browser->getBrowserLanguages() );
```

The getBrowser function will return one of the following values: ie, safari, opera, mozilla or other. The getPlatform function will return one of the following values: win, mac, linux, unix or other.

The getAgent function will return the full HTTP user agent string as provided by the browser.

This class is very handy to display specific contents based on the browser of the user. You might e.g. specify a different stylesheet for Macintosh users compared to the stylesheet used for Windows users.

15.3 YDLanguage

The YDLanguage class helps you to choose a language from a list of supported languages based on the supported languages from the browser.

This function will check the HTTP_ACCEPT_LANGUAGE header to find out which languages the browser supports. It will then check that against a list of languages that the class supports and will return the name of the most appropriate langauage.

This class will also take into consideration the right priority of the languages specified by the browser. Let's take a look at a small example illustrating this. In this example, the browser supports the following languages in this specific order: nl (dutch), fr (french) and en (english). The code sample looks as follows:

```
$lang = new YDLanguage( array( 'en', 'nl', 'fr' ) );
echo( $lang->getLanguage() );
```

If you run this example, the output of the getLanguage function will be "fr". French is the first language specified by the browser and is supported by the server. If not matching language was found, the first language specified by the server will be used.

If no list of supported languages was specified when instantiating this class, the class will assume that only English is supported.

15.4 YDObjectUtil

The YDObjectUtil module provides you with some class and object related static functions. The following functions are available:

- **isSubClass:** This function will check if the specified object is subclassed from the specified class name or is an instance of the specified classgetAncestors.
- **getAncestors:** This function will list you the parent classes of a specific class.
- **failOnMissingMethod:** This function will raise a fatal exception if the specified object doens't contain the specified function.
- **serialize:** This function returns a gzipped stream containing a byte-stream representation of an object that can be stored anywhere.
- **unserialize:** This function takes a single gzipped serialized variable (the output from the YDObjectUtil::serialize() function) and converts it back into a PHP value.

***Note**: all these functions have to be called statically and thus do not require an instance of the YDObjectUtil class.*

15.5 YDTimer

The YDTimer class offers you with a very easy timer class. It will time the execution of whatever you want. The class starts counting when you instantiate it, will return you the elapsed time when you call it's getElapsed function. The result is returned in seconds.

Let's have a look at a code example:

```
$timer = new YDTimer();
$elapsed = $timer->getElapsed();
```

This $elapsed variable will now contain the time it took from the instantiation of YDTimer class till the function call to the getElapsed function.

This class is very handing for timing certains parts of your application. You can create multiple instances of the YDTimer class which takes care of timing once specific part.

15.6 YDBBCode

Needs to be written

16 Best practices

This chapter will give you an overview of common patterns and best practices for using the Yellow Duck Framework.

Needs to be written.