www.phparch.com

November 2013
VOLUME 12 - ISSUE 11

php[architect]

Functional Testing in Symfony 2

Interoperability: The Future of PHP

Laravel - A Modern PHP Framework

Adianti Framework

Frameworks

ALSO INSIDE

Adianti Framework

Pablo Dall'Oglio

Adianti Framework provides a complete architecture for developing PHP applications. It is a component-based and event-driven framework specializing in the development of business applications. In this paper, I'll introduce you to the basic concepts of the framework.

DisplayInfo()

Related URLs:

- Adianti Framework – http://www.adianti.com/framework
- Adianti Studio – http://www.adianti.com/studio
- Framework Quick Start - http://adianti.com/framework-quickstart

Introduction

As you know, a framework is an abstraction which provides generic functionality that can be extended by developers when building specific applications. Most people use frameworks to be more productive in software development and to spend time meeting software requirements, not to deal with low-level aspects like persistence and presentation.

As a developer, I was lucky enough to start using PHP back in 2000. At that time, the team I worked with had created an entire academic ERP system (called SAGU) in PHP 4. A year later (2001), Rasmus Lerdorf came to Brazil for a Free Software conference. Rasmus told us that SAGU was the biggest PHP system in the southern hemisphere at that time. The team was young and did not know much about good architecture practices. The source code was entirely procedural. The system fulfilled the user needs, but the team was not proud of that code, and maintenance was painful.

As time passed, I learned a lot about architecture and design patterns. PHP improved a lot with PHP 5 in 2004, and yet by 2006, frameworks for PHP development were just gaining traction, so I started to write a new one focused on the development of business applications, like the old academic ERP system. The development of this kind of system required certain features: a fixed set of components to build standard interfaces, because the system must be homogeneous; a good persistence layer to let developers worry about business rules, not about SQL statements; easy to learn technology, because you will have a heterogeneous group of high skilled developers and also less skilled ones, and everyone must be able to write clean code.

These requirements guided me in the development of the Adianti Framework, and the main points of the framework are the development of business applications and the ease to learn. You don't have to learn a couple of technologies like JavaScript, jQuery, and others to start developing, just PHP. That's because the framework has a couple of PHP components that encapsulate different technologies, like jQuery and Twitter bootstrap, so it reduces the need to integrate different technologies to start developing. For those that would like even more productivity, there's also Adianti Studio Pro, an IDE that provides wizards for datagrids and automatic form generation, beyond an interface designer.

Persistence

The framework persistence layer is built around the Active Record and Repository Design Patterns. To start everything, we must declare a subclass of `TRecord`, that is an Active Record implementation that provides methods like `store()`, `delete()`, `load()`, and others. In the following sample, there's the `Customer` class, that is an Active Record. All Active Records must use a single primary key. Adianti Studio Pro can automatically generate methods to deal with relationships like compositions, associations, and aggregations inside the Active Records. Suppose `Customer` has a composition with the `Contacts` class. This way, when loading or storing a `Customer` Active Record, also its `Contact` information (composed objects) will be handled in this operation.

```
class Customer extends TRecord
{
    const TABLENAME = 'customer';
    const PRIMARYKEY= 'id';
    const IDPOLICY =  'max';
    // {max, serial}
}
```

The implementation of database operations leverages the PDO library. It uses transactions and exception handling by default. We can stack transactions and also log all of the automatic operations done by the framework persistence layer. In the following sample, we are using the `Customer` class, which is a subclass of `TRecord`, to load a customer (31) and change one of its attributes. The framework maps the attributes automatically to the database columns. We can restrict which attributes we want to map.

```
TTransaction::open('samples'); // open transaction
$customer = new Customer(31);
$customer->phone = '51 8111-3333'; // changes the phone
$customer->store(); // stores the object
new TMessage('info', 'Object updated');
TTransaction::close(); // closes transaction
```

To deal with collections, we can use the Repository pattern (`TRepository` class). This class deals with collections of objects. The `TRepository` class uses a criteria object (`TCriteria`) that is an implementation of the Composite Pattern. This way, we can create composite filters. In Listing 1, we are loading all customers of female gender and showing some data.

LISTING 1

```
1. TTransaction::open('samples'); // opens a transaction
2. $criteria = new TCriteria;
3. $criteria->add(new TFilter('gender', '=', 'F'));
4. $repository = new TRepository('Customer');
5. $customers = $repository->load($criteria);
6. foreach ($customers as $customer)
7. {
8.    echo $customer->id . ' - ' . $customer->name . '<br>';
9. }
10. TTransaction::close(); // closes the transaction
```

Page Controllers

All the application pages made with the Adianti Framework must be subclasses of `TPage` or `TWindow`. The only difference is that `TWindow` opens over the application in a separate window. The page content is added to the page itself using object composition. The framework offers a group of containers and widgets to build the application interface. In this sample, we are creating a simple page with a label inside. In the web environment, the application flow is implemented using the Front Controller Pattern. To render a class, you must enter: `index.php?class=SimpleView`, for instance.

```
class SimpleView extends TPage
{
   public function __construct()
   {
      parent::__construct();
      parent::add(new TLabel('Hello World'));
   }
}
```

This sample demonstrates how to run specific methods. In this case, we must specify the class name, the method to be executed, and any additional parameters in the URL: `index.php?class=SimpleView&method=onHello&name=Pablo`. This way, the method `onHello()` from the `SimpleView` class will be executed. The `onHello()` method will receive a parameter (`$param` in this case), that will contain the request (`$_REQUEST`) for this call.

```
class SimpleView extends TPage
{
    public function onHello($param)
    {
        parent::add(new TLabel('Hello ' . $param['name']));
    }
}
```

The framework also offers an implementation of the Template View pattern. In this case, we can use HTML fragments to compose the interface. The `THtmlRenderer` class takes in HTML and allows us to add this HTML inside a page. The `THtmlRenderer` class also allows the developer to perform some operations like: enabling or disabling HTML sections; replacing HTML variables with static content or with framework widgets; and repeating HTML sections. In Listing 2, we are just enabling a section and replacing some variables with static content.

Containers

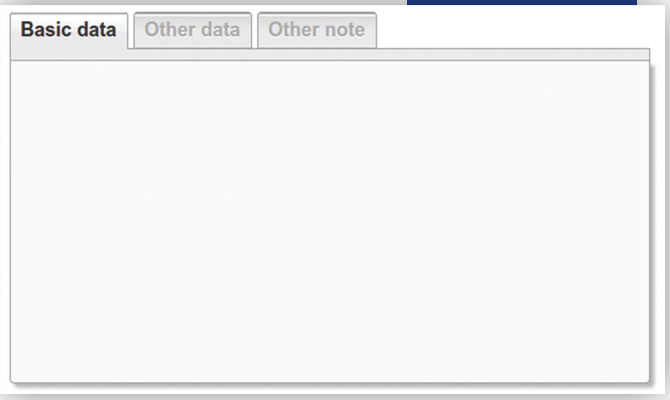
The framework offers some containers to build the application pages. In this sample, we can realize objects like `TNotebook`, `TTable`, and `TPanel` that are containers used to build the interface. The interesting part of this approach is that you can run your application both as a default web application and also as a desktop application. The Adianti Framework provides two implementations: one that uses web technologies like jQuery and another one that uses the GTK library for the desktop. We can put containers inside containers (as shown in Listing 3) and also use the Template View to build the interface under the web.

In Figure 1, we can see this container.

LISTING 2

```
1. class TemplateView extends TPage
2. {
3.    public function __construct()
4.    {
5.    parent::__construct();
6.    $replace = ['code' =>1, 'name'=> 'Mary'];
7.    $html = new THtmlRenderer('app/resources/customer.html');
8.    $html->enableSection('main', $replace);
9.    parent::add($html);
10.   }
11. }
```

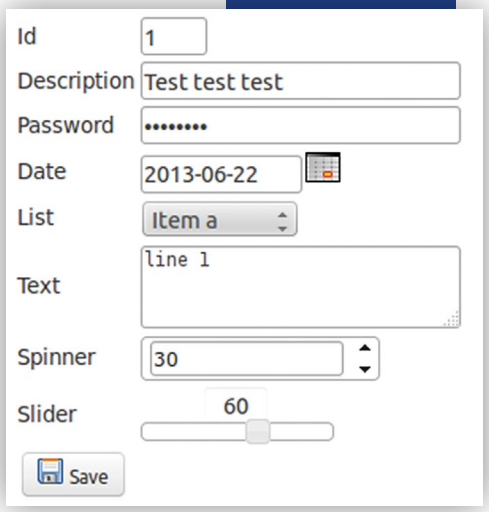
FIGURE 1



LISTING 3

```
1. class ContainerNotebookView extends TPage
2. {
3.    function __construct()
4.    {
5.       parent::__construct();
6.
7.       // creates the notebook
8.       $notebook = new TNotebook(400,200);
9.
10.      // creates the containers for each notebook page
11.      $page1 = new TTable;
12.      $page2 = new TPanel(370,180);
13.      $page3 = new TTable;
14.      // adds two pages in the notebook
15.      $notebook->appendPage('Basic data', $page1);
16.      $notebook->appendPage('Other data', $page2);
17.      $notebook->appendPage('Other note', $page3);
18.      parent::add($notebook);
19.   }
20. }
```

FIGURE 2



LISTING 4

```
1. class TestView extends TPage
2. {
3.    private $form;
4.    function __construct()
5.    {
6.       parent::__construct();
7.       $this->form = new TQuickForm;
8.
9.       $id          = new TEntry('id');
10.      $description = new TEntry('description');
11.      $password    = new TPassword('password');
12.      $date        = new TDate('date');
13.      $list        = new TCombo('list');
14.      $text        = new TText('text');
15.      $spinner     = new TSpinner('spinner');
16.      $slider      = new TSlider('slider');
17.      $spinner->setRange(0,100,10);
18.      $slider->setRange(0,100,10);
19.      $list->addItems(array('a'=>'Item a', 'b'=>'Item b'));
20.
21.      $this->form->addQuickField('Id',    $id,    40);
22.      $this->form->addQuickField('Description', $description, 200);
23.      $this->form->addQuickField('Password', $password, 200);
24.      $this->form->addQuickField('Date', $date, 100);
25.      $this->form->addQuickField('List', $list, 100);
26.      $this->form->addQuickField('Text', $text, 120);
27.      $this->form->addQuickField('Spinier', $spinner, 120);
28.      $this->form->addQuickField('Slider', $slider, 120);
29.      $text->setSize(200,50);
30.
31.      $this->form->addQuickAction('Save',
32.      new TAction(array($this, 'onSave')), 'ico_save.png');
33.      parent::add($this->form);
34.   }
35. }
```

Rich Widgets

The framework offers lots of widgets to build forms. To arrange these widgets on the screen, we can use a table container to display the elements in rows and columns, but we can also use a panel container, displaying the elements in absolute coordinates. As you can see, there are many ways to build an application form, but the simplest way is to use the `TQuickForm` class, placing one component above the other. In Listing 4, we are creating a form and adding some input components inside of it. There is also an action button (Save) that is connected to the `onSave()` callback. We could also wrap this form inside a notebook or another container. The framework provides many kinds of components to use inside a form like `TEntry` (text input), `TDate` (date picker), `TPassword` (password entry), `TSpinner`, and `TSlider`, among others.

In Figure 2, we can see the form in action.

The form action is represented by an object of `TAction` class. This class encapsulates a PHP callback. In the following code, we can see the `onSave()` method that is executed when the user clicks on the save button of the form. In this case, the form data is collected by the `getData()` method which returns an object with the form data. In this case, we are just showing some information using the default message dialog (`TMessage`).

```
public function onSave($param)
{
   $data = $this->form->getData();
   $message = 'Id: '          . $data->id . '<br>';
   $message.= 'Description: ' . $data->description . '<br>';
   $message.= 'Date: '        . $data->date . '<br>';
   $message.= 'List: '        . $data->list . '<br>';
   $message.= 'Text: '        . $data->text . '<br>';
   new TMessage('info', $message);
}
```

Interface Designer

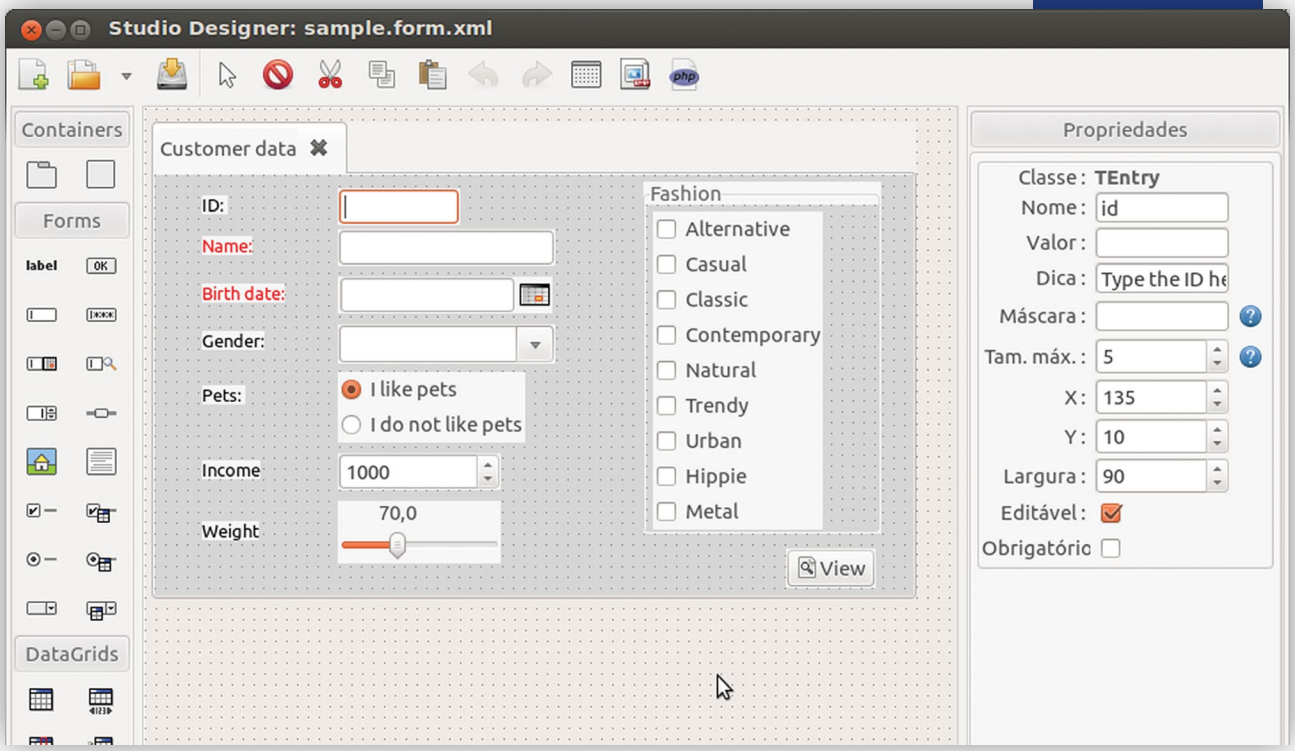
Some developers love to define interfaces by hand; others do not. For those who prefer to draw interfaces using drag and drop, there's the Adianti Studio Designer, part of Adianti Studio Pro. With Adianti Studio Designer, you can draw interfaces which are stored in XML files. We can take these XML files and use a framework class called `TUIBuilder` to render the application interface. This class will render the interface both for the web environment and also for a desktop environment (using the GTK library). The `TUIBuilder` class parses the XML file, renders the interface, and makes the designed objects available as regular framework objects through methods like `getWidget()`.

Taking a look at the interface, which is saved in an XML file, the code in Listing 5 takes this file and renders it. All the actions defined in the Designer are automatically mapped to class methods.

LISTING 5

```
1. class DesignFormView extends TPage
2. {
3.    private $form;
4.
5.    function __construct()
6.    {
7.       parent::__construct();
8.
9.       $this->form = new TForm;
10.
11.      try
12.      {
13.         $ui = new TUIBuilder(500,300);
14.         $ui->setController($this);
15.         $ui->setForm($this->form);
16.
17.         // reads the xml form
18.         $ui->parseFile('app/forms/sample.form.xml');
19.
20.         $this->form->add($ui);
21.         $this->form->setFields($ui->getFields());
22.      }
23.      catch (Exception $e)
24.      {
25.         new TMessage('error', $e->getMessage());
26.      }
27.
28.      parent::add($this->form);
29.   }
30. }
```

FIGURE 3



In Figure 3, you can see the Adianti Studio Designer in action. You can build an interface using native framework containers (`TFrame`, `TNotebook`), form components (`TEntry`, `TPassword`, `TButton`, `TDate`, `TSlider`, `TSpinner`), and also datagrids.

Final Considerations

Adianti Framework is an open source project that is growing quickly in Brazil. People are recognizing the framework as an easy-to-learn technology, with ready-to-use components, allowing heterogeneous teams to build quality business applications. The framework does not intend to be a general purpose solution. It is probably not the best option to build a blog, a public web page, or things like that, but it is a great option for building business applications that run internally inside an organization.

PABLO DALL'OGLIO has created many projects for PHP development (Tulip IDE, Agata Report, and Adianti Framework). He is active within the Brazil PHP community and is the author of books on PHP-GTK, PHP Reporting, and Object Orientation in PHP. He also teaches software engineering to university students.

@pablodalloglio