Coding Style Guide

Being a Kohana application, the Ushahidi platform naturally takes on Kohana's coding standards - which we will go ahead and replicate in this section.

NOTE: Ushahidi is based on Kohana 2.3.1 but the Kohana framework has since evolved and therefore, some of the standards outlined here may not apply to later iterations of the framework.

It is encouraged that you follow the coding style outlined in this section in order to make the code more readable, consistent and allow for easier sharing and contributing.

Class Names and File Location

There are 4 types of classes namely:

- controllers
- models
- libraries
- helpers

Controllers

Being an MVC framework, controllers stand in between models and views in the application. They pass information on to the model when data needs to be changed and they request information from the model. The rules for naming a controller are:

- It must reside in the **`application/controllers`** directory
- The filename must be in lowercase e.g. `reports.php`
- The controller class must map to the filename with the first letter capitalized and must be appended with **`_Controller`**} e.g. **`Reports_Controller`**
- The controller class must be subclass (directly/indirectly) of the **`Controller`** class

Additional documentation on controllers can be found here: http://docs.kohanaphp.com/general/controllers

Models

Models provide a convenient way of interacting with the database. They are a part of Kohana's ORM architecture and therefore abstract the gory details of platform-specific database calls. Kohana however doesn't force you to use models.
The rules for naming a model are:

- Models reside in the **`application/models`** directory
- Model filenames are lowercase and should be in the singular form of the name e.g. `incident.php` NOT `incidents.php`
- Model class names are capitalized and must be appended with **`_Model`** e.g. `Incident_Model`

Additional documentation on models can be found here: http://docs.kohanaphp.com/general/models

Libraries

Rules for creating libraries are:

- Library files reside in the **`application/libraries`** directory
- Library class names must begin with a capitalized letter e.g. `Geocoder`
- Library file names must begin with a capitalized letter and must be named the same as the library class e.g. **`Geocoder.php`**
- A library may have **`_Core`** appended to the class name so that it can be extended (sub-classed) in the same way as Kohana's built-in libraries

Additional documentation on libraries can be found here: http://docs.kohanaphp.com/general/libraries

Helpers

Helpers are classes that are used to isolate "handy" or "useful" functionality.

Rules for creating helper classes are:

- Helper files reside in the **`application/helpers`** directory
- Helper file names must be in lower case and named the same as the helper class e.g. **`category.php`**
- Helper class names must be in lower case and may be appended with **`_Core`** e.g. **`category_Core`**

Additional documentation on helpers can be found here: http://docs.kohanaphp.com/general/helpers

Coding Standards

For the source code to be consistent, we encourage all developers to follow the coding style guidelines.

Brackets

Use BSD/Allman Style bracketing. This style puts braces associated with a control statement or function definition on their own line. The exception to this rule is the opening brace for a class definition, which can be on the same line (as the class name).

```
if ($x > 1)
{
    print 'Greater than 1';
}
else
{
    print 'Less than 1';
}

// The opening brace for a class definition can be on the same line
class SampleClass {
```

Naming Conventions

Kohana uses under_score naming, not camelCase naming.

Classes

```
// Controller class uses _Controller suffix
class Boiler_Controller extends Controller {

// Model class uses _Model suffix
class Plate_Model extends Model {

// Regular class
class Chicken {
```

Functions and Methods

Function names should be in lowercase and use under_scores to separate words:

```
function show_contacts($foo)
{
    $foo->bar();
}
```

Variables

Variable names should lowercase and use under_score, not camelCase:

```
// Correct
$foo = 'bar';
$long_example = 'uses underscores';

// Incorrect
$hasWindowCurtains = 'No window curtains';
```

Indentation

Use tabs to indent your code. The use of spaces is forbidden.

Vertical spacing (for multi-line) is done with spaces. Tabs are not good for vertical alignment because different people have different tab widths.

```
$text = 'This is a long text to demonstrate the guideline for indenting your '
      .'code. The practice is to use TABS and not spaces. Also vertical alignment '
      .'for multi-line is done with spaces because not everyone uses the same tab '
      .'width.';
```

Single-line Statements

Single-line `IF` should only be used when breaking normal execution (e.g. `return` or `continue`)

```
// Acceptable
if ($foo == $bar)
    return $foo;

if ($foo == $bar)
    continue;

if ($foo == $bar)
    break;

// Not acceptable
if ($foo == $bar)
    $foo += 1;
```

Comparison Operations

Use `AND` and `OR` for comparison.

```
// Correct
if (($foo AND $bar) OR ($a AND $b))

// Incorrect
if (($foo && $bar) || ($a && $b))
```

`elseif`, NOT `else if`

```
// Correct
elseif ($a > $b)

// Incorrect
else if ($a > $b)
```

Parentheses

- There should be one space after the statement name
- The "!" (bang) character must have a space on either side to ensure maximum readability. Except in the case of a bang in type casting, there should be no whitespace after an opening parentheses or before a closing parentheses.

```
// Correct
if ($foo == $bar)
if ( ! $foo)
if ( (int) $foo)

// Incorrect
if($foo == $bar)
if (!$foo)
if ( $foo == $bar )
if ((int) $foo)
if (! $foo)
```

Ternaries

Ternary operations should follow a standard format. Use parentheses around expressions only, not around variables.

```
// Correct
$foo = ($bar == 5) ? $foo : $bar;
$foo = $bar ? $foo : $bar;

// Incorrect
$foo = ($bar) ? $foo : $bar;
```

When separating ternaries into lines, spaces should be used to line up the operators, which should at the front of successive lines:

```
$foo = ($foo == $bar)
    ? $foo
    : $bar;
```

Basic Documentation Commenting

We use phpDocumentor, a tool for creating documentation directly from both PHP and external documentation, to keep track of all our code documentation. The documentation within the code is done using PHPDoc which is an adaptation of Javadoc for the PHP programming language.

The objective of documenting code is to makes it easier to understand and lower the barrier to entry for developers wishing to contribute to the codebase. It also facilitates painless maintenance (refactoring, bug fixes etc).

The first thing to take note of is that PHPDoc comments must be enclosed in DocBlocks. A DocBlock is a C-stlye comment that begins with a `/**` an d with a leading asterisk `*` on each line. Any line within a DocBlock that doesn't begin with a `*` will be ignored. Example:

```
/**
 * Example use of DocBlocks in PHP
 */
```

Secondly, DocBlocks must precede the code you are adding comments to. For example, if you wanted to document the function `foo()`, you would proceed as follows:

```
/**
 * DocBlock comment for function "foo()"
 */
function foo()
{
}
```