# Sensible PHP and Sensible Living

A new, simple PHP framework and its applications

November 2009

# Sensible PHP and Sensible Living

We are:

Aaron Palmer
James Jhurani
Longyi Qi

# 1. Sensible PHP and Sensible Living

This report is a small summary of our new php mvc framework, Sensible PHP and its first application, Sensible Living.

## 1.1 Introduction

Sensible PHP is a small, basic and simple framework, and it’s new and great! Sensible PHP aims to make common web-development tasks fast and flexible.
Sensible Living is the first application of Sensible PHP, based on Sensible PHP, we made a blog and a dictionary to implement the features of Sensible PHP, and to test how well Sensible PHP works and deploys in the real projects.

## 1.2 Features

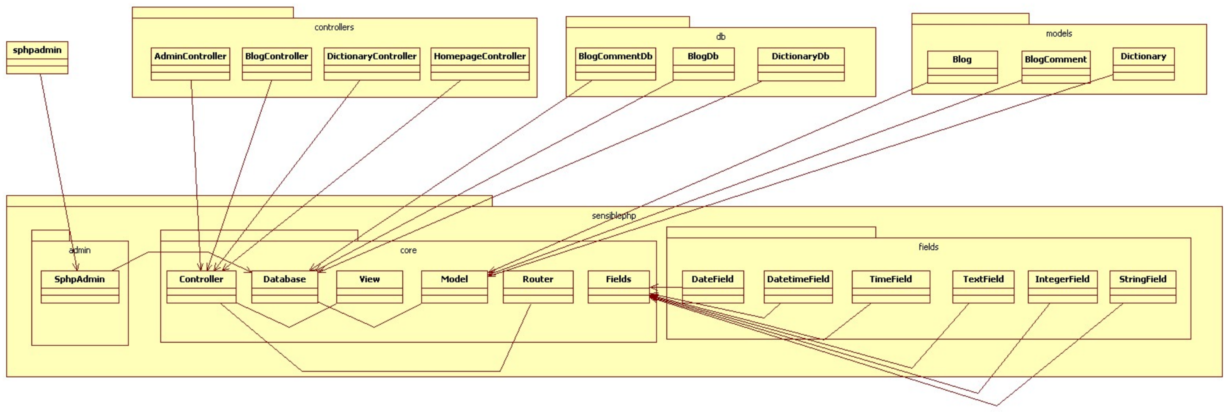
### 1.2.1 Features done

- By using our management script, it's pretty easy to initialize the project, create applications and sync with database.
- We focus on adhering to the DRY principle, so it's easy to modify the models, and write controllers, additional database logics and views.
- Flexible in coding, reuse and deployment.
- Easy to handle static media files, like Javascript, cascade style sheet, images, audios and videos.
- Easy for modification and extension as we adhere OCP principle.
- No worry on URL address, simply use the application name and the action name, user can get the result.
- Trash the old applications when running the clean or init project command.

### 1.2.2 Feature in mind

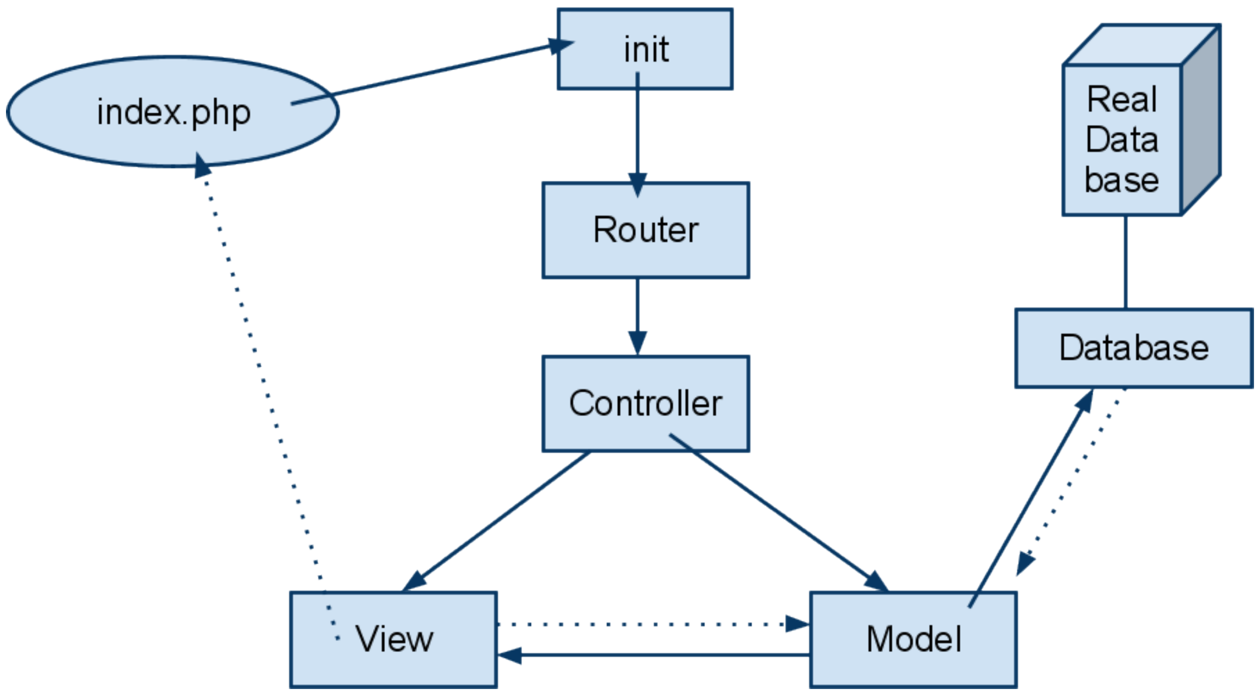
- Form Generation and validations
- Automatic admin interface
- Template system
- Cache system
- Error notification email to the admin

### 1.3 Class Diagram



The full image is stored in {svnroot}/docs/uml_diagram/class_diagram.jpg

### 1.4 Action Flow Diagram



## 2. Technology and Tools

- Programming Language: PHP 5
- Libraries: PhpUnit(for tests), XDebug(for code coverage), Pdepend(for metrics generation)
- Database: MySQL
- Editor: TextMate(Mac), vim(Linux) and notepad(Windows)
- Web technology: HTML 4
- Server: Apache httpd

- Agile development practice: Redmine, Google Group mailing list, Google docs and Google wave

## 3. Design Principles

### 3.1 Don't Repeat Yourself Principle

We really want to say, our code is really DRY, and in three aspects -
First, our framework is DRY, as the framework developer, we don't want Copy and Paste a single line of code, it's a technical debt, we make our code cohesive and usable, so we never repeat ourselves.
Second, our applications are DRY, as a user, code from initializing the project, creating applications, writing models, writing additional and special database logics(if need, the most frequently used database logics have been implemented by framework, yes, we never want to repeat ourselves), writing controllers, until writing views, in each aspect, we try to make our code more usable and flexible.
Third, our application deployment also is DRY, for example, a user may write a blog system with sensible php framework, and one day, another guy want to have a blog in the same server, too. So, what the second user needs is just to initialize his own project, configure the settings to define his own database and include the original blog application into the php class path in his own project, and sync the database, done. We learned this concept from Django, a python mvc framework. But for the third aspect, there are some additional work left, well, at the moment, when the second user including the first user's blog, he will include other applications from first user at the same time, the reason for this is the file structure of current version, we will modify the file structure to let each application has its own folder, which contains this application's models, controls, database logics and template pages, it's more flexible and easily to implement DRY when deployment.

### 3.2 Open-Closed Principle

Our code is open for extension and closed for modification.
Like all models({svnroot}/trunk/models), controllers({svnroot}/trunk/controllers), database logics({svnroot}/trunk/db) and views({svnroot}/trunk/views) of applications are OCP achieved, so, each time user want to add a new application, just write the new related code for this application, and no need to modify the existing code.
And in framework, all field types({svnroot}/trunk/sensiblephp/fields) are also OCP achieved, simply add new Field type classes in php path, and use it.

### 3.3 Single Responsibility Principle

And we make our code SRP, in all method, class, package and component level.
Strong cohesive, that only do one thing and do one thing well. As higher cohesion results in good reuse, then the reusability of our code is very high.

### 3.4 Acyclic Dependence Principle

See our class diagram, the dependency structure is a directed acyclic graphic without cycles, so it's easy for us to maintain our code and do small increment deployments during the development.

### 3.5 Stable Dependency Principle and Stable Abstraction Principle

We achieve this as in our project, all components with high I metrics depend on the ones with low I metrics, and what's more, the result of our A, I and D' is great!

## 4. Design Patterns

- The whole project we wrote is a MVC pattern, we isolates business logic from input and presentation, by decoupling models and views, MVC helps to reduce the complexity in architectural design and to increase flexibility and reuse of code.
- What's more, MVC is a Observers pattern, which define one-to-many dependency between objects so that each time one object changes state, all it's dependents are notified and updated automatically. For example, admin user makes some modifications, all readers will see the change immediately (for web application, readers may need to reload the page or use either Ajax technology or HTML 5 server-push technology.)
- In detail, we also use some other Design Patterns, like Object Adapter. For example, DatatimeField is an implementation of Fields abstract class, what's more, it has DateField and TimeField as Adaptees.

## 5. Test Cases

We did our utmost to test our codes, for both sensible php framework and sensible living applications, we use phpunit 3.3 as our test framework, and use XDebug to generate the html report. Our test report are divided into two parts, one for sensible php framework and the other for sensible living applications, the detail code coverage reports are stored in {svnroot}/docs/test_code_coverage_report folder.

### 5.1 Tests for sensible php framework

Here is the summary of our framework code coverage report. Our tests coverage all classes, and 92.16% methods, and 88.22% lines, some tests of exception are left for future work.

| | Coverage | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Lines | | | Functions / Methods | | | Classes | | |
| Total | 🟩 | 88.22% | 277 / 314 | 🟩 | 92.16% | 94 / 102 | 🟩 | 100.00% | 12 / 12 |
| conf | 🟩 | 100.00% | 1 / 1 | 🟩 | 100.00% | 1 / 1 | 🟩 | 100.00% | 1 / 1 |
| sensiblephp | 🟩 | 88.18% | 276 / 313 | 🟩 | 92.08% | 93 / 101 | 🟩 | 100.00% | 11 / 11 |

### 5.2 Tests for sensible living applications

And the below is the summary of our application code coverage report. As we have already done the tests for sensible php framework above, so we can ignore the conf and sensiblephp part in this report, so the accurate test coverage for our applications is 100%.

| | Coverage | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Lines | | | Functions / Methods | | | Classes | | |
| Total | 🟨 | 69.82% | 155 / 222 | 🟨 | 56.86% | 29 / 51 | 🟩 | 87.50% | 7 / 8 |
| conf | | 0.00% | 0 / 1 | | 0.00% | 0 / 1 | | 0.00% | 0 / 1 |
| controllers | 🟩 | 100.00% | 128 / 128 | 🟩 | 100.00% | 20 / 20 | 🟩 | 100.00% | 4 / 4 |
| sensiblephp | 🟥 | 29.03% | 27 / 93 | 🟥 | 30.00% | 9 / 30 | 🟩 | 100.00% | 3 / 3 |

And in tests for applications, we use a mock framework natively integrated in phpunit, it convenience our way to write tests. See the following example, as we want to write a test case for deleteBlogCommentByBlogId method as below,

```
function deleteBlogCommentByBlogId($id) {
        return $this->delete("WHERE `blog_id`=" . $id);
}
```

We can use the mock framework to write in the following way,

```
public function testDeleteBlogCommentByBlogId() {
        $mockBlogCommentDb = $this->getMock('BlogCommentDb', array('delete'));
        $mockBlogCommentDb->expects($this->once())->method('delete')->with($this->equalTo('WHERE `blog_id`=1'));
        $mockBlogCommentDb->deleteBlogCommentByBlogId(1);
}
```

As we have done the tests for delete method in the framework tests, so here in applications tests, what we need is to make sure these new methods can call the methods in framework correctly, and that's fine. So in the test case above, first we create a mock object of *BlogCommentDb*, and we define our expectation as the delete method with parameter *WHERE `blog_id`=1* will be called only once (at least once and at most once), and then call the *deleteBlogCommentByBlogId* method with a parameter *1*, then run the tests, if the method runs as our expectation, then the tests pass, if not, an exception will be throwed by the test framework.

## 6. Proud

### 6.1 MVC framework itself

We are really proud that what we did IS NOT only how to use MVC framework, instead, we made our own framework ourselves. What's more, although this framework is simple, but I'd like to say, it's a good one! The core features we implemented are really good.

### 6.2 DRY

Here, we want to emphasis DRP again, because in our project, everything are really DRY. For code, our code is cohesive that we don't allow the same code to display twice. For practice, when user uses this framework, both in writing application and deployment, he or she will feel the happiness from DRY.

### 6.3 OCP

This is another aspect we proud. All the fields implementation are OCP achievement. Need a new FieldType? Just add it, and it works immediately, that's all. And all the applications' models, database logics, controllers are OCP achieved, too. Users can easily add new applications without modifying the existing code.

### 6.4 Test code coverage and other matrices

I am so proud about the test coverage we got by using tools. The detail report generated by these tools are stored in {svnroot}/docs folder.

## 7. Not proud

Some features of framework have not be implemented, like Form Generation and validations, automatic admin interface, template system and cache system are not done yet.
What's more, for sensible living applications, only blog and dictionary are done, product catalog and couple are left.

## 8. Screenshots and Demo

We offer a real-animation screenshots, go to http://cents.fazey.org and run our demo as much and as long as you like.

## Reference:

1. Bscphp, http://code.google.com/p/bscphp/, a basic and simple php framework that developed by Jiajun Gao (chief developer), Longyi Qi, Chen Chen and Qi Qin
2. Django, http://www.djangoproject.com/, a high-level Python Web framework that encourages rapid development and clean, pragmatic design.
3. Struts, http://struts.apache.org/, an elegant, extensible framework for creating enterprise-ready Java web applications.
4. Ruby on rails, http://rubyonrails.org/, an open source web application framework for the Ruby programming language. It is intended to be used with an Agile development methodology that is used by web developers for rapid development.
5. Slides of Course Software Design by Dr. Venkat Subramaniam
6. Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides