Andrew Brown
27 October 2013

pocket-knife: a RESTful, PHP web service framework

Introduction

In developing web-based applications, a common problem developers face is how to save and subsequently present data for users. The problem is compounded by the growing classes of users: desktop users want HTML web pages, mobile users need minimally-styled HTML, and other developers want raw data in the form of XML or JSON. Though the problem is common, large-scale solutions like Content Management Systems (CMS) rarely fit—the answer selected in this project is to build a web service framework upon which to later build custom applications.

Web service frameworks are not new: web services based on SOAP and WSDL technologies are standard in Java and Microsoft environments. However, the WS-* pattern (?) for service-oriented architectures is just too complicated, according to Pautasso. Hence the development of RESTful (REpresentational State Transfer) web services based on Roy Fielding's 2001 dissertation. [Pautasso] notes that these REST architectures are “of the web” not built “on the web”.

The growth of RESTful architectures and the spread of the REST paradigm has been significant. Every major web company (e.g. Facebook, Google, Twitter, etc.) has provided REST Application Programming Interface (API) end points spawning thousands of articles and tutorials online. But REST has also influenced academia; in fact, during the initial stages of the writing of this report (January 2011 to March 2012), the *ACM Guide to Computing Literature*, which records bibliographic citations from major publishers in computing, records 174 articles and papers spanning which are related to RESTful architectures.

This project develops a RESTful, PHP web service framework, *pocket-knife*, and compares it against other web service frameworks available online. Additionally, it presents working examples of *pocket-knife* to demonstrate key features built into the framework.

Goals

The project’s goals were:

- To faithfully implement REST principles
- To make the Resource pattern simple
- To provide only enough features to make the framework self-sufficient yet focused

In code, the framework would have to allow something like:

```
<?php
/* include core pocket-knife files
Require ‘./start.php’;

/* create RESTful resources and link them to the HTTP verbs
class FileStore extends Resource{
	public function GET(){
		/* code to GET resource in a format-agnostic way; in other words,
		/* retrieve a Resource and let the web service determine the content-type
	}
```

```
        public function POST($resource){
                /* ...
        }
}

/* start the web service
$settings = new Settings('config.json'); /* load configuration files from a variety of formats
$service = new Service($settings);
$service→execute(); /* start the web service quickly
?>
```

Why PHP?

PHP (PHP: Hypertext Preprocessor) is the most prevalent web-based programming language, largely due to its simplicity and ease of setup. It is tightly incorporated into the most common web server technology, Apache. According to a January 2013 Netcraft study, 244 million Internet sites use PHP to as a server-side programming language [netcraft].

Though it has its detractors, mainly for its legacy inconsistencies and sprawling library structure, PHP is a solid language for a web service framework: 1) it has been proven successful in large-scale environments, 2) it is object-oriented, and 3) it is a technology native to the web. All of these form the basis for using PHP to build *pocket-knife*.

Why a Framework?

In building solutions, developers must choose an approach that balances development time and design freedom. The chart below simplifies this continuum into three categories: general-purpose libraries, frameworks, and domain-specific solutions. On the general side, more development time is invested in order to produce a custom solution to the problem; this approach allows the developer the most freedom in regards to architecture, optimization, extensibility, and features. On the specific side, design freedom is sacrificed in order to produce a faster solution. In the middle, frameworks attempt to balance both ends of the continuum.

General-Purpose Libraries	**Frameworks**	**Domain-Specific Solutions**
PHP modules	CakePHP	Wordpress
PEAR	Ruby on Rails (non-PHP)	OpenCart
Java Servlet (non-PHP)	pocket-knife	OwnCloud

Table 1: Examples of Application Solutions

An incorrect decision at this step of the development process could lead to problems. For example, imagine a custom course registration system built to coordinate the schedule of several large rock-climbing gymnasiums on the East Coast (I worked on such a system in 2008). It was built from scratch in Perl, using only built-in functions and modules. The system worked well initially and conformed exactly to the customer's needs; in this sense, the solution had been built with complete design freedom. Unfortunately, bugs in the code surfaced several years after launch; the original developer was switching jobs and could not support the customer. When our company was hired to fix the bugs, the work billed to the customer was significant: debugging required an understanding of the entire system.

A common approach (and one frequently used for the scenario above) would involve including or building generic modules to create abstractions for common actions, objects, etc. When this is combined with a known architecture, the result is a framework. Though not optimal in all situations, frameworks like *pocket-knife* can simplify and speed up development while allowing enough freedom to produce custom solutions.

Why REST?

In his 2000 doctoral dissertation, Roy Fielding published the REpresentational State Transfer (REST) architectural style. The concepts were not newly invented: since 1995 he had been involved in creating the Apache HTTP server, the web server that as of this project's publishing commands 45% of the web server share across the internet [netcraft]. He had also been involved in creating the HTTP 1.0 (1996) and HTTP 1.1 (1999) standards. Therefore, his dissertation—and REST—was intended as a guide for how HTTP was meant to be used.

In chapter 5, Fielding described REST in terms of constraints and elements:

Constraint	**Description**
Client-Server	Servers provide data; clients consume data
Stateless	Session state must be kept on the client
Cache	Responses may be cached and used for subsequent client requests
Uniform Interface	Clients access data through URNs or URLs; also, implies Hypermedia as the Engine of Application State (HATEOAS), meaning there is no API beyond a set of URLs and the given HTTP verbs
Layered System	Clients only see one upstream layer/node though data may flow through multiple nodes
Code-On-Demand	Optionally, clients access application features by downloading applets or scripts

Table 2: REST Constraints [Fielding]

Element	**Description**
Resource	Any piece of data that can be named; an abstraction
Resource Identifier	The name (URL or URN) given the data; should be static
Representation	The data; a concrete sequence of bytes

Table 3: Major REST Data Elements

The constraints and elements, though described abstractly in the dissertation, map specifically to HTTP. In this context they can be summarized by very simple principles, such as those proposed by Alex Rodriguez:

1. Use HTTP methods explicitly: the HTTP verbs GET, PUT, POST, and DELETE (note: there are several other rare verbs) should be the only method calls the application makes. When applicable, pass data in the HTTP body, not in the URL.

HTTP Verb	CRUD method	Data Transfer (Common Usage)
GET	Read	Only in response
PUT	Update	Request and response
POST	Create	Request and response
DELETE	Delete	Only in response

Table 4: Description of HTTP Verbs

2. Be stateless: do not maintain session state on the server. This not only avoids the performance hit of additional processing on the server but conforms more closely to the original intent of HTTP. Each HTTP request should contain all of the data (in the HTTP body) and the metadata (in the HTTP headers) for the server to form a proper response.
3. Expose directory structure-like URLs: the URIs exposed by the application should be human-readable and rule-based. This creates consistency and simplicity for the client-side developer. Additionally, avoid long queries and hide the file extensions (e.g. ".php") so URIs remain constant even if the system is ported to another language.
4. Transfer XML, JSON, or both: On the server end, use the HTTP Accept header of the client request to determine what MIME type to send back. The resource identifier should remain static; the representation should change to fit the client's preferred content. [Rodriguez]

Similar Frameworks

Frameworks with the RESTful architectural style have been implemented in PHP. Lane gives a list of some of the frameworks in this class, disregarding those that are generic enough to be made RESTful (e.g. CakePHP; does not enforce REST constraints) and those that merely expose "RESTful URLs" (i.e. URLs like http://www.example.com/job/1/create) but do not conform to all REST constraints:

1. DAVE: a "minimalist, multi-node, transactional API framework written in PHP". Exposes the actions Delete, Add, Edit, and View. (https://github.com/evantahler/PHP-DAVE-API)
2. Epiphany: a micro framework that routes RESTful URLs to PHP functions. Includes database and session helpers, among other things. (https://github.com/jmathai/epiphany)
3. FRAPI: consists of an HTML administrative interface to manage actions and a public API within which to code the business logic. (http://frapi.github.com/)
4. Recess: MVC-based (Model-View-Controller) framework with HTML administrative tools for managing applications. (http://www.recessframework.org/)
5. Slim: a micro framework that routes RESTful URLs to anonymous functions. (http://www.slimframework.com/)
6. Tonic: uses resources and content negotiation; most like pocket-knife in theory. (http://peej.github.com/tonic/)
7. Zend Framework: includes the class Zend_Rest_Server for linking classes or callback functions to RESTful URLs. (http://framework.zend.com/manual/en/zend.rest.server.html) [Lane]

In examining each of these, the frameworks were analyzed according to five categories: 1) what type of architecture they framed (for this category, the generic Model-View-Controller [MVC] pattern was used to classify them), 2) whether they were fully REST-compliant, 3) whether they provided authentication and authorization features, 4) whether they automatically cached responses, and 5) whether they performed automatic content-type detection using HTTP headers like *Accept*.

Framework	**Type**	**REST**	**Auth**	**Caching**	**MIME Detect**
pocket-knife	MVC	Yes	Yes	Yes	Yes
Zend	MVC	Yes	Yes	Yes	Yes
Yii	MVC	Yes?	Yes	Yes	No
Recess	MVC	Yes	No	No	No
Epiphany	C	Yes	No	Yes	No
Tonic	C	Yes	Yes	No	Yes
Slim	V + C	Yes	No	Yes	No
DAVE	MVC	Yes?	No	Yes	Yes
FRAPI	V + C	Yes	No	Yes	Yes

Table 5: PHP Web Service Framework Comparison

The frameworks examined fall into several rough categories. The Zend Framework and Yii are *large, widely-used* PHP frameworks and are not aimed solely at creating API endpoints; as a result, creating a RESTful web services with these may not be intuitive and may require a significant amount of research in the documentation (i.e. Zend Framework). Recess is similar to these in that it is a full MVC system but different in that it is built specifically for web services; however, it lacks the features and does not seem to have been changed since 2010. Epiphany, Tonic, and Slim fall in the *micro framework* category; their focus is on code simplicity—they do one thing well. Finally, DAVE and FRAPI fall into the *experimental* category, since DAVE offers de-coupled MVCs and Delete, Add, View, Edit (DAVE) actions as a replacement for HTTP verbs and FRAPI offers a graphical UI for creating controller routes.

Distinguishing Features

pocket-knife implements features that have proven useful in developing two commercial systems and one academic project (as a UI frontend to test various NoSQL databases). Hosted on GitHub (http://github.com/andrewsbrown/pocket-knife), the framework has 461 commits to date representing 8,800 lines of code with an additional 6,200 lines of documentation in over 90 PHP classes. All core features are covered by PHPUnit tests. Additionally, the framework is distinguished from the frameworks above by the following features:

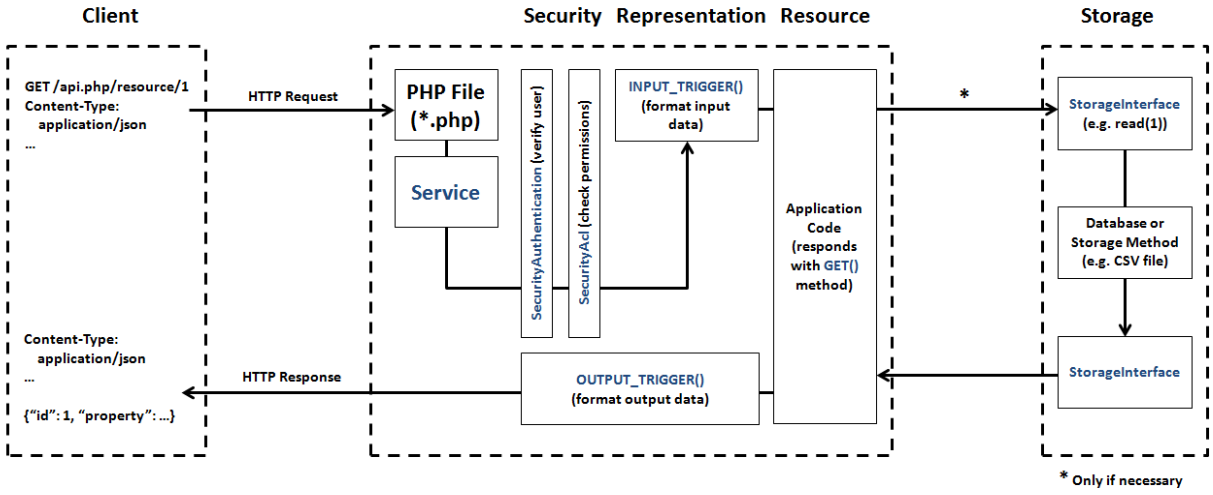
1. *RESTful URLs*: the URIs, by necessity of being built on PHP, are parsed after the *.php* web service endpoint. We call this file the anchor and the slash-delimited strings after it the tokens; e.g. in *rest.php/book/5*, *rest.php* is the anchor and *book* and *5* are both tokens. This approach is consistent with popular usage; Marinos et al describe a similar pattern for accessing database records using RESTful URLs. Their approach, however, contains parentheses; e.g. http://example.org/{database}/{table}({id}) [Marinos].

2. *Security: pocket-knife* implements a security mechanism based on two core classes: *SecurityAuthentication* and *SecurityAcl*. During an HTTP request to a service, *SecurityAuthentication* will verify the user's identity (current methods include HTTP Basic Authentication, HTTP Digest Authentication, passing credentials as HTTP headers, session authentication using PHP, and Facebook OAuth 2.0). The identity is checked against a list of users (users also have roles, similar to groups), itself a *ResourceList* that can be stored with any of *pocket-knife*'s storage methods. Once verified, *SecurityAcl* will check whether the user has permission to access the desired resource. These checks are performed against a list of fine-grained rules (another *ResourceList*) in the form {"role": …, "name": …, "resource": …, "id": …, "method": …}; *SecurityAcl* can parse these rules from reasonably-formatted English sentences as seen below. Both the user list and the ACL list are fully configurable in configuration files. An example JSON configuration file creating an *admin* user and giving this user permissions:

```
{
    "authentication":{
        "enforce_https": true,              /* optionally enforce HTTPS connections */
        "authentication_type": "digest",    /* will use HTTP Digest Authentication here */
        "password_security": "plaintext",   /* optionally encrypt passwords */
        "users":[{
            "username": "admin",
            "roles": ["administrator", "regular_user",],
            "password":"secret_password"
        }],
        "storage": { "type":"memory" }      /* keeps this configuration loaded in memory
                                               during the request cycle; all other pocket-knife
                                               storage options (SQL, CSV, etc.) are available */
    },
    "acl": [
        "admin can * *\/*",                 /* admin can access all verbs on all resources */
        "* can GET magazine\/*",            /* everyone can GET magazines and pages */
        "admin can GET page\/14",           /* only admin can get page 14 */
        "administrators can OPTIONS page\/*" /* the administrator can use the OPTIONS verb */
    ],
}
```

3. *Facebook Authentication*: the growth of OAuth as an authentication tool merited some attention in *pocket-knife*. As a proof-of-concept, Facebook OAuth 2.0 is implemented as an authentication mechanism (see *SecurityAuthenticationFacebook*) although access levels to the REST service are still controlled locally with *SecurityAcl*; a default "facebook-user" role is assigned to each user authenticated in this manner. One issue with OAuth, at least with Facebook, is its dependence on the browser: the user is redirected to the Facebook login page to enter their credentials. If the user is not communicating with a browser (e.g. using a script), this authentication type will fail. This issue, and the conflict it raises with REST architectures, is described in depth by [Richardson]. Facebook has a device-based login API but is only available to certain developer clients.

4. *Caching*: HTTP fully supports caching content on the client side with HTTP headers like *Expires*, *Cache-Control*, *Last-Modified*, and *ETag*. *pocket-knife* makes a substantial effort to use these methods for informing the client of what is cacheable. For example, once a resource is accessed by a GET action, it will be cached by default in the *cache* directory and used to issue *304 Not Modified* codes or up-to-date data to the client as necessary. As expected, non-idempotent actions like PUT, POST, or DELETE will clear affected resources from the cache. Though other frameworks do implement caching, most require developers to write code using *get()* and *put()* methods; *pocket-knife*, on the other hand, implements caching transparently as a feature in all resources.
5. *Content-Type Detection*: Using the *Accept* HTTP header from the client, *pocket-knife* will attempt to match the requested MIME content type. In other words, the server can return any resource in a variety of formats. Using the *Representation* class (in keeping with the REST model), the currently supported MIME content types are: *application/json*, *application/octet-stream* (raw file), *application/xml*, *application/x-www-form-urlencode* (POST parameters format), *multipart/form-data* (POST parameters with uploaded files), *text/html*, and *text/plain*.
6. *Storage*: Most resource requests will require some type of storage across requests. The *StorageInterface* system allows resources to access databases or data stores using common methods. Currently, *pocket-knife* implements storage for: SQL (all versions available using PHP PDO), MongoDB, CouchDB, JSON files, CSV files, in-memory data structures, and Amazon S3



(in progress). The graph below explains the layers that the HTTP request must pass through to reach to the storage layer (because the framework handles the security, representation, and storage layers through the use of configuration files, the developer is freed to spend most time on their application code):

7. *Resource Patterns*: Though some of the frameworks above implement features such as active record and ORM, *pocket-knife* goes further along these lines. This bears further explanation in the next section.

Resource Patterns

One of the goals of the *pocket-knife* project was to make RESTful resources simple to interact with. Recall that in the REST model, a resource is the named abstraction for any piece of data and that it has a corresponding identifier (in the form of a Uniform Resource Identifier, URI) and representation (the data response sent to the client, of any content type). This was implemented in *pocket-knife* using the *Resource* class and its methods (e.g. *getURI()*).

In building solutions, developers notice recurring patterns in the resources they implement. In implementing a *Library* resource that contains *Book* resources (see example later on), we notice similarities to a *Group/Users* solution or an *Album/Pictures* solution—they all implement *List/Item* relationships. This is not a new discovery and is covered in detail by Tilkov:

Resource Type	Description
primary resource	core concepts of the modeled domain, e.g., pictures and albums
subresource	part of another resource, which should also be addressable directly
list resource	list of all concrete resources of a primary resource
filter resource	list of concrete resources with desired properties
projection resource	contains only a subset of attributes of another resource
aggregation resource	aggregates attributes of different resources to reduce the interaction amount
paging resource	splits large resources in different pages
activity resource	stands for one single step of a workflow

Table 6: Tilkov's Categories, Translated in [Schreier]

Though Tilkov seems to cover all available resource types, it is interesting to consider whether his list is comprehensive; perhaps further examination would reveal that his *primary resource/subresource* relationship could be sub-categorized into relationships like *Tree/Branch/Leaf*, *Object/Properties*, or the above *List/Item.* This is precisely the direction taken in implementing *pocket-knife*; by making the base *Resource* class generic enough, sub-classes can build out resource patterns to position developers closer to a solution.

The concept of resource patterns bears further investigation and is not fully implemented in *pocket-knife* but the following patterns are available or in progress:

1. *ResourceList, ResourceItem*: this pattern relates items to lists. All of the HTTP verbs work with *ResourceList*, but apply to the entire list; for example, a GET will return the list of *ResourceItems* from storage. It contains distinctive properties: the class extending *ResourceItem* must be set as a string in *ResourceList→item_type* and *ResourceList* will return items in *ResourceList→items*. Example usage: library/book.
2. *ResourceTree*: this pattern relates resources to each other in a tree structure. It uses *ResourceTree→branches*, an array, to link to branch nodes; each level of the tree must be accessed from the URL tokens in a request like, *api.php/tree /level-1/level-2/level-3*. Example usage: flow chart.
3. *ResourceGeneric*: this pattern involves single resources that are non-editable. It only implements a GET method; it serves as an entry point into the API for resources that otherwise would require extensive scripting. Example usage: access files, build an HTTP proxy.
4. *Cache*: the class that does caching for the *pocket-knife* system is also a descendant of *Resource*. It implements all HTTP verbs though it is not publicly accessible as this raises security concerns for

private cached resources. It represents a pattern commonly built in web services and web applications and can be extended; eventually the *Settings* class will become a resource pattern as well. Example usage: other cache systems.

5. *ResourceTagged* (in progress): this pattern adds tagging to an already-built resource and allows for relating resources by tag. It uses *ResourceTagged→tags*, an array, to store tags. Much like Tilkov's filter resource, it would implement the HTTP verb GET to return a list of similarly tagged resources using *api.php/[resource]/tag:[tag-name]*. This borrows concepts from [Brooks et al] in their discussion of tagging and hierarchical clustering. Example: link tagging, article tagging.

By extending the classes in these patterns, the developer can quickly build a working *List/Item* relationship, as seen in the example below. The reader may notice that each pattern does not mandate a storage format (although each class does by default include one); this is purposeful, allowing the developer to choose the optimal storage method for the problem at hand.

Example

To illustrate the use of *pocket-knife*, we create a problem scenario in which we build a *Library* resource that contains multiple *Book* resources. The 148-line solution begins with a web service endpoint, a file which we call *index.php*:

```
require '../../start.php'; /* load pocket-knife files
require 'Book.php'; /* load Book class
require 'Library.php'; /* load Library class

/* if no tokens are added to the URL, the user probably wants to view the library
try{ WebUrl::getTokens(); }
catch(Error $e){ WebHttp::redirect(WebUrl::createAnchoredUrl('library')); }

/* start service
$configuration = new Settings();
$configuration->load('config.json'); /* load settings from a JSON file
$service = new Service($configuration);
$service->execute();
```

Next we create the configuration file, *config.json*:

```
{
    'acl': true, /* allows any and all requests in the service */
    'representations': ['text/html', 'application/json', 'text/xml'] /* only allow these MIME
types */
}
```

Finally, we create the class files *book.php* and *library.php*, which are classes implementing *ResourceItem* and *ResourceList*, respectively. Due to length, only *library.php* is shown here as it explains in enough detail how to build resources:

```
class Library extends ResourceList {
    // publicly accessible properties will be viewable/editable by clients
    public $name = 'Savannah Public Library';
    public $location = '2002 Bull Street, Savannah, GA 31401';
    // we declare the ResourceItem type here, see book.php
    protected $item_type = 'Book';
    // this property defines where the data will be stored, see Storage folder for more
    protected $storage = array('type' => 'json', 'location' => '../data/books.json');
```

```
    // because Library is such a bland type, we do not override the GET/PUT/
    // POST/DELETE methods defined in ResourceList; we do, however, use OUTPUT_TRIGGER,
    // fired before the Representation is sent back to the client, to add some
    // templating to our HTML responses.
    public function OUTPUT_TRIGGER(Representation $representation) {
        // delete the entire cache for non-idempotent methods
        if( WebHttp::getMethod() == 'PUT' || WebHttp::getMethod() == 'POST' ||
WebHTTP::getMethod() == 'DELETE'){
            Cache::clearAll();
        }
        // add HTML templates and redirects
        if ($representation->getContentType() == 'text/html') {
            switch (WebHttp::getMethod()) {
                case 'GET':
                    $representation->setTemplate('template-library.php', WebTemplate::PHP_FILE);
                    break;
                case 'OPTIONS':
                    $representation->setTemplate('template-options.php', WebTemplate::PHP_FILE);
                    break;
                case 'PUT':
                case 'POST':
                case 'DELETE':
                    // non-idempotent requests will redirect to the main page
                    WebHttp::redirect(WebUrl::createAnchoredUrl('library', false));
                    break;
            }
        }
        return $representation;
    }
}
```

After building both *book.php* and *library.php* and any necessary templates, the web service is ready. The *Library* resource is available at http://localhost/pocket-knife/www/demo/index.php/library and, once they are created (i.e. POSTed), *Books* are available at URLs like http://localhost/pocket-knife/www/demo/index.php/book/1. The initial URL will change depending on the server *pocket-knife* is installed in, but this example is indeed available in the *www/demo* directory. The code is hosted on GitHub at http://github.com/andrewsbrown/pocket-knife.

Another Example

[Zhang's] MentalSquares model for diagnosing mental illness was implemented in a further demonstration of *pocket-knife's* flexibility. Though the code is too lengthy to present here, it is available at http://github.com/andrewsbrown/mental-squares. The approach built on the library example above to link two different types of objects, *Patients* and *Tests*, into a system for recording psychological evaluations. A screenshot of the system starting point is below:

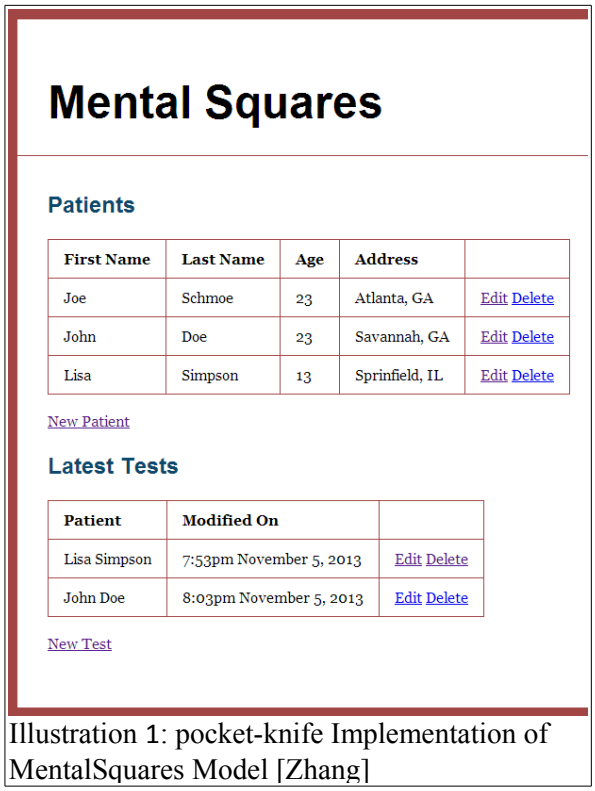


Illustration 1: pocket-knife Implementation of MentalSquares Model [Zhang]

Future Work

During the development of *pocket-knife*, several avenues for future work opened up:

1. Add more resource patterns, especially *ResourceTagged* and *ResourceActivity* (as part of a *ResourceWorkflow*).
2. Simplify and re-factor the *Service→execute()* method; all security, caching, and representation classes have hooks here.
3. Add more storage, security, and representation types; this is an on-going process as new technologies (e.g. Amazon S3, which is still incomplete) can be integrated into *pocket-knife*. Some possibilities include adding Atom RSS as a representation type and a token- or certificate-based authentication method as a new security type.
4. Implement actual HTTP request/response testing; the PHPUnit tests currently access the *Service* methods for testing but should be testing the actual HTTP payloads coming from a live server.
5. Add Hypermedia as the Engine of Application State (HATEOAS) functionality to resource patterns. This would add link references to resource representations so a user could theoretically discover the resource structure just from the HTTP data (e.g. in a *List/Item* pattern, *ResourceItems* would have links to their parent *ResourceList* and the next/previous *ResourceItems*).
6. Implement a web service management UI to provide a graphical interface for routine administrative tasks like inspecting/clearing the cache, viewing logged-in users, etc.

7. Implement Web Application Description Language (WADL) functionality to provide a machine-readable syntax for the OPTIONS HTTP method, which currently just lists the available HTTP methods and properties available for a resource.
8. Determine how to implement TRACE and PING; they are HTTP verbs but they have little use in the RESTful model.
9. Test *pocket-knife* performance against other web service frameworks. The difficulty with this is determining a fair test to run, since in general the frameworks listed above perform some trade-off between ease of programming and framework size.

Conclusion

The *pocket-knife* project meets the goals stated in the introduction. With the exception of HATEOAS, which can be added at a later date, the project fits the REST constraints. Even with the relative complexity of resource patterns, the Resource class and subclasses are still easy to understand and interact with. And after significant pruning of unnecessary classes, the project remains focused: a feature-rich yet fast RESTful web service framework.

The lessons learned are deeper than just the written PHP code and the organization of a large project. The time spent thinking about programming—the common problems, the common patterns—was invaluable in developing the author's mindset towards computer science. The search to find programming shortcuts and make them intuitive and natural in the framework forced considerable research into how things are solved by others. In the end, building *pocket-knife* was similar to making a tool, albeit one quite complex and with lots of features.

Bibliography

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, California, 2000.
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Netcraft. “PHP Just Grows and Grows”. Netcraft.com, 2013.
http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html

Netcraft. “October 2013 Web Server Survey”. Netcraft.com, 2013.
http://news.netcraft.com/archives/2013/10/02/october-2013-web-server-survey.html

Alex Rodriguez. “RESTful Web services: the basics”. devloperWorks, IBM, November 2008.
https://www.ibm.com/developerworks/webservices/library/ws-restful/

Roy Fielding, et al. *Hypertext Transfer Protocol – HTTP/1.1*. The Internet Society, June 1999.
http://www.ietf.org/rfc/rfc2616.txt

Kin Lane. “Short List of RESTful API Frameworks for PHP”. ProgrammableWeb.com, September 2011.
http://blog.programmableweb.com/2011/09/23/short-list-of-restful-api-frameworks-for-php/

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. Dissertation. University of California, Irvine. AAI9980887.
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf

Silvia Schreier. 2011. Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design* (WS-REST '11), Cesare Pautasso and Erik Wilde (Eds.). ACM, New York, NY, USA, 15-21. DOI=10.1145/1967428.1967434 http://doi.acm.org/10.1145/1967428.1967434
S. Tilkov. *REST und HTTP: Einsatz der Architektur des Webs für Integrationsszenarien*. dpunkt.verlag, 2009.

Daniele Bonetta, Achille Peternier, Cesare Pautasso, and Walter Binder. 2012. S: a scripting language for high-performance RESTful web services. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (PPoPP '12). ACM, New York, NY, USA, 97-106. DOI=10.1145/2145816.2145829 http://doi.acm.org/10.1145/2145816.2145829

Cesare Pautasso and Erik Wilde. 2010. RESTful web services: principles, patterns, emerging technologies. In *Proceedings of the 19th international conference on World wide web* (WWW '10). ACM, New York, NY, USA, 1359-1360. DOI=10.1145/1772690.1772929 http://doi.acm.org/10.1145/1772690.1772929

Alexandros Marinos, Erik Wilde, and Jiannan Lu. 2010. HTTP database connector (HDBC): RESTful access to relational databases. In *Proceedings of the 19th international conference on World wide web* (WWW '10). ACM, New York, NY, USA, 1157-1158. DOI=10.1145/1772690.1772852 http://doi.acm.org/10.1145/1772690.1772852

Leonard Richardson. 2010. Developers enjoy hypermedia, but may resist browser-based OAuth authorization. In *Proceedings of the First International Workshop on RESTful Design* (WS-REST '10), Cesare Pautasso, Erik Wilde, and Alexandros Marinos (Eds.). ACM, New York, NY, USA, 4-8. DOI=10.1145/1798354.1798377 http://doi.acm.org/10.1145/1798354.1798377

Christopher H. Brooks and Nancy Montanez. 2006. Improved annotation of the blogosphere via autotagging and hierarchical clustering. In *Proceedings of the 15th international conference on World Wide Web* (WWW '06). ACM, New York, NY, USA, 625-632. DOI=10.1145/1135777.1135869 http://doi.acm.org/10.1145/1135777.1135869

Zhang, W-R., Pandurangi, A.K., Peace, K.E., Zhang, Y-Q. and Zhao, Z. 2011. MentalSquares: A generic bipolar Support Vector Machine for psychiatric disorder classification, diagnostic analysis and neurobiological data mining. In *Int. J. Data Mining and Bioinformatics*, Vol. 5, No. 5, pp.532–557.