Seagull PHP Framework

Werner M. Krauß

When using PHP for your daily business, you'll always have the same problems to solve: user authentication, DB queries, validation of user-input you get via $_GET or $_POST. Why not make a framework with reusable code for all such jobs? You don't have to reinvent the wheel every time, do you?

This article is not to explain the basic structure of frameworks; if you need information about that, please have a look at the former issue of this magazine [2/2005]. Instead it will show you how to build an application using the Seagull PHP Framework.

What is Seagull?

Seagull (*http://seagull.phpkitchen.com*) is an OOP application framework mostly based on PEAR classes and licensed under BSD license (*http://seagull.phpkitchen.com/LICENSE.txt*). It is easy to install and uses good coding practices, design patterns, database abstraction and separation of content and presentation. It's completely modular and new features can easily be added to the system. The developer community also pays considerable attention to maintaining a cleanly structured codebase, observing security guidelines and respecting web standards like XHTML and CSS.

The framework offers some ready-to-use modules like *Publisher* – a lightweight CMS, a *Contact us* module, a *Guestbook*, a module for setting up a list of FAQs (*Frequently Asked Questions*) and even a *shopping cart*.

The Seagull Framework consists of:

- base framework: The framework itself is made up of a set of base classes organized according to the MVC design pattern that take care of permissions, authentication, sessions, input/output and database abstraction,

What you should know...

Seagull will work best for you if you have at least 1–2 years experience and a good exposure to object oriented concepts, PEAR and design patterns.

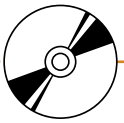
What you will learn...

We show you how to build a huge application using the Seagull PHP Framework.

On the net

1. *http://seagull.phpkitchen.com/* – Seagull Home Page
2. *http://seagull.phpkitchen.com/docs/wakka.php?wakka=Talks&v=235* – Seagull Talks

On the CD



On CD you will find Seagull Framework and all listings from article.

- components: Each generalised area of functionality comes in the form of a component; your business requirements may be matched to components, e.g. modules, blocks, items.
- libraries: Most task-specific functionality comes from libraries, quite often from PEAR [*http://pear.php.net*], that can be independently updated when upgrades/improvements are available,
- entities/entity managers: Each object in the application such as Member, Group, Property, Document, Article, etc. is represented as an entity. You can quickly prototype entities using the tools Seagull provides so that skeleton classes are created and updated automatically.

System Architecture

Before starting with Seagull, let's have a look at the directories it contains. You can see the complete structure in Figure 2. In detail:

- [*Root directory*]: *init.php* and *constants.php,*
- *etc/*: et cetera: basic configuration files, sql-files, etc...,
- *lib/*: libraries (Seagull, PEAR and other) and data files like arrays for country names or languages in *lib/data/,*
- *modules/*: each module has its own subdirectory,
- *var/*: for all temporary data like compiled templates, DB_DataObject entities, logfiles and sessions. This directory has to be writeable by the webserver,
- *www/*: application webroot which contains front controller script, themes and javascript. Only this directory should be viewable to the web. Otherwise make sure to protect the rest with *.htaccess* files.

Basic Classes

Basic tasks like database connection, sending emails or formatting output are done using the Seagull Base Classes, contained in */lib/SGL/*. You find the complete API documentation at the project homepage (*http://seagull.phpkitchen.com/apidocs/*) for browsing or download.

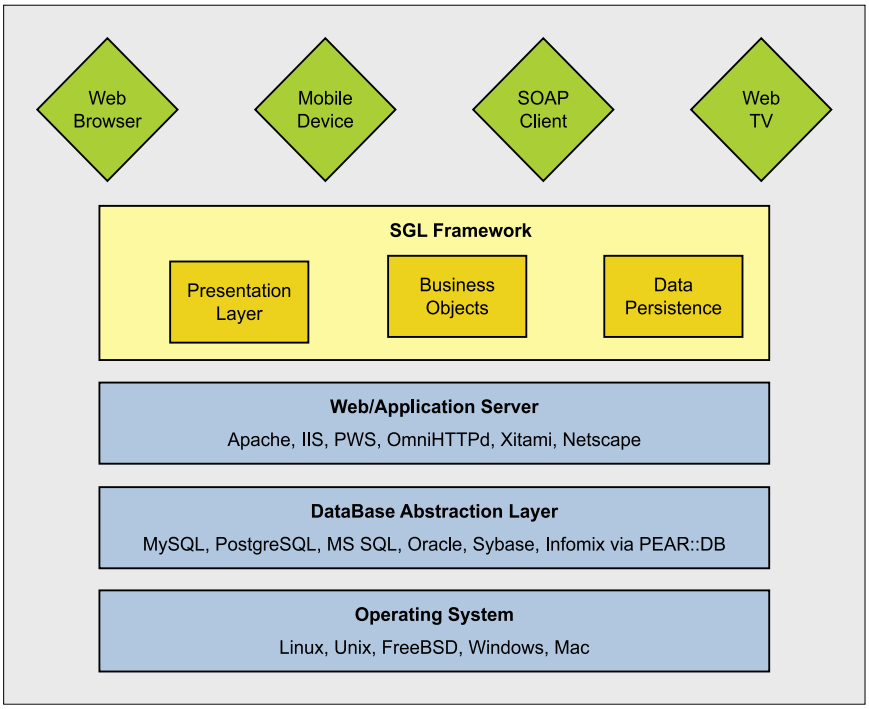


Figure 1. *System Overview*

Templates and Themes

Seagull uses templates and themes for separating data from layout.

By default the PEAR package HTML_Template_Flexy (*http://pear.php.net/package/HTML_Template_Flexy*) is used. Flexy compiles all HTML templates into PHP scripts that are never touched by the developer, so you don't have to bother about parsing template files at every request.

A theme in turn is a collection of folders placed in *www/themes/*. Each folder contains the HTML templates for the module it represents.

System reqirements

Seagull operates almost identically on Windows and Unix platforms. At the moment MySQL, PostgreSQL and Oracle are supported, but theoretically all databases supported by PEAR::DB (e.g. MSSql, SQLite or ODBC) can be used without problems.

History

The project was opened in 2001 by Demian Turner who wanted to create a simple and stable framework using some popular design patterns for his project. Since October 2003 it's been hosted on SourceForge (*http://www.sourceforge.net/projects/seagull/*).

Why use Seagull?

There are lots of PHP frameworks out there, so why should you consider using Seagull?

- It's an Open Source project with support from a large developer community.
- Most of the libraries it uses are PEAR libraries, which are recognised as very good quality and are well known by many PHP developers.
- It uses common design patterns such as MVC and validate/process/display workflow.
- It's licensed under the BSD license, meaning it can be used for commercial, closed source projects without the requirement that you must contribute all your work to the project.
- It offers a front controller with search-engine-friendly URLs like *http://www.example.com/index.php/contactus/action/list/*

Model-View-Controller

The MVC pattern (Model View Controller) stipulates a clean separation of processor logic (Controller), manipulating data (Model) and presentation (View).
A good discussion of this pattern in the context of PHP can be found in a former issue [2/2005] of PHP Solutions magazine

The journey begins...

Let's say you want to create a portal site. The users should add content pages and share documents. This is quite an easy job using Seagull PHP Framework. You have to install the framework, import the users, manage permissions, use some modules (like FAQ or the CMS module for sharing articles) and modify the look and feel so it fits with your corporate identity. In addition you can extend the functionality the framework provides by creating a new module, e.g. a simple event calendar.

In the following sections you will see how this is done.

First Steps

Now log in using the user/password combination *admin/admin*. First please change this password for security reasons. To do so, after logging in as admin, go to *My Account* and click the button *change password*.

Configuration & Preferences

The *admin* can see the items *Configuration* and *Modules* along with the normal menu structure. *Configuration* is for global configuration items like site name, database connection details, etc. By clicking *Modules* you can see all installed modules. The ones with blue background are configurable or have functions for administration. Every module has its own *conf.ini* file in its module directory, however this is not currently editable using the browser.

Figure 2. *Directory structure*

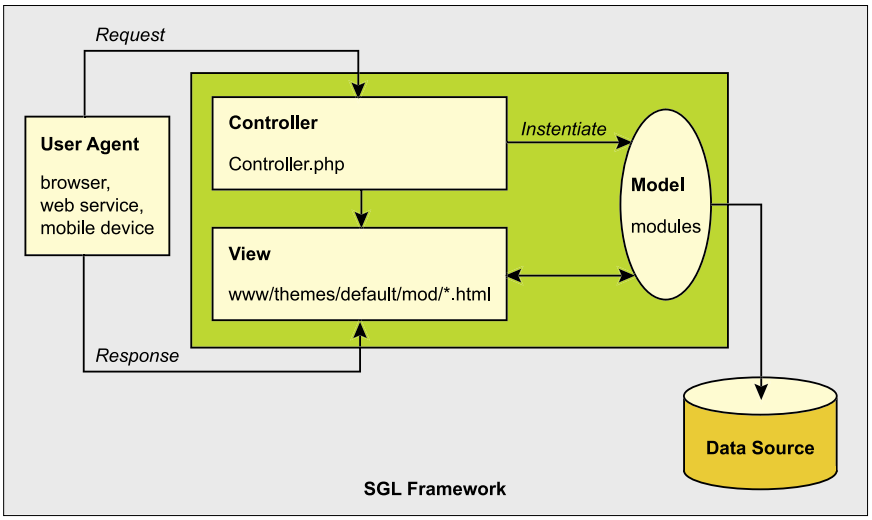


Figure 3. *SGL Framework – MVC*

After installing Seagull you will see the user interface in the English language, but you can change this using the preferences. Unlike configuration, which applies to all users, preferences can also be adapted by every user by going to *My Account -> Preferences.* Preferences include items such as language, time zone or how many results per page should be shown, etc.

To change the language settings, click *Modules* and then the module *Users and Security*. Now you are inside the user manager and see a second menu on top: click the rightmost item, titled *prefs,* to reach the *Preference Manager*.

Edit the value *language* and input *de-iso-8859-1* to switch the user interface language to German. This value is put together from the short term for german "de" and the used character set "iso-8859-1" (Latin-1). At the moment Seagull supports 12 languages including German, French, Spanish, Polish, Czech, and even Chinese. You can see a full list of the languages and their values in the docs (*http://seagull.phpkitchen.com/docs/wakka.php?wakka=Localisation*).

Register new users

New users can either register themselves via the registration form contained in *Register Now* or be added by the admin using *Users and Security -> add user*. A few additional features are available to *admin*

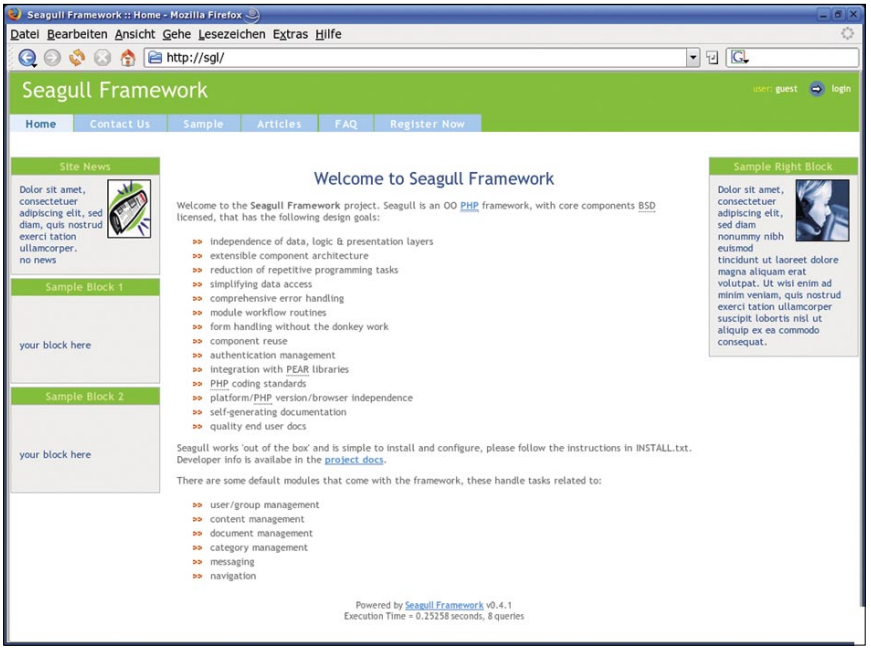


Figure 4. *First view after installation*

such as the ability to set the user's role, his/her default organisation (see box), and whether they are active or not. Also, admins can reset the password for any user, and optionally have them notified by email of the new password. Finally, admins can tweak the user's permissions on an individual basis, customising as necessary.

By default a new user has to be activated by *admin*, however this can be changed in the user module's *conf.ini* file.

Rights Management

Seagull has a fine grained rights management system. Every method of a module's page manager has its own permission. The naming convention is `modulemgr_method`. For the module "FAQ", for example, there are perms like `faqmgr_list` for listing faqs, `faqmgr_add` and `faqmgr_insert` for adding entries etc. It's also possible to create a module wide permission for all functions, useful for admins. In this case the permission's name is just the manager name, in this case `faqmgr`.

What if more users should get the same rights, maybe because they are working in the same area? Adding many permissions every time by hand is a time consuming and boring work. So why not group permissions together into roles? Just create the role *FAQAdmin*, add the perms to it, e.g. `faqmgr` for whole module and now every user who is working in this area can get the perms he needs by just assigning the role to them.

But be careful: Seagull doesn't save the role – but the rights the role contains – on a per user basis. If you want to add new perms to the role existing users don't have the new rights automatically. You first have to sync each user's permission. In return you also can add or delete some permissions that are (not) contained in the original role. This is done with a few mouse clicks using some comfortable functions Seagull offers for this to ease your work.

Practice: Define some Roles / Perms to Roles / group Users by Roles

Adding a new role is very easy: You can either start with a blank one by choosing *add role* and enter name and description or duplicate an existing one and edit the name by clicking on the role's edit button. Now you can see the newly created role ready for filling with permissions. Click the *change* button and you see a selection of possible roles to add on the left and the list of roles it contains on the right.

After creating some roles you can now assign this role to existing users by editing the user's data.

Let's add some users to the system using the *User Import Manager* (see box *Importing users*). That's a very easy way of adding more users at once. Unfortunately, at the moment of writing this article, one can only add username, first and last name and the email address. So you have to add other required data like address or security questions for retrieving a new password by hand. This function will be improved in the future.

Publisher: Intro

Now we go further and have a look at another module, the *Publisher* module. This is, as the name suggests, for publishing content, a CMS (*Content Management System*). You can manage articles and upload documents and categories using this module.

After accessing the module's administration interface you can see three sections on the top of the page: *Categories*, *Documents* and *Articles*.

Categories offers you the possibility to have different category trees for each

Virtual Host

It's better and more secure to install Seagull on a separate server or virtual host. Point your webroot to `[path-to-seagull]/www/`, so that these are the only files available to the public; config-files with sensitive data, libraries and uploaded files that should only be viewed by authenticated users, will be protected.

Installation

Installing Seagull is very easy. You just need a webserver (like Apache or IIS), PHP (version 4.1 or newer – PHP5 works, too), a database (e.g. MySQL, PostreSQL, Oracle) and you can start:

- Download the most recent version from the project homepage (*http://seagull.phpkitchen.com*) and unpack it in your webroot directory.
- The webserver needs write permission to the *var/* directory. On a Unix system you can do this by issuing `chown -R nobody [web-doc-root]/seagull/var` at the command line.
- Now you can load Seagull in your browser (e.g. by typing *http://localhost/seagull/www*). You have to enter the details like Host, User, Password etc. of an empty database created prior to the installation.
- After entering all values, press the button *Create schema and load default data*. The installer will create the database schema for you and load the default data. This will take some seconds depending on the speed of your machine.
- Once you have received messages about the schema being created and default data loaded, hit the *launch Seagull* link and you can see the fresh installed framework as shown in Figure 4.

It is recommended to install the framework on a virtual host. See box *Virtual Host* for details.

Installation using PEAR

This method is the easiest and fastest way to get Seagull up and running. There are a few requirements:

- You must be running a recent version of PHP 4.3.4+ with the base PEAR packages installed.
- You must set the `pear data _ dir` to your webroot or point it to anywhere on your filesystem and subsequently create a Virtual Host to expose the www directory. This is done with the switch `-d data _ dir=/path/to/data/dir`. To view your current settings use `pear config-show`.
- Your preferred package state must be set to beta, as the current state of the Seagull project is 'beta'. This is done with the switch `-d preferred _ state=beta`.

So to install Seagull, the following commandline sequence is necessary (on one line):

```
$ pear -d data_dir=/path/to/web/root -d preferred_state=beta install --onlyreqdeps
                    http://kent.dl.sourceforge.net/sourceforge/seagull/seagull-
                    0.4.0.tgz
```

Once you have performed the installation with the PEAR package manager, don't forget to revert your PEAR configuration settings to their original state. Complete installation as

module. In addition you can grant or deny access to a category and all its content depending on the user's role.

The section *Documents* is for uploading documents. You can choose category, name and description.

Last but not least, you can easily add content to your site using the *Article Manager*. Seagull allows you to create three types of articles, internally called items. This is easily customisable, but only the following three types are shipped with the distribution:

- *HTML Article*: This article will show up in the category hierarchy. This can be useful for intranet applications, or if you have a large body of work that needs to be categorised.
- *Static HTML Article*: If you want to make standalone pages that will be linked to by their own tab in navigation and don't show up in categories.
- *News Item*: These will appear in the left-hand column in the *Site News* block.

Publisher Practice: Make Static Page

For better understanding, let's create a static page that will have its own tab in navigation:

Please go, as admin, to the *Publisher* module and launch the *Article Manager* by clicking on the *Articles* button. Choose *Static HTML Article* in the select box and hit *new article*. Now you see the form for adding content to your article. Choose start and expiry dates, the title and the content. If you use a Mozilla-based browser or Internet Explorer you can simply edit the layout of your article using the WYSIWIG editor, as shown in Image 5.

Importing users

To create a site with many users you also have the possibility to use the import feature (e.g. using Thunderbird's address book as your source). To do so, first upload the CSV file using the Publisher module (choose Documents), then go to the user module, click on *import users*, choose the file you uploaded, and an organisation and role, and then click *process file*. Easy, isn't it?
After that you have to activate the users and add some more information about them.
A specification about the structure of the file is listed in the Seagull documentation (*http://seagull.phpkitchen.com/docs/*).

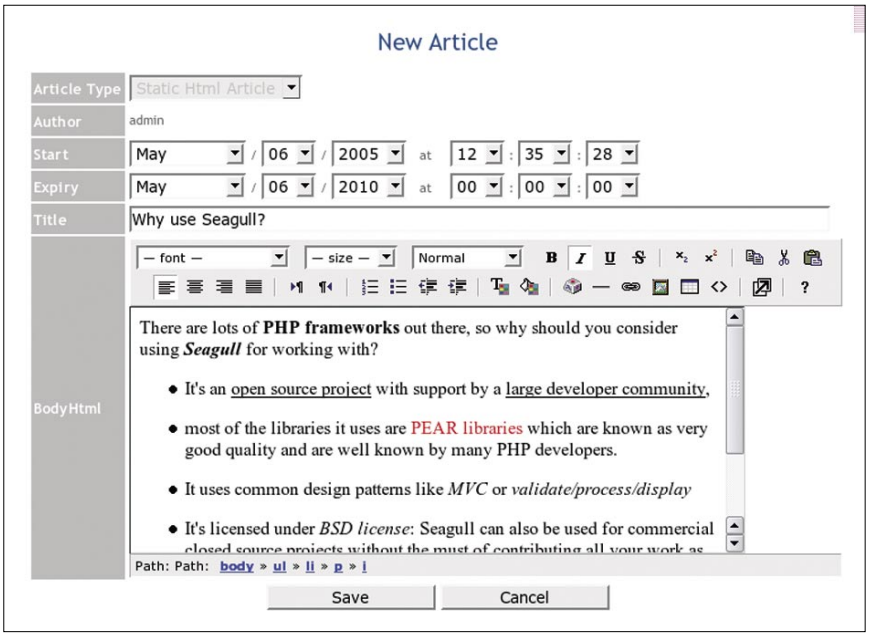


Figure 5. *Adding a static HTML article*

After saving, you see your new article in a list and marked *For Approval*. To make it visible to the audience, click *approve* and then *publish*. This has to be done, also, after every time you edit the item again.

Create Navigation

Now create the navigation tab for the static HTML article you wrote before:

Go to the *Page Manager,* and hit the *new section* button. Enter the page title, choose *static articles* and your article form the select boxes. Then set the *Status* to active and choose which roles can see this article. After saving you can see a new navigation tab for your article, as shown in Image 6.

You can just as easily add navigation items for your modules. Just select *dynamic pages* instead of *static pages* and you'll see a form where you can select the *module*, the *manager class* and the *action* method. The value for the FAQ module e.g. is *module: faq*, *manager: faqMgr*, *action: none*, which is the default action defined in the module's class.

User grouping using organisations

Users can be grouped into organisations, arbitrary groups of users, typically useful when they work for the same company or department. The *org* functionality is optional and can be turned off in the user modules *conf.ini* file, i.e., some smaller, simpler installs of Seagull may not require user groupings.
Organisations are currently designed with a nested set hierarchy, so for example the following situation could be accommodated quite easily:

- company ABC has many suppliers,
- it also has a number of distributors,
- some suppliers are also clients, or have clients of their own,
- some clients have resellers under them,

In such a case the structure of organisations would be

- company ABC->supplier DEF->client GHI->reseller JKL.

Therefore, using a nested set `getBranch()` operation, you could trace the supply chain from company to reseller with ease, easing the amount of code required for *admin* operations. Using the Organisation manager you can set the organisation type to further aid grouping. In the above scenario this would equate to company, supplier, client, reseller respectively.

In addition, each organisation has the concept of a default role and default preferences, so each user created in the organisation would assume the defaults assigned. This can be useful when managing a large number of users in the system.

Seagull uses a themeable, driver-based navigation system. At the moment the only driver available is called *SimpleNav*, but it should be easy to create e.g. a *PEAR::HTML_Menu* based driver.

You can find several stylesheets for horizontal and vertical navigation shipped with Seagull. They are all based on CSS, no JavaScript is needed. To switch the themes click on the *modify nav bar appearance* button at the top of the *Page Manager* screen. This brings you to a preview page where you can edit the navigation themes on a per-role basis.

It's easy to create new themes for *SimpleNav*, because it only outputs an unordered list (`<ul>`). So all you have to do is to create a new CSS file for it.

Using Blocks

Using the *Block* module you can easily add content blocks to your site. Some of the blocks shipped with Seagull are e.g. for site news, login, random messages or category navigation.

You reach the Block Manager via *Manage Modules ->Blocks*. It allows you to add and edit entries and lets you change the order of the displayed blocks. New block types can be added by putting a new class in the directory *modules/block/classes/blocks.*

Figure 6. *The finished static HTML article*

When editing the block information, the *Name* parameter of the block corresponds to the PHP file you want to use for content rendering.

If you want to e.g. add the login block delivered with the Seagull distribution you have to activate it in the administration screen of the block module. You can edit the title of the Block, the CSS class for displaying it, whether it's shown on the left or right side and in which sections, the items you defined in the navigation module before, the block should appear. And – of course – you can edit the order. Later I will show how to code a new block for a new module.

Coding Standards

Seagull follows the PEAR coding Standards (*http://pear.php.net/manual/en/standards.php*) as much as possible. Some highlights you may not be used to are:

PHP

- all variables follow the bumpy caps notation, i.e. `$thisIsMyVariable` and not `$this_is_my_variable`,
- all class names start with a capital letter and are then followed with bumpy caps,
- constants are capitalised with multiple words separated by underscores,
- things like 'true' and 'null' are always lower case,
- the majority of classes are commented with PHPdoc notation,
- require_once and include_once do not use parentheses as they are not functions,
- code is commented as much as possible for the benefit of others,
- php short tags, i.e. `<?`, are not used as they are incompatible with XML processing. use `<?php` instead,
- descriptive variable names are used whenever possible to increase readability – descriptive but not verbose like you often seen in Java,
- double quotes, i.e. “string”, are not used unless it is unavoidable as they incur extra overhead by required PHP to parse their string contents,
- lines of code must not be longer than 80 characters.

SQL

There are also a few rules you should follow when designing a new table:

- all table and column names in lowercase,
- primary key name is: `tablename_id`,
- don't use MySQL's autoincrement; use sequences provided by PEAR::DB instead,
- entities must be referred to in the singular, not plural, so use 'book' instead of 'books',
- table name relations must be separated by underscores, i.e., user_preference.

Create a new module: event manager

Well, so far we have made a pretty interesting site. But what if we need functionality that is not shipped with Seagull? It's quite easy to create new modules for this framework, although you have to learn the structure and class names, but that's true for every framework. Once you know how it works, it's very easy and fast!

Our module should be a simple event calendar, containing functions to *add/edit/delete* events and for listing all events.

Module Skeleton Generator

How to start creating a new module for Seagull? You can either choose a module that is similar to your needs and modify it, or begin with a new one. In the former case, just copy the old module files and directories and rename them properly. In the latter you can either start from scratch or use the *Module Skeleton Generator* which is part of the *Maintenance* module.

This function is not meant for creating the whole module; that is not possible yet. But, using a simple form, it creates the basic structure, including directories for classes, translations and templates, and some basic files for your new module.

To use it, you have to give the webserver write permission to the *modules/* and *www/themes/default/* directories where the files are created. Then go to the maintenance module and fill out the form *Create a module*. You can choose the module name, manager name (name of the PHP class) and what functions it should have.

Let's call the new module *Event* and the manager class *EventMgr.* You see the generated class in Listing 1.

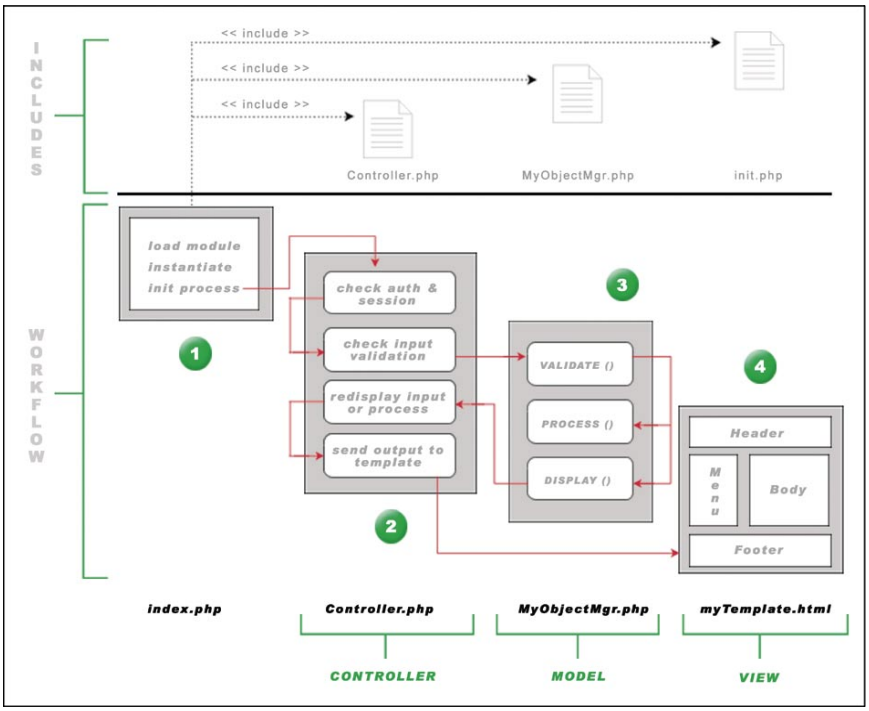


Figure 7. *Workflow*

The class, called `EventMgr`, is extended from the class `SGL_Manager` which prototypes a basic *validate/process/display* workflow (see box *Validate – Process – Display*). The new module just has to have the additional functionality implemented. In the constructor `EventMgr()` you see some variables defined: first the module name, page title and default Flexy template to use for generating HTML output. Next the array `$this->_aActionsMapping`, which maps actions strings the module receives via `$_GET` or `$_POST` to the right methods. Some actions have more methods mapped, e.g.: `'insert' => array('insert', 'redirectToDefault')`. In this case the data is first inserted to the database but no output is generated. Then the script redirects to the default action, here `list()` for listing the data. You can define as many actions to perform as you like.

The method `validate($req, &$input)` is for validating the input variables it gets from the request object `$req` and writes it to the `$input` object. The methods for adding, listing etc. are empty at the moment, but you can see that every action method has the `$input` and `$output` objects passed by reference.

Seagull also needs some other files to run the module: the `modules/event/conf.ini` where you can define module configuration as shown in Listing 2 and the language files in `modules/event/lang/` which are just PHP files containing an array of translated language strings (see Listing 3).

SQL

Now let's create a database table for the module. It should contain the event's id, start date, expiry date, creation and last update dates, title and also a short and long description. See Listing 4 for the complete SQL file.

The next step is to create the *DB_DataObject* entity for the new table. Seagull allows developers the option of using the PEAR::DB_DataObject (*http://pear.php.net/manual/en/package.database.db-dataobject.php*) package in their modules, an object-relational mapping engine that makes inserts, updates and other interaction with the database easier. We'll use it here.

Go to the *Maintenance* module and hit the *Rebuild DataObjects Now* button. After that you can see the newly created file *var/cache/entities/Event.php*. This file contains a class that offers some very simple functions for interacting with your newly created database table, called *DataObjects_Event*. To use the new entity object, you need to include it into your module's script. Add this in front of the class definition:

```
require_once SGL_ENT_DIR . '/Event.php';
```

Create methods for listing and adding/editing

OK, the very basic structures are made, now we can fill this skeleton with actions. Let's start with the `list()` method as shown in Listing 5.

It's quite easy to create this function. After sending a message to the log and defining the page title, the *DB_DataObject* object `$eventList` is created. We want our resultset ordered by expiry date, so we add an orderBy statement to the DataObject: `$eventList->orderBy('date_expire');`.

Next we tell it to show only up-to-date events. For this we have to create a SQL datetime string using a function the framework provides: `SGL_Date::arrayToString()`. `$eventList->find();` controls whether the database returns anything: if yes, `$eventList->fetch()` selects all events we want to list.

All we have to do after that is copy from the object to an array for outputting, using `clone()`. In this loop we can also check if start and expiry date are the same. If they are, only one date should be shown. At last we pass the result-array to the output-object so we can use it in the template (`$output->results = $aEvents;`) where it's called `{results}`.

Validate – Process – Display

Seagull provides a validate/process/display workflow, which simply means that all data that passes through the system must be filtered by the following methods:

- validate: the raw `$_REQUEST` is passed in to this method, validations are performed, and acceptable data is mapped to an `$input` object
- process: if all data is valid, the `$input` object is passed to the process method, which redirects to the relevant action method. Once data has been manipulated, it is mapped to an `$output` object
- if one or more validations have failed, all data is deemed to be invalid, and is passed directly to the `display()` method with appropriate error messages, i.e., to be presented back to the user for correction
- display: this takes the data, whether valid or invalid, adds a few system properties like execution time, etc., and sends it to the template engine for rendering into HTML.

If we tried to run the script now, we would get an error, that no template can be found – and yes, there are no files in *www/themes/default/event/*, where this module's templates are to be stored. So let's start writing the first template for the list-action, *eventList.html* as seen in Listing 6.

That's all? Yes. Your template gets included into a master template, so you only have to define what your module does. The `{if:results}` statement determines whether there are results to show. If not, you just get a little failure notice - *No Events found* - translated into your language using the language files (see above).

Let's have a closer look at the line `<tr class="{switchRowClass()}" flexy:foreach="results,key,valueObj">`: This is a simple foreach loop for showing the result sets, one by one, similar to `foreach($results as $key => $valueObj)`. The background of each table row is altered by the function `switchRowClass()`.

As there are still no events to show, the next step is to add functionality for adding events to the database as shown in Listing 7.

Adding stuff to the database is done in two steps: first display the form, then get the user's input, validate it and finally add it to the database. For this reason we have two actions for that: `add()` and `insert()`.

In `add()` we just define some variables and pass them to the template. Have a closer look at how the date selectors are made: Seagull offers some very handy functions for that: First define an array with values for day month and year and then pass it to the function `SGL_Output::showDateSelector()`. The counterpart is `SGL_Date::arrayToString()`, and we use it inside the `insert()` action when the user submits the data.

To make validation work correctly we also have to add some lines of code to the method `validate()`. First we define the values we want to get from the user input, then we check if expiry date is greater than the start date and if a title is set. If an error occurs, then an error message is thrown and the user sees the form again, including the values he typed in before. You can see the whole validation method

Listing 1. *Output of Module Skeleton Generator*

```
<?php
/**

 * Type your class description here ...
 * @package event
 * ...
 */
class EventMgr extends SGL_Manager
{
    function EventMgr()
    {
        SGL::logMessage(null, PEAR_LOG_DEBUG);
        $this->module       = 'event';
        $this->pageTitle    = 'Event Manager';
        $this->template     = 'eventList.html';

        $this->_aActionsMapping =  array(
            'add'       => array('add'),
            'insert'    => array('insert', 'redirectToDefault'),
            'edit'      => array('edit'),
            'update'    => array('update', 'redirectToDefault'),
            'delete'    => array('delete', 'redirectToDefault'),
            'list'      => array('list'),
        );
    }

    function validate($req, &$input)
    {
        SGL::logMessage(null, PEAR_LOG_DEBUG);
        $this->validated    = true;
        $input->error       = array();
        $input->pageTitle   = $this->pageTitle;
        $input->masterTemplate = $this->masterTemplate;
        $input->template    = $this->template;
        $input->action      = ($req->get('action')) ? $req->get('action') :
                    'list';
        $input->aDelete     = $req->get('frmDelete');
        $input->submit      = $req->get('submitted');

        //  if errors have occured
        if (isset($aErrors) && count($aErrors)) {
            SGL::raiseMsg('Please fill in the indicated fields');
            $input->error = $aErrors;
            $this->validated = false;
        }
    }

    function _add(&$input, &$output)
    {
        SGL::logMessage(null, PEAR_LOG_DEBUG);
    }
[...]
}
?>
```

Listing 2. */modules/event/conf.ini*

```
[EventMgr]
requiresAuth    = false
```

Listing 3. *Sample of language file*

```
<?php

$words = array('Events' => 'Events',
/*  Events */
  'Event Manager' => 'Event Manager',
  'Event Manager :: Browse' =>
    'Event Manager :: Browse',
  'Event ID'   => 'ID',
  'Date created' => 'Date created',
  'Start' => 'from',
  'Expiry' => 'to',
  'Short Description' =>
     'Short Description',
  'Long Description' =>
    'Long Description',
/*  list */
  'No Events found' => 'No Events found');
?>
```

Listing 4. *SQL-schema for event module*

```
CREATE TABLE event (
   event_id int(11) NOT NULL default '0',

   date_start datetime default NULL,
   date_expire datetime default NULL,
   date_created datetime default NULL,
   date_last_updated datetime default NULL,
   title varchar(255) default NULL,
   text text,
   PRIMARY KEY  (event_id)
) ;
```

Listing 5. *method list()*

```
function _list(&$input, &$output)

{
   SGL::logMessage(null, PEAR_LOG_DEBUG);
   $output->pageTitle = 'Events';
   $eventList = & new DataObjects_Event();
   $eventList->orderBy('date_expire');

   //where clause for getting only not expired events
   list($day, $month, $year) = explode('/', date('d/m/Y'));

   //initialise input array with current date/time
   $aDate = array( 'day' => $day, 'month' => $month, 'year' => $year);
   $eventList->whereAdd('date_expire >= \''. SGL_Date::arrayToString($aDate).'\
                        '');
   $result = $eventList->find();
   $aEvents = array();

   if ($result > 0) {
      while ($eventList->fetch()) {
         //check if start date and expiry date are the same.
         //if yes show only one date in template

      if ($eventList->date_start == $eventList->date_expire)
         $eventList->date_start = '';
         $aEvents[] = clone($eventList);
        }
    }
    $output->results = $aEvents;
}
```

Listing 6. *Template for listing all events: eventList.html*

```
<h1 class="pageTitle">{translate(pageTitle)}</h1>
<span flexy:if="msgGet()">{msgGet()}</span>
<table class="wide">
{if:results}
    <tr class="{switchRowClass()}" flexy:foreach="results,key,valueObj">
        <td><em>{if:valueObj.date_start}{formatDate(valueObj.date_start)} -
                        {end:}{formatDate(valueObj.date_expire)}: </em>
        <strong>{valueObj.title}</strong><br />
        {valueObj.text}
        </td>
    </tr>
    {else:}
    <tr>
        <td>{translate(#No Events found#)}.</td>
    </tr>
    {end:}
</table>
```

in Listing 8.

The methods for editing, updating and deleting use the same principles.

But how to reach these methods? Best is to make an additional interface for administration. A simple switch in `list()` decides if the admin template is shown or not (Listing 9).

The admin template contains a table where all events are shown and links for adding, editing and deleting are provided. The URLs are created using the `{makeUrl(#add#,#event#)}` function which generates the correct URL used by the front controller.

Registration of the new module

Next we have to add the new module to the module manager, so we go to *Modules -> Manage* and hit the *Add a module* button. We have to fill out the form with name ("event"), title ("A simple event manager"), configurable ("yes"), description (this text will be used in the *Module Manager* to describe this module), the admin URI and an icon filename. If you don't have an icon for your module yet just use an existing icon like *faqs.png*.

After clicking the *Add* button we see again the *Module Manager* screen. Our new module has been added at the end of the list of modules. Clicking on this listing leads us to the admin section of our newly created module.

Add rights / also to roles / sync users

If a user other than root wants see the events list, he gets the error *You do not have the required perms to perform this action*. What's wrong? We have not added the new perms for this module to the system. This is very easy to do: After selecting the *manage permissions* button from *User & Security* module we hit the *detect and add* button, select all permissions for our module *event* and add them with one more click to the permission list.

Now we can add the new permissions to roles. *Guest* e.g. will get *eventmgr_list*, so he can view all events. This will work after you logout or close the browser, because perms are stored in a cookie which lasts the lifetime of the browser.

Members should also be able to add and update new events. But just adding the permissions to the role doesn't give

existing users this rights. What's wrong?

As permissions are not retroactive we have to sync the current roles, too - but don't hesitate: it's done with a few mouse clicks using the *sync perms with role* function.

You can select the users you want to sync, if you want to sync them with the current or another role and if you want to add missing perms, delete extra perms or both. Using this tool can help you a lot keeping the rights management up to date.

Block for new module

Last but not least we can add a custom block for our new event module. For this we have to add a new block class to *modules/block/classes/blocks/* as shown in Listing 10.

Then we have to add it to Block Manager's *New Block* form: fill in name (EventBlock) and title (Events), which sections it should appear in and its position. Then hit 'submit' and the block details will be saved, but by default the block will be deactivated. To activate it edit it again and check the *activated* checkbox. Now you will see the new block.

Change Look and Feel: Themes

But this site still looks like a fresh Seagull install! No problem, this is very easy to solve. You just have to change two or three files and soon it will look very different.

To create your own look and feel you need to create your own theme by creating a new directory in the seagull/www/themes directory, e.g. seagull/www/themes/myTheme. Copy the default module's templates from *www/themes/default/default/* into your new

Using logs during development

It's quite useful to watch the log Seagull creates when creating and testing a new module. On *nix systems you can use

```
tail -f [path-to-seagull]
  /var/log/php_log.txt
```

The option `-f` makes the tail command listening to the log file, so it can always show the latest lines in your console window.

Listing 7. *Add / Insert Methods*

```
function _add(&$input, &$output){
    SGL::logMessage(null, PEAR_LOG_DEBUG);
    $output->template = 'eventEdit.html';
    $output->action   = 'insert';
    $output->todaysDate = SGL::getTime(); //make start and expiry date selectors
    list($day, $month, $year, $hour, $minute, $second) = explode('/', date('d/m/Y/H/i/s'));
    // initialise input array with current date/time
    $aDate = array( 'day' => $day, 'month' => $month, 'year' => $year);
    $output->dateSelectorStart = SGL_Output::showDateSelector($aDate, 'frmStartDate', false);
    $output->dateSelectorExpiry =SGL_Output::showDateSelector($aDate, 'frmExpiryDate', false);
}
function _insert(&$input, &$output){
    SGL::logMessage(null, PEAR_LOG_DEBUG);
    $event = & new DataObjects_Event(); // insert record
    $event->setFrom($input->event);
    $dbh = $event->getDatabaseConnection();
    $event->event_id = $dbh->nextId('event');
    $event->date_created = $event->date_created = SGL::getTime(true);
    $event->date_start = SGL_Date::arrayToString($input->aStartDate);
    $event->date_expire = SGL_Date::arrayToString($input->aExpiryDate);
    $success = $event->insert();
    if ($success) {SGL::raiseMsg('event saved successfully');
    } else {
      SGL::raiseError('There was a problem inserting the record', SGL_ERROR_NOAFFECTEDROWS);
    }
}
```

Listing 8. *validation of user input*

```
function validate($req, &$input){
    SGL::logMessage(null, PEAR_LOG_DEBUG);
    $this->validated    = true;
    $input->error       = array();
    $input->pageTitle   = $this->pageTitle;
    $input->masterTemplate = $this->masterTemplate;
    $input->template    = $this->template;
    //define some variables we get from user input
    $input->action      = ($req->get('action')) ? $req->get('action') : 'list';
    $input->aDelete     = $req->get('frmDelete');
    $input->submit      = $req->get('submitted');
    $input->event       = (object)$req->get('event');
    $input->aStartDate  = $req->get('frmStartDate');
    $input->aExpiryDate = $req->get('frmExpiryDate');
    if ($input->submit) {  //check if everything is ok
       if (empty($input->event->title)) { //title is obligatory
          $aErrors['title'] = 'Please fill in a title';
       }
       //generate timestamps first
       $startDate = mktime(0,0,0,$input->aStartDate['month'],
          $input->aStartDate['day'],$input->aStartDate['year']);
       $expiryDate = mktime(0,0,0,$input->aExpiryDate['month'],
          $input->aExpiryDate['day'],$input->aExpiryDate['year']);
       if ($startDate > $expiryDate) {
          $aErrors['date_expiry'] = 'Expiry date has to be after start date';
       }
    }
      if (isset($aErrors) && count($aErrors)) { // if errors have occured
       SGL::raiseMsg('Please fill in the indicated fields');
       $input->error = $aErrors;
       $input->template = 'eventEdit.html';
       $input->dateSelectorStart = SGL_Output::showDateSelector($input->aStartDate,
         'frmStartDate', false);
       $input->dateSelectorExpiry = SGL_Output::showDateSelector($input->aExpiryDate,
         'frmExpiryDate', false);
       $this->validated = false;
    }
}
```

Listing 9. *Simple switch in list() decides if the admin template is shown or not*

```
if (SGL_HTTP_Session::getUserType() == SGL_ADMIN) {

   $output->template = 'eventListAdmin.html';
   $output->pageTitle = $this->pageTitle . ' :: Browse';
} else {
   $output->pageTitle = 'Events';
}
```

Listing 10. *Block for event*

```
<?php

require_once SGL_ENT_DIR . '/Event.php';
/**
 * Event block.
 *
 * @package event
 * @author  Werner M. Krauß <werner.krauss@hallstatt.net>
 * @version 1.0
 * @since   PHP 4.1
 */
class EventBlock
{
   var $webRoot = SGL_BASE_URL;

   function init()
   {
      SGL::logMessage(null, PEAR_LOG_DEBUG);
      return $this->getBlockContent();
   }

   function getBlockContent()
   {
      SGL::logMessage(null, PEAR_LOG_DEBUG);
      $theme = $_SESSION['aPrefs']['theme'];
      $aResult = $this->retrieveAll();
      $eventList = $this->toHtml($aResult);
      return $eventList;
   }

    function retrieveAll()
    {
       $eventList = & new DataObjects_Event();
       $eventList->orderBy('date_expire');
       $eventList->limit(5);

       //where clause for getting only not expired events
       list($day, $month, $year) = explode('/', date('d/m/Y'));
       //  initialise input array with current date/time
       $aDate = array( 'day' => $day,'month' => $month,'year' => $year);
       $eventList->whereAdd('date_expire >= \''. SGL_Date::
                     arrayToString($aDate).'\'');

       $result = $eventList->find();
       $aEvents = array();
       if ($result > 0) {
          while ($eventList->fetch()) {
             //check if start date and expiry date are the same.
             //if yes show only one date in template
              if ($eventList->date_start == $eventList->date_expire) $eventList-
                     >date_start = '';
              $aEvents[] = clone($eventList);
          }
       }
      return $aEvents;
   }

{...}
```

theme directory *www/themes/myTheme/default/*, and do the same with the *css* and *images* directories. The other templates don't have to be copied to your new theme if you don't want to change them. Seagull automatically takes the ones from the default theme if it doesn't find them in yours. Normally you can change a lot just by using different CSS in the stylesheet, without touching the XHTML in the templates. Easy, isn't it? So new themes are always up to date if something changes in the module's templates.

You may think why to make a new theme if you only want to change one or two files. But it's better to seperate your changes from Seagull's default code, so you can update the framework more easily without touching the changes you made.

Let's change the following two files in your theme's default folder: banner.html and *footer.html*. There is also a file called *header.html* however this contains everything from the first <html> tag until the end of the opening <body> tag and does not need to be modified.

You can also modify the main CSS file: seagull/www/themes/myTheme/css/style.php

Activate the new theme by adding it to the global preferences: change the *theme* preference's value to the name of your new theme directory, e.g. *myTheme*, and voilà – you can see the new layout you have created by just modifying two or three files.

Conclusion

It surely doesn't make sense to set up a very small project on a framework. For a small site with 5-10 pages that never change it's better to use static HTML. But if you want to create medium or big applications containing user and rights management and/or a CMS system, it's very easy setting them up using the functionality the Seagull PHP Framework offers.

About author

The author joined the Seagull Developer Community in spring 2004 searching for a good framework he could use for his projects. He now maintains the project's documentation wiki. In his free time he loves to play the guitar in the old-music band "Schwartenhals".
Contact the author: werner.krauss@hallstatt.net