Guide to Building Websites with SWGfL Framework

South West Grid for Learning Trust Ltd, Belvedere House, Woodwater Park, Pynes Hill, Exeter, EX2 5WS
Registered in England and Wales, Company Number 5589479. Charity Number 1120354. VAT Reg. Number 880 8618 88

Contents

Technologies and Installation

The following outlines the required and desired technologies with links to acquire the packages.

Git

Git is a distributed version control system that is used to store and version the website code, and document the changes that are made to it and who made them.

To use Git on Windows, the following software should be installed:

- Git SCM, the core Git engine - http://git-scm.com/download/win
- TortoiseGit, a context GUI for Git - https://code.google.com/p/tortoisegit/wiki/Download?tm=2

Licence: GNU General Public License version 2.0

Node.js

A system for executing Javascript on the server. Asynchronous and non-blocking in nature, node.js makes very efficient use of resources by splitting tasks up into jobs that are processed by worker threads, on completion, an event is raised which will triggers a callback to continue processing the next part of the code.

Needed to run Grunt tasks.

To install node.js on windows, visit https://nodejs.org/

Licence: MIT

Grunt

Grunt is a task runner written in Javascript/node.js that enables tasks to be automated, such as building the output files for the website.

Once node.js is installed, it is worth installing Grunt globally on your machine, open a command window as an administrator and run the following command:

```
npm install -g grunt-cli
```

If you are installing an existing project that has a **package.json** file, see "Starting with the Starter Project"

Licence: MIT

PHP

Server-side scripting language for delivering dynamic websites. Download from http://windows.php.net/.

If using FastCGI as a handler on IIS, use a Non-Thread Safe (NTS) version, otherwise use the Thread Safe (TS) version. You also have to make sure that the same C++ compiler was used on the version you download as all your extensions, and Apache if that is your webserver, e.g. VC15.

Licence: PHP Licence v3.01

Apache Webserver

This is the software that handled all your requests for websites and servers up the PHP output and static assets.

Official Apache 2 is built using VC6, so you will have to download your Apache build from http://www.apachelounge.com/download/ to get VC11+ builds.

Licence: Apache Licence, Version 2.0

MySQL Community Server

Open source relational database management system. Download from http://dev.mysql.com/downloads/mysql/, be sure to note down the username and password you setup which can be used to connect your app to the database, in a production environment you would setup different users for each site, but local it is fine to use the same.

Licence: GNU General Public License version 2.0

phpMyAdmin

A PHP application that provides a web interface for managing a MySQL database. Download from http://www.phpmyadmin.net/home_page/downloads.php and extract to a folder in your webroot, then run http://127.0.0.1/[name-of-folder]/setup/ and follow the wizard.

Licence: GNU General Public License version 2.0

SASS

Syntactically Awesome Style Sheets, a superset of CSS designed to make CSS more modular. Enables style to be built in modules, use functions and variables. SASS files are written with the newer SCSS syntax, and uses grunt-sass for compilation.

See https://github.com/sindresorhus/grunt-sass for how to install the grunt task.

Licence: MIT

RequireJS/AMD

A framework for modularising Javascript, modules are built as AMD modules, and a grunt task resolves the dependencies and outputs a build file.

To install the Grunt task see: https://github.com/gruntjs/grunt-contrib-requirejs. When building your Javascript files, you should also use AMDclean to avoid having to build and AMD loader in, see https://www.npmjs.com/package/grunt-amdclean.

Licence: Dual Licenced, New BSD or MIT

SWGfL Framework

The SWGfL framework is a modular utility framework with a number of components for performing various tasks required to run a website:

Audit

A module for logging system events into a database table.

CMS

An MVC application that renders an interface for searching for and editing a single MySQL table of data. A table configuration defines the layout of each table you wish to edit and their editable fields. CMS's can also be embedded within one another.

Config

Provides functionality for merging, storing, and retrieving configuration arrays, this enables components to have default configurations which then get merged with a customised configuration before retrieval and insertion into the relevant application.

Cookies

A system for managing cookies used on the website, and for setting cookie permissions.

Database

An interface to a MySQL database, providing simple database operations and data retrieval, as well as helper functions for generating SQL statements.

Dataset

A barebones MVC framework that provides helper methods for building MVC applications and handling dataset operations such as searching, paging and sorting.

Development

Provides a static log to record application events and can render javascript onto a page to show the log in the console. To help you debug your application.

Email

An interface to compile and send HTML and multipart MIME emails.

Environment

Sets up an environment for running web applications through a server, and provides an autoload interface. Generates variables required to enable uniform internal and front-end addressing in your web application.

Event

Enables better decoupling of applications by providing an API for binding and triggering events.

Forms

An interface for checking form data and rendering controls, each control has its own class which can check and render its control type. Also provides other helper functions for registering variables and generating query stings.

Image

A library for manipulating images providing functions such as resizing, cropping, rotating, greyscale etc.

Inflection

Static utility API for converting singular and plural English words.

Javascript

Provides a number of modular Javascript's to enabling various functions.

PDF

Methods for converting HTML pages into PDF documents and images.

Scheduler

A system for configuring and scheduling tasks to run in the background.

Security

Provides functionality to enable users to login and logout, secure pages, and perform other security related functions such as password resetting, account activation, autologin, and CSRF protection (Cross Site Request Forgery).

Template

Provides templating and HTML tools, a renderer for error messages, and HTML header output control.

Video

Provides functions for streaming video files over a HTTP connections, supporting range requests.

Project Layout

This is the recommended layout you should use for your web application:

- application – Stores all the code for the project such as PHP files, SASS stylesheets and AMD modules
    - cms - Stores the content management system configuration for any custom content management systems
    - swgfl – This is where the SWGfL Framework should be included as a git sub-module
    - website – Create an MVC application to run the website
    - bootstrap.php – Includes/autoloads all the required libraries for the website, invokes the environment, and includes any custom configuration
    - bootstrap.scss – Includes all the SASS files for building the websites main stylesheet
    - bootstrap.js – Invokes the SOA for all the javascript code that should be built into the main javascript file
- config – Stores the custom configuration for the website
    - _config.scss – Setup any SASS variables here
    - config.inc.php – Custom configuration for the website PHP application, stores for example the database configuration
- database – Store any database schema dumps in here, this folder should be versioned
- docs – If generating documentation with Grunt, output the documentation into here
- httpdocs – All web readable files should be stored in here, and should be setup as the root of your website so only files in this folder are publically accessible. Note that if on the target server this folder is called something else, this folder can be renamed, but all references to files in that folder such as in your grunt file will need to be updated
    - css – Stores any static CSS files and fonts
        - build – Stores dynamically generated CSS built from SASS
    - graphics – Store website graphics used to implement the design of the site here
    - images – Store static images that are uploaded to the site through the CMS or are part of the content of the website
    - javascript – Stores any static Javascript files
        - build – Stores dynamically generated Javascript build with RequireJS

    - .htaccess – If using the apache webserver, store server directives in here, this file should redirect all PHP traffic to index.php, and setup headers for gzipping textual content and set cache control headers
    - web.config – Same as above but for IIS webserver.
    - index.php – Invokes application/bootstrap.php and sends it its folder address so the environment can extrapolate the internal addressing
- workspace – Keep any resources and original files in here
- .gitignore - Stores a list of files and folders that should be ignored by Git
- .gitmodules - References any sub-modules that are imported into the project
- .htaccess – A directive file for Apache Webserver to forward local requests to the httpdocs/ folder, this will enable you to access the website with http://127.0.0.1/website/ instead of http://127.0.0.1/website/httpdocs/
- config.env - Stores environment specific configuration
- gruntfile.js – Stores your grunt tasks and configuration
- package.json – Node Package Manager (NPM) dependency definition file

Starting a New Project with the Starter Project

The best way to start a new SWGfL Framework project is to clone the starter project and customise the code from there. If you wish to start from scratch, see "Using the Framework without the Starter Project".

Clone the starter project

The starter project can be found in S:\WebTeam\Git_Repositories\starter.git. Clone this project into a folder under your local web root.

If using TortoiseGit, you can clone the project by creating a new folder, right click and select "Git Clone". In the dialogue that appears, enter the URL of the Starter Project, and the directory to clone the project into, and click "OK".

Install the required Node Modules

The node modules that are setup for the Starter Project are defined in package.json, and need to be installed with the node package manager.

Open a command window in the project root folder you just cloned the project into. You can do this by holding down Shift and right clicking in the folder, this will enable the "Open a command window here" option. Click to open.

Enter the following command:

```
npm install
```

Install the database

Create a new database using phpMyAdmin, and import database/starter.sql. If you are not using global permissions to access your databases, setup the appropriate permissions.

Edit the configuration file

Copy config.sample.env and rename to config.env. Then in your IDE change the database access credentials.

Build the website assets

Using the command window we opened before, build the output files for the site using the following command:

```
grunt
```

Run the website

Open your web browser and enter the web address of the site, if you created a folder underneath your web root and have a local server, enter http://127.0.0.1/[website]/ (Where [website] is the name of the folder you created).

Create a new Git Repository

You should keep an upstream copy of your Git repo in S:\WebTeam\Projects\New System\[Your Project Name]\, create a new folder named [Your project name].git. Then create a bare repo here.

In TortoiseGit do this by right clicking the folder and clicking "Git create repository here...", bare repo will already be selected if the folder name ends in .git.

Make sure your local changes have been committed, then push the repo. In TortoiseGit this will bring up the "remote" dialogue, here enter the URL of the repository you just created and save the remote, which will now be called origin.

Develop your Project

You can now start developing your project.

Troubleshooting

If you get the following error:

```
> Command not found
```

When installing your node packages or running grunt, make sure that you have node.js in your PATH variable.

If there are any problems with the database, a description of the problem will be displayed on screen.

If there are other issues, make sure that development mode is on in the configuration file. Currently it is setup with a Regular Expression which analyses the hostname that the website is being run on to determine whether the website is being run locally. Development mode is then set to on if the website is detected as being run locally.

The pattern will capture either 127.0.0.1 or a NetBIOS name (with no dots in it) as local, so if your hostname is for example website.local, then the site will default to production mode.

Using the Framework without the Starter Project

If you are creating a new project from scratch or bringing the framework into an existing project, do the following (assumes you have already setup a Git repository for your project):

Add the project as a Sub-Module

In TortoiseGit, right click your project root folder, and select **TortoiseGit > Sub-Module Add...** then enter S:\WebTeam\Git Repositories\framework.git, enter **application/swgfl** in the Path field.

How the Starter Project Works

The starter project is an example of a basic barebones website with a login system and a content management system for the navigation, users, and groups, to provide sample code to develop projects with and promote good practice.

Bootloading

The website starts loading through **httpdocs/index.php**, this file takes the Webroot address and passes it to the bootstrap() function in **bootstrap.php**. This file loads the framework's environment and autoloads any other apps. Each application folder should have an **autoload.php** file to specify where its files are.

Website Module

The starter project has a module to run the website in **application/website**. Inside is a set of MVC classes that generates the website interface, the view retrieves the details of the current page and the list of pages in the navigation from the database through the model class.

Navigation System

The navigation system is setup to enable the database to deliver the required information for each page, such as title, description, page content, application, search engine settings, whether the page requires a login, and whether to show the page in the navigation.

The security system is tied in to kick you to the login page when you are not logged in, and to change the navigation to show the pages you have permission to access when you are logged in.

How Pages are Constructed

The view controller that controls constructing website pages is **websiteView::drawPage()**, it retrieves the page details from the database and compiles it into the main template, which is stored in **application/website/templates/template.php**.

The method **websiteView::drawBlocks()** has a switch statement to bootstrap each application defined in the applications table. Each page can have multiple blocks created inside it, each app renders its HTML in the order defined, which is then returned.

The SASS files for the website are in **application/website/stylesheets/**.

Edit these files in your new project to get the layout how you want it, pull out the information you need from the database, and add any other template functions / applications.

Content Management System Configuration

The content management system is configured by defining the table structure and editable fields for each table you wish to edit in a single file. In the starter project, the definition file can be found at **application/cms/cms.config.php**. This file defines a class for generating the configuration to prevent any scope where it is being used being polluted by its variables.

Three tables are setup in the starter project to define the nav, content, and images table for editing. More details on the available configuration options can be found in **cmsModel::__construct()** and **formField::$config**;

The account management configuration is defined in **application/swgfl/security/cms/cms.config.php**. This file provides a base to work from, and should not be edited. If changes need to be made to the account management system, a new file should be created to modify the output from this file before passing to the callee in **websiteView::drawBlocks()**.

Module Development

This section describes how you should develop your custom modules in your SWGfL Framework project:

Module Layout

Each module you develop for you website should be stored in the following format:

- application
    - [Your module] – make a folder with the name of your module
        - javascript – store any AMD modules for your module here
        - stylesheets – store any SASS files in here
            - _all.scss – link your SASS modules together with this file for inclusion in the main CSS output file
        - templates – store any php templates in here
        - autoload.php - Loads all the class references into the framework's autoloader
        - [module].config.php – Write the default configuration and setup procedure for your module in here (See configuration)
        - [module].class.php – If your application is a utility class or does not produce an interface, you will probably only have one main class file, so it should be named like this
        - [module].controller.php – If your application is MVC, your controller class should be named like this
        - [module].model.php – If your application is MVC, your model class should be named like this
        - [module].view.php – If your application is MVC, your view class should be named like this

Any other files required as part of your module should reside in this folder, or sub-folders as required.

Loading Modules

Whilst technically modules could be loaded in many different ways, this section describes the recommended way to load modules, an example of which can be seen in the starter project.

In the starter project each page can embed a multiple applications per page; the selected application is linked to the content table from the applications table. Each application has an appCode field which is an internal textual name for the application.

The method websiteView::drawBlock() contains the bootloaders for each application that has been defined in a switch statement against their codenames, each bootloader compiles the required configuration and passes it to any modules that are loaded, and wires their outputs to the correct array keys in the $content array for loading into the template.

Code Philosophy

The framework has been built with performance and modularity in mind, in order to build high quality and performant websites with the framework, the following guidelines should be used:

PHP Classes, Functions, Methods, Properties and Variable Names

All PHP classes, functions, methods, properties, and variable names should be named using camelCaseWords. Array indices should be lowercase-and-dashed.

```
$myArrayVar[‘property-name’] = …
```

Code Spacing and Comments

Blocks of code functionality should be separated by an extra line space, and should also be preceded by a comment on its own line:

```
public function doSomething() {

    // this bit of code does something
    $something = ‘done’;

    // this bit does something else
    $somethingElse = ‘done’;
}
```

Decoupling

Applications should be careful not to be coupled to more components than is necessary, there are number of ways of decoupling your applications that should be employed:

- Make sure your module has a well-defined role, and do not allow it to work outside of that role

- Only let your application be aware of components when it is its job to handle those components, if it is not, the data should be prepared for the application in the glue code that creates the object and the data should be passed to it
- Injecting data instead of reading it from a global location or loading the source application at the point where the data is needed
- Having the appropriate data output or API to enable some glue code to be written to couple components together, instead of a component being used inside the other module
- Injecting dependencies so they can be optional or replaceable

Most applications you build will require some sort of dynamic data to be available in your application that is sourced from another part of your system. In order to decouple components as much as possible, you application should only be aware of other objects when it is in control of them, or where the functionality is a utility and it is accepted that the components are coupled.

For this reason, constants and global variables should not be used anywhere in your application. Instead you should enable entry of a configuration into your application either statically (Globally available to every instance of that object), or at the point of object creation. Some sort of "glue" code should wire up the outputs of one module and pass it to the other module.

Decoupling with Events

The events class is provided to help decouple your application where other parts of your system need to be notified when something happens. You don't want your application to be aware of the other components in the system, only that is should tell something that something happened. This component allows you to bind callbacks to events that happen, and then trigger those events from another part of your application.

Your application should also not be coupled to the events class, so its dependency should be injected through the application configuration:

```
$config['event'] = '\swgfl\event\event::trigger';
$obj = new module($config);
```

And then in your application you can trigger an event if there is a trigger handler available:

```
if ($this->config['event']) {
    call_user_func($this->config['event'], ‘event.name’);
}
```

Events should be namespaced also, as “module.event”.

The binding of events should be done in glue code where possible, to call the module that needs to receive the data, decoupling the receiving component from the events module. E.g:

```
\swgfl\event\event::bind(Array(
    'module.event' => function () {
        otherModule::doSomething();
    }
));
```

A great example of this is the database class, which triggers an event when the database is connected and when queries are executed.

The config file for the development module (application/swgfl/development/development.config.php) binds a callback to this event which sends the data generated by each query to the development module, which is then available in the browser console so SQL queries can be easily debugged.

Neither the development or database modules are aware of the event module or each other, but the configuration scripts glue them all together so they work without any setup (The event module is injected into the database’s configuration from the database configuration file, using the unified config provided by the config module). If the default config is not going to set the module up as you want, simply write your own instead.

Configuration

To make applications as easy as possible to use without having to write lots of glue code to get it up and running, you should develop your apps to be as autonomous as possible without compromising configurability. This can be done through either proving a separate script with glue code that sets up a default configuration, or by building the default configuration directly into the application.

The config class provides functionality for generating a single configuration for every component of your website. Each configuration section should be split up into namespaces for each module of your application, which you will see is already implemented for the framework.

All applications that have any configuration that is to be delivered to the application to get it working will have a [module].config.php file in its module folder. If your setup procedure requires variables or is more complex than simply including some files, then inside should be a self-executing anonymous callback function to setup the object, this prevents the global namespace from being polluted with any variables that are used in your setup procedure.

```
call_user_func(function () { … });
```

A basic setup procedure could involve creating a default configuration array and then merging that configuration into the main configuration for that namespace, which will allow it to be overwritten by any custom configuration:

```
$config = Array(
     ‘default’ => ‘config’,
     …
);
\swgfl\config\config::setConfig($config, ‘module’);
```

You application may require the configuration to be set globally before the object is initialised, in which case your application should have a static configuration and a static method to set it, then it should be set in the configuration file:

```
$config = \swgfl\config\config::getConfig(‘module’);
module::setConfig($config);
```

Otherwise when you create an instance of your object the configuration should be passed to it:

```
$config = \swgfl\config\config::getConfig(‘module’);
$obj = new module($config);
```

For performance, and depending on the type of application, it is usually not worth checking that every configuration variable is in a correct state before applying a configuration into

your application. Sensible defaults can go a long way to mitigating this problem, but your app may crash when values are missing or are incorrect.

Environmental Variables

The environment class generates a number of environmental variables which should be used in your modules to facilitate server and front-end addressing. The environmental variables can be retrieved with the following code:

```
$env = \swgfl\environment\environment::getEnv();
```

This will return an array with the following keys:

- libfolder – The address of the root folder on the server, e.g. D:/Websites/project/
- serverfolder – The address of the web accessible folder on the server, e.g. D:/Websites/project/httpdocs/
- scriptname – The path and filename of the requested web page. e.g. **/some/folder/module/script.php**
- folder – The path from the domain root to the webroot, so if your website root is accessible through http://mydomain.com/some/folder/, then this variable will contain **/some/folder/**
- https – Boolean denoting whether the request was made over a secure connection
- Webroot - The protocol and hostname or the website, e.g. http://mydomain.com, without the trailing slash
- webhost – The protocol and hostname concatenated to the folder value, e.g. http://mydomain.com/some/folder/

Addressing and URLs

Your application and modules should have the environmental variables described above passed to them in a configuration array, all internal and web facing addresses should then be built using them to address the root folders of your application. As a rule of thumb, URL parts should always have forward slashes at the end to make concatenation easier.

HTML Output

Where your application needs to output HTML, then a template should be used, this is effectively a view (whereas your module.view.php file is a view controller):

```
\swgfl\template\template::compileHtml($content, __DIR__.'/templates/my-template.php');
```

CSS Class Names

Block-Element-Modifier (BEM) is an approach to front-end development designed with flexibility and ease of modification in mind, technically it is a naming convention that follows the format:

```
block__element--modifier
```

It is a much more descriptive naming convention for you CSS class names, and removes any hierarchical issues in your CSS.

BEM methodology should be implemented in your SWGfL Framework website.

Read more here: http://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/

MVC Applications

There are many different ways of setting up an MVC application, this is the approach that should be taken in this framework:

- Controller – Should provide the public API for the module, handle all user data and pass it to the methods in the model, it should make top level decisions and raise errors
- Model – All business logic should go in here, checking form data, updating the database, retrieving data from the database, sending email etc.
- View – Responsible for rendering HTML, and all the control logic associated with that (It is effectively a view controller, and the templates are the views), including retrieving the required data. Data should not be retrieved in the controller and passed to it, if the rendering engine is required to be separate from the data source, a control method should be built into the view to retrieve the data and pass it to the renderer

The dataset module is available for setting up MVC applications, extend your controller, model, and view with the three dataset classes, and in your controller constructor, run:

```
parent::mvc('moduleModel', 'moduleView'[, $argsForModel, …]);
```

The database class is extended by the datasetModel class, so the database will also be natively available in your model.

Documentation

Your code should be self-documented using PHP DocBlocks. You can use grunt-shell to generate the API documentation using for example PhpDocumentor or ApiGen. See https://github.com/sindresorhus/grunt-shell for how to install and use grunt shell.

The Phar archives for your documenter should be downloaded to the **docs/** folder in your project and documentation generated into a subfolder of that (Shell command):

```
php docs/apigen.phar generate -s application/swgfl -d docs/framework
```

The starter project contains a grunt task to generate documentation, you will need to copy the appropriate PHAR archive into the **docs/** folder.

Useful Grunt Tasks

Watch

Watch is a task that listens for changes in specified files and invokes an action when one of those files is changed, this is useful for development for example when you update a SASS file, the watch task can build your CSS file for you so when you load your website in your browser on your development machine, you can see the output of the changes you just made.

See https://github.com/gruntjs/grunt-contrib-watch.

PostCSS

A module for binding CSS transform modules together, it can do things such as autoprefix properties for different browsers, inline small assets, optimise inline SVGs, and minify the output.

See https://github.com/nDmitry/grunt-postcss