Apskel

(Application Skeleton -- PHP Web Application Development Framework)
(DRAFT)

INSTALLATION AND OVERVIEW

Advantages

The general advantages and disadvantages of using Apskel over any other framework depends upon ever changing factors that vary from one to another. However, it might be helpful to consider the advantages of using Apskel over direct coding a web application in PHP. And consider the same for any alternatives to Apskel. The major effort with Apskel is in making web application development fast and easy to learn, develop, and maintain while also being robust and scalable up, down, and outward. The following covers each advantage.

Auto-Generated Code. Apskel includes tools for designing and auto-generating all the general overhead code. You only need to describe the web requests it will service and the database tables. Apskel manages migrations through the development/deployment cycle to substantially reduce the risk of problems. Code is also auto-generated for conversion of database tables/data from one version to any thereafter.

Configuration Tool. Although you may edit configuration files directly, a tool exists to simplify and automate the process.

Automatic Version Migration Management.

Configurable Policies for Use of Multiple Database Servers.

Cleaning and Translation of Request Parameters.

Automatic RESTful API & Associated Documentation.

Automatic Conversion of Request Output to a Number of Different Formats.

Automatic Identification of What Version and Environment Running Code is It.

Installation

After having acquired the Apskel installation package (apskel-{version}.tgz where {version} is the version you acquired), put it in the directory where it will live. For the purposes of these documentation, the (highly advisable) /var/www/ location is presumed. If that is not where you want the files to live then adjust “/var/www” throughout these documentation, accordingly. Next,

```
cd /var/www
tar -xzpf ./apskel-{version}.tgz
```

Ok. It’s basically installed. To use though, you will need to have Apache2 installed with mod_php and mod_rewrite enabled. You will also need an accessible MySQL server. And then, you will need to build or install a web application into it.

OVERVIEW

The environment refers to the settings used and the application version refers to the case-base used. Both are identified by the domain name of the URL or a substring thereof.

URL Composition

The URL may be composed of a protocol, an address, a path, and a query. Each of these are described, as interpreted by Apskel, in turn below:

Protocol. The protocol spans from the beginning of the URL to just before the first "://" character string found. Apskel recognizes "http", "https", and "cli" (meaning the command line).

Address. The address is the domain name (DNS name). This is a period delimited text string beginning after the first "://" and ending just before any "/" or "?" character. Apskel uses this (or a substring thereof) to identify the environment/version to use.

Path. This is the section of the URL beginning immediately after the domain name and up to any first "?" character. The path is separated by "/" characters into path nodes. Normally, the first and second of these specify the module and controller/method, respectively. If however, there is no such module exists then the session default module is used in its place. If the controller/method doesn't exist within the module then the module's default controller/method is used. Otherwise, all additional path nodes are interpreted as query parameters. Where in the form "field=value", "value" is assigned to query parameter "field". Where there is no "=" character in a path node, the query parameter by that name is created but assigned an empty string. In either case, if such the query parameter already exists then it is overwritten.

Query. The query begins after the "?" ends the path. This is the standard HTTP1.1 query string, specifying "GET" query parameters as per that standard. Note, however, that query parameters entered as part of the path will override any of these or any "POST" parameters.

The file structure is as follows. {TBD: Elaborate.. what, why..}

```
.
├── application_0.0
│   ├── architecture
│   ├── framework.php
│   ├── modules
│   │   ├── agent
│   │   │   ├── agent_brain.php
│   │   │   ├── agent_controllers.php
│   │   │   ├── agent_models.php
│   │   │   ├── agent_registration.php
│   │   │   ├── agent_views.php
│   │   │   ├── resources
│   │   │   │   ├── interface.js
│   │   │   │   └── main.css
│   │   │   └── templates
│   │   │       ├── interface.html
│   │   │       └── save-interface.html
│   │   ├── controllers.php
│   │   ├── models.php
│   │   ├── user
│   │   │   ├── resources
│   │   │   ├── templates
│   │   │   ├── user_controllers.php
│   │   │   ├── user_models.php
│   │   │   ├── user_registration.php
│   │   │   └── user_views.php
│   │   └── views.php
│   ├── resources
```

```
│   │   ├── development
│   │   ├── jquery-1.8.1.min.js
│   │   ├── main.css
│   │   ├── production
│   │   └── staging
│   └── templates
│       ├── development
│       ├── production
│       └── staging
├── environments_0.0
│   └── production_settings.php
├── identification.php
├── LICENSE
├── notes.txt
├── README.dm
├── tools
│   ├── build
│   └── configure
└── webroot
    ├── favicon.ico
    ├── index.php
    └── robots.txt
```

THE STEP-BY-STEP PROCESSING OF A WEB REST

A web request is initially received by the index file ("~/webroot/index.php"). The index instantiates the Identity and Framework classes. Upon instantiation, the Identity class determines what application version and environment to use based on a mapping from the domain name (or a substring thereof) used by the request. The version of the Framework class used, is based on the the application version. The Identity object is passed into the Framework class upon instantiation. Finally, the "serviceRequest" method of the Framework class is called and its results are sent back to the client (browser or command line).

In short:

identify -> framework (interpret request -> call module's request handler -> send results to view) -> echo results to client

Step 1 -- Identification

Upon instantiation of the Identity class, the URL is parsed into the request protocol, domain name, path, and query parameters. The "mappings" attribute of the Identity class is the array used for mapping the domain name to the application name, version, and environment. The following example illustrates:

```
$this->mappings[] = array(
        'recognizer'=>'apskel.dev',
        'application'=>'Apskel',
        'environment'=>'development',
        'version'=>'0.0'
);
$this->mappings[] = array(
        'recognizer'=>'apskel.stg',
        'application'=>'Apskel',
```

```
        'environment'=>'staging',
        'version'=>'0.0'
);
$this->mappings[] = array(
        'recognizer'=>'apskel.com',
        'application'=>'Apskel',
        'environment'=>'production',
        'version'=>'0.0'
);
```

The “recognizer” element is a string that, if is found to be (or be a substring of) the actual request’s domain name, will match. The first match found in sequential order is the one used.

The “application” element sets the global name for the application. This is for anywhere the name of the web application is required, such as for logging purposes.

The “environment” element is intended to represent the migration stage of the application. For example, “development”, “staging”, or ”production. Each environment will have its a separate settings file (database connection credentials, etc) and resources (such as css style sheets, images, or javascript files) for the environment will override those from the application or module levels. For example, you might want a different background colors on pages for development, verses staging, verses production to prevent confusion as to the context a user is currently in while viewing pages.

Finally, the “version” element of the “mappings” array tells what application version to use. The Framework class code and application level resources, views, and modules are determined by this. All are derived from under the “~/application_0.0” directory, where “0.0” is the version. Of course, the version is just a text string and so could be anything you like. Similarly, the environment settings file is “~/environments_0.0/production_settings.php” where “0.0” is the application version and “production” is the environment (as mentioned in the previous paragraph).

Step 2 -- Main Request Interpretation

First, any mappings from path aliases are made. These may be specified in the environment settings file (which in turn, may reference a database table for them). If an alias matches, the path is converted from the alias to whatever the alias is set to change it to. Processing thereafter, continues as if it were the path received from the web request. The purpose of this is to provide a means for shorter, simpler, more intuitive, or multiple paths than just the standard /module/request (MVC-like) default structure, as described in the next paragraph.

Second, request variables are determined. These may be standard HTTP1.1 POST or GET variables. However, they may also be interpreted in an Apskel specific format. The elements of the path are right-slash separated, as per the HTTP1.1 standard. If the first of these is a recognized module name, it is interpreted as such. If the second of these is a recognized request under that controller, it is recognized as such. Anything else or farther to the right is interpreted as a parameter specification in either the form of “param” or “param=value”. In the first form, the parameter named “param” is assigned a value of null. In the second form, the parameter named “param” is assigned the value of “value”. This enables a controller/module’s request the flexibility to interpret the path any way it chooses while simultaneously providing a simple standard for writing URL-based RESTful calls.

So a very standard MVC-like request to logout a user and then redirect to google might look like this:

http://somesite.com/user/logout/redirect=https://google.com

If the module request is not specified then the current default is used. If the module is not specified, then the default module is used. In either case, all else to the right would be interpreted as a parameter... even a misspelled module or request name.

Step 3 -- Calling the Module's Request

Modules are called through the Framework's "serviceRequest" method. If called without any parameters, the request is considered to be a "main request", meaning the URL components are used to determine it and a finalized result is returned as it is to be presented to the client (browser or command line). Otherwise, the result is returned in a standard data structure format or whatever the module's request otherwise determines to return. The purpose of this is for module's to make requests from other module, in a more convenient internal way.

In any case, the following sub-steps are taken in calling a request:

(a) If the "resource" module, grab and return the specified resource file in priority of environment, application version, module (where the first overrides the last). For css, this is a concatenation. For all other file types, it is a replacement. The "resource" module is a special built-in functionality. In this case, the mime-type is set by file extension and its contents returned to the index for outputting to the client. No further steps are executed.

(b) In any case other than the above, the existence of the module and its request within it, are verified. If this fails, an appropriate error is returned in a format determined by the protocol (cli as text or http/https as HTML). This will auto-produce any relevant RESTful API documentation (using the module's registration specification).

(c) The parameters and sanitized and defaulted or marked as missing, if required by missing--as per the module's registration specification. Sanitization runs each value through the PHP addslashes() function for general protection against various kinds of code injection attacks. Also, any extraneous parameters (those not registered for the request in the module's registration specification) are discarded. If a required parameter is not present, it is added to a "missing" parameter list. If a parameter is expected and not provided but a default value is provided for it then it is added with the specified default value (and not added to the missing list). Note that, any parameters in raw form are still separately available through standard PHP avenues such as the $_GET, $_POST, or $_REQUEST arrays.

(d) If the parameter named "fresh" doesn't exist then it is automatically created and assigned the value of boolean false. This is to let a module request know if is continuing communication with the user, or if the user just showed up here.

(e) The module's request handler is called, passing in the sanitized parameter array and the missing parameter array.

Step 4 -- Within the Module's Request Handler

A module's request handler, if build off the auto-generated model, will begin by setting all expected parameters to default values. Then, it will extract the actually received values from the input array holding them as like-named local variables. The purpose of this is to (a) have a place to set any defaults no specified in the module's registration specification--this is sometimes useful for required but missing parameters; and (b) just to have an awareness of

what the input parameters are for convenience while coding in this request handler.

Next, there is an “if” statement as to whether or not the “fresh” parameter is true or not. Behavior is often going to differ based on this, so it’s there by default when auto-created.

Finally, after the request handler’s logic (initially a TODO comment), the return statement is made. If you return a string, this will be interpreted as HTML and therefore wrapped and returned as such. This is for writing small simple pages, if desired. Otherwise, the request handler should return an array of two values (the following array elements):

(0) The first element of the return value may be either a string (representing pre-formatted text) or an associative array of view parameters. Either way, it is interpreted according to the format specified by the second element of the return value.

(1) The second value should always be an associative array with the ‘format’ value specifying the kind of view. If that view type requires any additional specification then those should also be added to that associative array.

Step 5 -- The View

The view is called after the module’s request handler. Whatever is received from it is analyzed to determine what to do with it. If not an array, it is presumed to be direct HTML and returned as such. Otherwise, the return is interpreted as an array of two values: a “response” and a “format”. The format determines what to do with the response--what kind of view formatting to perform. If no format is specified, the format is presumed according to the protocol (HTML for HTTP or HTTPS else plain text for CLI (command line)). Either way, the following formats are supported and appropriate mime-types are set for each:

“preformatted” -- just return the raw response, using the format’s also provided “mim-type” parameter to set the mime-type as.

“text” -- convert the response to text (an array will show indented accordingly)

“html” -- an array will format as HTML indented bullets; if a main request, main HTML wrappers are added.

“xml” -- an array will be converted to XML; if a main request, XML header is provided.

“json” -- an array will be converted to JSON

“view” -- the specified “view-file” will be called, its like-named class instantiated and method called, then return text returned. It must set any mime-type itself, else default HTML.

“template” -- the “template-file” is loaded, “{{parameter}}” formatted parameters within it are replaced by like-named array elements. Sub-arrays are presumed that the key is the name to a sub-template to put its values within using the format {{parameter:file_name.html}}. Also prior to populating a template, any {{=file_segment.html}} referenced files are loaded and inserted in place. Recursive levels of data are supported.

“direct-html” -- presume the result is pre-formatted HTML; if a main request, wrap with HTML headers.

If a main request, the results of this are wrapped appropriately (e.g. HTML will have <html> .. </html> around it) and it is returned. Hence, the index will then echo this back to the client. If not a main request, then normally just the response and format arrays will be returned. This is because it will normally be another module making non-main request calls--aka a sub-request.

TODO: were to put this paragraph:

For Apskel, the term “module” refers to a certain set of components packed together under an application version’s “modules” directory, such as “~/application_0.0/modules/user”.

At a minimum, this should include a registration file and a controller file but should normally also include also a models file, possibly a views file, a resources directory, and a views directory. This is designed to support vast latitude in the design philosophy and/or methods used in each module.

DEPLOYING & ADMINISTERING A WEB APPLICATION

Pre-Deployment Considerations

Installation

Configuration

Deployment

Post-Deployment Considerations
DEVELOPING A WEB APPLICATION

Required Design Factors

Creating a New Version and New Modules

Creating New Database Tables

Coding Controllers, Models, and Views

Bringing the Application Components Together

Packaging for Deployment

USER MODULE

AGENT MODULE

The Agent module provides an agent for chatting with users in natural languages via reasoning logic that can be built or modified by users. Conversations are centered around topics. A topic may be created such that invoking it will automatically inject a certain set of contexts and an initial simulated user statement. A topic may be invoked either through specific conversational interactions or else by a topic-specific URL. This is intended as an analogue of Wikipedia articles.

Overview

The URL by which one arrives at the agent determines the initial topic. From here, a user statement may be sent to the agent. The agent identifies the meanings of the user statement, determines its reaction, and returns a response. The reaction is determined by conditions associated with the user statement and/or what is in memory. Execution of a reaction may modify what is in memory and/or return a response. The response may be comprised of two parts: verbal and non-verbal. The verbal part is textual (consisting of any number of lines of text). The non-verbal part may consist of special cues to the client. Among these are those to enter a certain statement for the user, if the user does, does not, or regardless of whether or not the user does or does not, within a given period of time or by a certain date/time. Otherwise, they are cues pertaining to the processing of the last user statement, such as to indicate if something was put to memory, if the agent it anticipating something from the user, etc. These may be useful for emoticons but are not vital.

Identifying the Meaning of a User Statement

A user statement is a simple line of text provided by a user. Numerous meanings exist in an agent database, each of which has a "recognizer" pattern. The meaning of the user statement is identified by matching against its recognizer. The search for the "best match" compares statement to recognizer in the following succession of ways:

Full Punctuation and Casefulness
Full Punctuation and Caseless
No Punctuation but Caseful
No Punctuation and Caseless

For each of the above, the search occurs from the longest to shortest recognizer in number of characters, wildcard variables counting as 1 character each. A wildcard variable in a recognizer matches any text within its place, in the user statement. For example, the recognizer "My favorite wine is [wine]." would match the user statement "my favorite wine is a riesling", such that the wildcard variable [wine] is assigned the value "a riesling". This variable may be used in reaction's conditions and actions. For example, to respond by saying "What is it about [wine] wine that you like?", such that the user sees "What is it about a riesling wine that you like?".

After the above methods of trying, if no meaning can be identified portions of the user statement (from longer to smaller) are compared to find a match, through the same process mentioned above. And finally, if still no match is found then the user statement is re-interpreted as "What is the meaning of: [statement]?" where [statement] is the complete user statement. Therefore, a meaning should exist to deal with unidentifiable user statements in this context.

Selecting a Reaction to a Given Meaning

Each meaning may have different "reactions", each of which has a "priority" (zero being the top and higher numbers being lower in priority), "conditions" (that may evaluate to either true or false), and "actions" (a set of things to do, in order). Reactions valid for use are those with conditions that evaluate to true. Among concurrently valid reactions, one is selected according to the meaning's paradigm: natural, cyclic, or random:

Natural. This selection paradigm is designed as the most similar to human nature and is the default. Among concurrently valid reactions, the least recently used with the highest priority is selected. It's actions are then executed, in sequence. That is, if the user makes

a statement that matches the same meaning, multiple times within a session, each different reaction will occur in order of priority. When all are used, the cycle will repeat. However, gradually this recency effect will dissipate such that those used will gradually drift back to their original places in order of priority. For multiple reactions of identical priority, the order is undefined.

Cyclic. Concurrently valid reactions are simply chosen in order of priority, where identical priorities are undefined order.

Random. Concurrently valid reactions are randomly picked and priority is ignored.

Wildcard Variables

Any textual statement may contain wildcard variables, specified by its name encased in brackets.

In a textual matching operation, a wildcard variable could be used as a wildcard to match the whole statement except for any text within the area of the wildcard. For example, the recognizer “My name is [name].” will match statements such as “My name is Bob.” and “My name is Joe.”, in each case assigning the respective name to the wildcard variable [name]. H

Where might the following forms be necessary to use verses presumed form?
[>name] [<name] [:name]

Reaction Conditions

A Reaction’s Conditions section should comprise of a single cohesive condition statement that evaluates to true or false. Blank is interpreted as true. The conditional statement may be composed of standard “and”, “or”, “not”, and parentheses logic with the addition of the following specialized functions, usable before and after “and” and “or”, or after “not”. A condition function is always enclused in parentheses and its name starts immediately after the opening parenthesis, as follows:

- (IS {=|>|<}n "..")
  True if n, greater than n, or less than n (respective to (nothing), >, or < being used) number of “..” matches are found in memory. The “=” symbol may be excluded as it is the default presumption.
- (ISALL ".." {=|>|<} "..")
  True if, for all of the first “..” matches, a match from the second “..” exists with like-named variables values greater than, less than, or equal (respective to <, >, or = being used).
- (ISANY ".." {=|>|<} "..")
  True if, for any of the first “..” matches, a match from the second “..” exists with like-named variables values greater than, less than, or equal (respective to <, >, or = being used).
- (CAN ”..”)
  True if a path from the current status to the given agent response (“..”) can be plotted.

In the above, the lowercase “n” (excluding the quotes) represents a literal number. The “..” (including the quotes) represents a textual statement with optinal bracketed wildcard variables in it. Matched memories assign to the variables. Order of multiple values is ignored in comparisons. Where both are numeric in a comparison, the comparison is performed by their numeric (not textual) equivalents.

One future prospect is to add support for an “(EXCLUDING “..”) sub-function. For example, the following ISANY function would consider the matches of the first “..” only after first excluding any matches from the second “..”.

(ISANY “..” (EXCLUDING “..”))

Conditional functions are extensible. The “agent_conditionals.php” file should include definitions for all. Each entry for a function includes an expression to match the conditional function and its parameters plus a PHP function to evaluate it. That function must return a boolean true or false, only.

Reaction Actions

A Reaction’s Actions section is a sequence of action commands. Action commands are not case-sensitive. The parameters are always text in quotes, however, these quotes may include variables. A variable is specified inside of brackets (i.e. “[“ and “]”). In an action that outputs, the variable’s values are written. In an action that inputs, the variable’s acts as a wildcard and its value/s is/are thus assigned. Multiple values are returned as comma delimited. Agent version 1.0 action commands are listed below:

- SAY “..”
  Outputs a textual message to the user.
- REMEMBER ".." [{FOR|UNTIL} ".."]
  Outputs a textual statement to long term memory. Optionally, the number of minutes "FOR" may be specified or a
  date/time "UNTIL"
- RECALL ".." [{AND|OR}]
  Inputs any variables in the string from long term memory. Multiple finds will be assigned as comma delimited values. If "AND" or "OR" are appended then the last value (if more than one match) will be prefixed with it. Wildcard variables already collected are written in as already assigned. Wildcard variables not collected are sought out for recollection from memory.
- FORGET ".." [AFTER ".."]
  Removes all matching long term memories, variable names are ignored as variables are only used as wildcards. If an "AFTER" value is given, it is used to scheduled to be forgotten after the given date/time or minutes to pass.
- EXPECT “..” AS “..”
  Schedules that, if the next user statement is as specified, to interpret that as the other specified statement. The first statement is an input (assigns to any variables) and the second statement is an output (writes out contents of any variables).
- INTERPRET AS “..”
  Performs as if the user also just entered the given statement. This does not exclude any actions under the current reaction. All outputs to the user are concatenated, in order.
- WHAT IF “..”
  Notes the response statement and long term memories that would result if the given user statement were entered under the current status.
- HOW COULD “..” ELSE “..”

Solves for and notes what sequence of user statements (if any) would lead from the current status to the specified response statement. If not solvable then interpret as the else statement.

- WORK TOWARD ".." ELSE ".."
  Solves for what sequence of interactions (if any) would lead from the current status to the specified response statement and takes the next actionable step to try and get there. If not solvable then interpret as the else statement.
- AFTER n INTERJECT ".."
  Arranges that after n (minutes or date/time), the statement ".." is assumed as if made by the user. This enables making interjections to the user at designated times or after specified periods of time passed.
- EXPECT BY n ELSE ".."
  If no response by user in n seconds then interpret as user statement being "..". This enables giving the user a period within which to respond, else taking other action.

The action command system is also extensible. Two particularly likely prospects for future actions include those to extract targeted data from Wikipedia and math functions (simple arithmetic and the GNU Science Library). The file "agent_actuations.php" should hold an entry for each action type. Each entry comprises of a pattern to match the action and its parameters plus a PHP function to perform its work, accordingly.