pocket-knife

The Code

All code is hosted at https://github.com/andrewsbrown/pocket-knife.

Introduction

`pocket-knife` is a mini-framework for making sites and services. It is fast and simple. It is RESTful. It is minimal. It is a collection of the tools I use every day to make web development easy.

The idea is that RESTful PHP applications do not have to be complex to be powerful; that simplicity also makes them fast. Web developers build the same components (services, settings, sites, and sequences) again and again: the `pocket-knife` components are lightweight so they can be easily extended and fully tested (note: working on this...) so you can trust their reliability.

The framework is built along two lines: 1) a collection of common functions, and 2) four common components. The common collection includes functions like `WebHttp::request()`, which performs any method of HTTP request on a URL—this is a quite common task and can be tricky without cURL (and this function does not use cURL). The four components use these common functions to simplify the development of complex applications. They are:

> 1) the `Service` component exposes a RESTful web service with content-type detection, customizable ACL and authentication, templating, and a bunch of supported database drivers.
>
> 2) the `Settings` component edits and stores Settingss--again, with all of the buzzwords.
>
> 3) the `Sequence` component is a multi-step web service (think shopping cart or sign-up form).
>
> 4) the `Site` component picks up any stray HTML/PHP pages and organizes them with site-mapping and search functions.

Together, the components make up an entire framework covering most web application architectures. A blog? Basically a `Service`. The admin section of a mail client? A `Settings` component. A sign-up form for new users? A simple `Sequence`. And the conglomerate of HTML and text files on my home server? A `Site`.

Requirements

`pocket-knife` uses the following:

- PHP5

- json (PECL)
- PDO, if you choose to connect to MySQL
- Mongo, if you choose to connect to MongoDB

That is it. Let's start.

Getting Started

We can make a calculator quite easily:

1. Get the code:

```
git clone git://github.com/andrewsbrown/pocket-knife.git
```

2. Create the calculator class, `calculator.php`:

```
<?php
class Calculator extends ResourceStatic{
     public $a;
     public $b;
     public function sum(){
          if( !is_numeric($this->a) || !is_numeric($this->b) )
throw new ExceptionService('Inputs must be numeric!', 400);
          return $this->a + $this->b;
     }
}
?>
```

3. Create the resource anchor, `service.php`, and include `pocket-knife` (see below).
4. Include that `Calculator` class we just made (see below).
5. Create a `Settings` component that allows all users access to all classes (don't worry, the ACL can get more complex) using GET variables as input and JSON as output (see below).
6. Create the service and let it run free:

```
<?php
include('pocket-knife/start.php');
include('calculator.php');
$settings = new Settings( array(
     'acl' => true,
     'input' => 'application/x-www-form-urlencoded',
     'output' => 'application/json'
));
```

```
$service = new Service($settings);
$service->execute();
?>
```

7. Access it from the browser with: `http://[your_server_here]/[path_to_files]/service.php/calculator/sum?a=100&b=200`

Too simplistic, right? We could make the same thing happen with one line of PHP (let's see... `<?php echo json_encode($_GET['a'] + $_GET['b']); ?>`). Please, read on.

More Complexity

In the simple example above, we saw several key pocket-knife concepts: using plain-old PHP classes to do the work (business logic), creating a Settings object with all of our configuration values, and letting Service handle the connecting parts by calling execute(). This model scales to more complex applications.

Before we look at another example, one aside: there are many PHP frameworks that can form the basis of a web service. Many of these are solid frameworks (Zend, CakePHP, Yii) that use the MVC architecture to build very powerful systems. For certain projects, however, they are too much: I usually do not need views or models or to learn an exhausing new programming paradigm. Never fear: pocket-knife will stick with its four core components and will do so, uncompressed, in less than 1MB. And it will be fast (TODO: need benchmarks).

Now, let's build a blog (to follow along better, take a look at the nosql-blog code found at https://github.com/andrewsbrown/nosql-blog):

1. Get the code (unless you did this the first time; one pocket-knife installation can serve as a library for all projects):

```
git clone git://github.com/andrewsbrown/pocket-knife.git
```

2. Create a post class, Post.php:

```
<?php
class Post extends ResourceItem{


    public function create($item = null){
        if( !is_null($item) ) $item->created = date('Y-m-d
H:i:s');
        $item = WebHttp::clean($item, 'html');
        $id = parent::create($item);
        $_GET['message'] = 'Item created successfully';
        $url = WebRouting::createUrl('posts');
        WebHttp::redirect($url);
        return $id;
    }
```

```
    public function update($item = null){
        if( !is_null($item) ) $item->modified = date('c');
        $item = WebHttp::clean($item, 'html');
        $item = parent::update($item);
        $_GET['message'] = 'Item updated successfully';
        $url = WebRouting::createUrl('post/'.$this->getID());
        WebHttp::redirect($url);
        return $item;
    }

    public function delete(){
        $item = parent::delete();
        $_GET['message'] = 'Item deleted successfully';
        $url = WebRouting::createUrl('posts');
        WebHttp::redirect($url);
        return $item;
    }
}
```

Before we move on, let's look at some of the features of the code. First, notice that the class is extending ResourceItem. This is a RESTful representation of an item in a list, just as ResourceList represents a list and ResourceStatic represents an object unrelated to anything else. ResourceItems inherit common methods like create, read, update, and delete. Browse the code or API for more on this.

This brings up another point: where is `ResourceItem`? All `pocket-knife` classes use the upper-case letters to locate themselves within the file structure. For example, ResourceItem is located in [pocket-knife directory]/Resource/Item.php.

Let's look at create(). It takes an item (FYI: pocket-knife uses stdClass objects to pass these around), adds the created property, cleans it using a convenient WebHttp method, saves the data using parent::create(), and then redirects us to another page. The remaining functions are variations on this.

Notice that we don't overwrite read(). We do this because we have no need to redirect, add properties, etc., as we did with the other functions. In fact, for some services, you may have no need to overwrite any functions at all. For example, if we ported this application to a JSON service we could use all of the default methods.

3. Create a posts class, Posts.php:

```
<?php
class Posts extends ServiceObjectList{ }
```

```
?>
```

Notice that here we have no need to overwrite any methods, since we will only be using the read() method to return all results.

4. Create a template and UI. For space, I do not post all code here, but it is available at https://github.com/andrewsbrown/nosql-blog. Also, I could have chosen a different approach, such as using ServiceMap to map requests to different templates or skipped the UI altogether; instead I made a simple PHP template importing each PHP snippet as needed, as in the following excerpt:

```
<?php
    if( !@$error ){
        list($class, $id, $method) = Service::getRouting();
        if( $class == 'posts' && $method == 'read' ) include
'snippet-all.php';
        if( $class == 'post' && $method == 'read' ) include
'snippet-view.php';
        if( $class == 'post' && $method == 'edit' ) include
'snippet-edit.php';
    }
?>
```

An excerpt from 'snippet-view.php':

```
    <fieldset>
        <legend>Post</legend>
        <span class="field">
            <label for="post-id">ID</label>
            <span id="post-id"><?php echo @$data->id; ?></span>
        </span>
        <span class="field">
            <label for="post-created">Created On</label>
            <span id="post-created"><?php echo @$data->created; ?
></span>
        </span>
          ...
```

$data holds the information returned from our storage system (see below).

5. Create the service and let it run free (only an excerpt shown, see 'index.php' for more details):

```
// setup application
$configuration = new Configuration(array(
    'acl' => array('* can access * in post', '* can access * in
```

```
posts'),
     'template' => 'ui/template.php',
     'storage' => $storage_configuration
));

$app = new Service($configuration);
$app->execute();
```

where $storage_configuration is one of:

```
// choose DB to use
$dbs = array(
    'sql' => array(
          'type'=>'pdo',
          'username'=>'dev',
          'password'=>'dev',
          'location'=>'localhost',
          'database'=>'blog',
          'table' => 'posts',
          'primary' => 'id'
     ),
    'couch' => array(
          'type'=>'couch',
          'location'=>'localhost',
          'database'=>'blog'
     ),
    'mongo' => array(
          'type'=>'mongo',
          'location'=>'localhost',
          'database'=>'blog',
          'collection' => 'posts'
     ),
    'json' => array(
          'type'=>'json',
          'location'=>'data/db.json',
          'schema'=>''
     )
```

```
);
```

This example uses a very loose ACL, the template defined at 'ui/template.php', and a curious storage method: a choice between four different databases. In common practice you may only use one, but this application was built to test the capabilities of the framework.

6. Access it from the browser with: `http://[...]/service.php/posts`

   One more thing: the resource actions, like Post::create() or Posts::read(), are mapped to the HTTP methods POST, GET, PUT, and DELETE (create to POST, read to READ, update to PUT, delete to DELETE). Actions on the ResourceItem are accomplished by using these HTTP methods on a RESTful URL: in the example above (and assuming an anchor of service.php) we are requesting a Posts::read() by doing a GET on http://[...]/service.php/posts. Alternately, we could skip the HTTP methods entirely and perform a common GET or POST using the function name in the URL, like http://[...]/service.php/posts/read. For a specific post, we might use http://[...]/service.php/post/23/delete.

   Again, please see the github code (https://github.com/andrewsbrown/nosql-blog) for more detail.

API

[TODO: import from PHP documentation]