What is SPUD?

SPUD is a mechanism for saving and retrieving information about site users, without requiring them to log in or set up an account. User's are assigned a SPUD via a cookie containing a unique identifier. Whenever someone enters information about themselves on a form (signup, contribution, etc.) it is saved in their SPUD. The next time they view a SPUD-aware form, the form can be pre-filled with the saved information.

SPUD allows for different pieces of information to be saved for different lengths of time. Some information, such as name and address, should not be retained for too long. Coming back to a site you haven't visited for a year and having it remember your name is likely to be upsetting to many people. Less personal information (like zip code or state) can be retained for longer periods of time without causing any great distress to the user.

In addition to saving a standard set of information about a site visitor, SPUDs can also contain application specific data/preference settings. For example, a blog application could use SPUD to remember a user's preferred comment display style (threaded or unthreaded) even though the user created an account.

Of course, by virtue of SPUDs being cookie based, data is attached to a particular browser. A user visiting the site on another computer will not have his or her SPUD data displayed.

Using SPUD

First, include the `spud` class file:

```
require_once 'utils/spud/spud.class.php';
```

Next, get a `spud` object:

```
$spud = spud::start();
```

If you want to take some action (such caching a page) based on the presense of a SPUD, you can say:

```
if (spud::has_spud()) {
  $spud = spud::start();
  ... do something ...
} else {
  ... do something else ...
}
```

Once you have a `spud` object, you can call the `get` object function to access and change standard values:

```
$firstname = $spud->get ('firstname');
print "Hello, ".($firstname ? $firstname : 'Friend')."!\n";
```

You can also pass an array to `get` . If you do so, the result will be an array containing all of the values you requested that are present in the SPUD:

```
$values = $spud->get (array ('firstname', 'lastname'));
print "Hello, ".$values['firstname']." ".$values
['lastname']."\n";
```

To set a value, you can call `set` object function:

```
$spud->set('firstname', $_POST['firstname']);
```

You can also set multiple values at once with the `set_from_array` and `set_from_object` functions. These will locate all array elements or object

variables that have a name that matches one of the standard SPUD variables. It will then set the values in the SPUD accordingly. For example:

```
if ($form->validate()) {
    $form_values = $form->exportValues();
    $stg_signup = new stg_signup();
    $stg_signup->assign_from_hash($form_values);
    $stg_signup->save();

    $spud->set_from_object($stg_signup);
}
```

You can also do the inverse and assign values to object variables that match the names of SPUD values. For example:

```
$default_stg_signup = new stg_signup();
$spud->assign_to_object(&$default_stg_signup);
```

Standard SPUD Fields

The section above mentions "standard" fields. Here is a list of the fields SPUD knows about and the amount of time they will be remembered:

firstname 2 days
lastname 2 days
addr1 2 hours
addr2 2 hours
city 365 days
state_cd 365 days
zip 365 days
country 365 days
phone 2 hours
email 2 days
userid 30 days
password 30 days
lastlogin 30 days
source 180 days
subsource 180 days

Using SPUD for Custom Fields

In addition to the standard fields mentioned above, SPUD can be used to store application/module-specific values as well. Retrieving custom fields is done with the `get_custom` object function. In addition to a field name, this function also takes an `$app`parameter (in this example, "blog"). You should make sure that your chosen `$app` is not in use by any other applications/modules.

```
$comment_style = $spud->get_custom('blog', 'comment_style');
```

Setting a value is done with the `set_custom` object function. To set a custom value, you must specify a time-to-live in seconds. This is the amount of time SPUD will remember this piece of information.

```
$spud->set_custom ('blog', 'comment_style', $_POST
['comment_style'], 2592000); /* 30 days */
```

Note that the various utility functions for setting/getting more than one value at a time are not available for custom fields.

Client-Side SPUD

Overview

In addition to calling SPUD from within a PHP script, there is a client-side SPUD library which allows you to call SPUD from within javascript. This library uses AJAX to communicate with the server to get/set SPUD values. This is useful for saving UI preferences and other preference information which may be needed by the client.

To use client side SPUD, first include the required javascript files:

```
<script src="/utils/ajax/ajax.class.js" type="text/
javascript"></script>
<script src="/utils/spud/spud.js" type="text/javascript"></
script>
```

- Note that the Control Panel automatically includes both of these files. You do not need to include these `script` tags on Control Panel pages.*

Once these files have been included, there will be four functions available in javascript: `spud_get` , `spud_set` , `spud_get_custom` , and

`spud_set_custom` . All functions take an object literal containing the necessary arguments.

spud_get

Arguments:

- `field/fields` : *(required)* The name of the field or fields to get. Can be a string or an array of strings. Each field must be a "standard" field from the list above.
- `callback` : *(required)* A reference to a javascript function to call once the request has completed. See below for details.
- `callback_param` : *(optional)* An arbitrary value that will be passed to the callback function. This can be used to pass a unique identifier when there is the possibility for multiple spud requests to overlap.
- Example:*
- ```
  <script type="text/javascript">
  ```
- ```
  function my_callback (value, myid) {
  ```
- ```
    alert ("First Name: " + value + "\nID: " + myid);
  ```
- ```
  }
  ```
- ```
  spud_get({field: "firstname", callback: my_callback, callback_param: "123456"});
  ```
- ```
  </script>
  ```

spud_set

Arguments:

- `field` : *(required)* The name of the field to set. This must be a "standard" field from the list above.
- `value` : *(required)* The value to set for the field.
- `callback` : *(optional)* A reference to a javascript function to call once the request has completed. See below for details.

- `callback_param` : *(optional)* An arbitrary value that will be passed to the callback function. This can be used to pass a unique identifier when there is the possibility for multiple spud requests to overlap.
- Example:*
- ```
  <script type="text/javascript">
  
  spud_set({field: "firstname", value: "Howard"});
  
  </script>
  ```

spud_get_custom

Arguments:

- `app` : *(required)* The app name to use. See the explanation of the PHP (server-side) get_custom function for details.
- `field` : *(required)* The name of the field to get.
- `callback` : *(required)* A reference to a javascript function to call once the request has completed. See below for details.
- `callback_param` : *(optional)* An arbitrary value that will be passed to the callback function. This can be used to pass a unique identifier when there is the possibility for multiple spud requests to overlap.
- Example:*
- ```
  <script type="text/javascript">
  
  function my_callback (value) {
  
    alert ("Favorite Color: " + value);
  
  }
  
  spud_get_custom({app: "colorpicker", field:
  "favoritecolor", callback: my_callback});
  
  </script>
  ```

spud_set_custom

Arguments:

- `app` : *(required)* The app name to use. See the explanation of the PHP (server-side) get_custom function for details.
- `field` : *(required)* The name of the field to set.
- `value` : *(required)* The value to set for the field.
- `ttl` : *(required)* The time-to-live setting (in seconds) for this piece of data. After this much time has elapsed, this value will be automatically removed from the user's SPUD.
- `callback` : *(optional)* A reference to a javascript function to call once the request has completed. See below for details.
- `callback_param` : *(optional)* An arbitrary value that will be passed to the callback function. This can be used to pass a unique identifier when there is the possibility for multiple spud requests to overlap.
- Example:*

```
<script type="text/javascript">

function my_callback(result) {

  if (result) {

    alert("Set Value!");

  } else {

    alert("Error Setting Value");

  }

}



spud_set_custom({app: "colorpicker", field:
"favoritecolor", value: "blue", ttl: 3600, callback:
my_callback});
```

-
- `</script>`
-

spud_set_from_url

This will search through the query string for the requested fields, and call spud_set with each one found.

- Arguments:*
- `field` : *(required)* The field or fields whose values you want to set, if present in the URL's query string. Can be a string or an array of strings.
- Example:*
-
-

```
spud_set_from_url('firstname' );
spud_set_from_url(['zip', 'email']);
```

-
-

spud_populate

This will populate a form by fetching each field value using spud_get and updating the value of the associated elements.

- Arguments:*
- `form_elements:` *(required)* A hash whose keys are field names, and whose values are either the elements or the element ID strings of the element to be populated.
- Example:*
-

-

```
spud_populate({
'zip': 'zip',
'firstname': firstnameElement
});
```

-
-

Defining a Callback

The **A** in **AJAX** stands for **asynchronous**. Because of this, client-side SPUD uses a callback to alert the client code when requested data is available or when an action has been completed. The callback function can take one or two arguments.

The first argument will be the result of the request. In the case of a `get` or `get_custom` call, this will be the value retrieved from the server. If an error occured, it will be the boolean `false` . For `set` and `set_custom` , the value will be either `true` or `false`depending on whether the set succeeded or failed.

The second argument will be the value of `callback_param` (if specified). If, for example, there are two form controls on a page, the `callback_param` could be set to the name of the form controls for which a value is being gotten or set.

Comprehensive Example

The example below shows how to use client-side SPUD to make a form that remembers its values between visits. If you change a value of a field, it will be saved in your SPUD. You can come back to the page for up to 1 day (86,400 seconds) and the last value you entered will still be shown.

```
<script src="/utils/ajax/ajax.class.js" type="text/
javascript"></script>
```

```
<script src="/utils/spud/spud.js" type="text/javascript"></
script>

<input type="text" id="field1" size="20" onblur="save_field
('data1', this.value);" />
<input type="text" id="field2" size="20" onblur="save_field
('data2', this.value);" />

<script type="text/javascript">
function save_field(name, value) {
     spud_set_custom({app: "myapp", field: name, value: value,
ttl: 86400});
}

// when we're loading, set the initial values of each field
spud_get_custom({app: "myapp", field: 'data1', callback:
set_field, callback_param: 'field1'});
spud_get_custom({app: "myapp", field: 'data2', callback:
set_field, callback_param: 'field2'});

function set_field (value, field_id) {
    if (value instanceof Boolean && value == false) {
        alert("could not load value for " + value);
    } else {
        document.getElementById(field_id).value = value;
    }
}
</script>
```

JSONP for SPUD

While the above AJAX calls are useful for client access on the same host, there are cross-domain XHR issues (even with sub-domains) which make the libraries unusable. One work-around is by using a web proxy, but in some cases (i.e., for those running their web front-end entirely on a CDN) this may not be possible.

The general solution for this is to provide the ability to return JSONP callbacks. It's a script-injection technique which provides a simple way to work with data from a trusted source.

Here's an example of how a third party would use this:

```
<script>
function callback(spud) {
  alert(spud.userid);
```

```
}

var script = document.createElement("script");
script.setAttribute("src", "http://my.barackobama.com/page/spud?jsonp=callback&type=get&field=userid");
script.setAttribute("type","text/javascript");
document.body.appendChild(script);
</script>
```

The differences from the AJAX support:

- JSONP calls are via GET not POST
- Calls made directly against AJAX resource not using spud.js script
- type 'get_custom' and 'set_custom' not (yet) supported
- multiple field gets allowed via 'getm'

Here is an example of how the 'getm' type is used to retrieve multiple fields:

http://my.barackobama.com/page/spud?jsonp=callback&type=getm&field=firstname,lastname,email,lastlogin

Configuring SPUD

The SPUD system has several [blue_config] important blue_config variables:

$ `enabled` : If set to 0, the SPUD system will be disabled site-wide. Code that calls SPUD will still function, but no values will ever be saved or returned.

$ `cookie_domain` : The domain for which SPUD cookies are saved. Should almost always be **.sitedomain.com. This value should be set to something reasonable by default, however, if the first access to a SPUD-enabled page was via a clientname.bluestatedigital.com URL, then the `cookie_domain` will be set to *.bluestatedigital.com. *You must change this before the site goes live on it's official URL.**

$ `cookie_expire` : The amount of time (in seconds) that a `spud` cookie containing a unique SPUD key should stick around in a user's browser. This should be at least as large as the largest possible "retention time" for any particular piece of data. It defaults to 2 years.

Using SPUD Data

Because SPUD data is collected in the background as a user interacts with a site, it's important to not put too much trust in it. Here are some guidelines:

1) SPUD data should never be saved into a database without a user first having the opportunity to edit it. For example, you can use SPUD to pre-fill the "zip code" section of a form, but you should never assume that the SPUD data is correct and save a zip code straight from a SPUD.

2) cons data always takes precedence over SPUD data.

3) Never depend on SPUD data being available. SPUD data may exist or it may not. Data that existed five minutes ago may have expired by the time you go looking for it again. Code should always fall back to reasonable behavior if the desired SPUD data is not available.

The Guts

SPUD works by assigning each user a unique cookie. This cookie maps to a row in the `app_spud` table in the database. In this table, a user's SPUD data is saved as a serialized string.

SPUD data is saved to the database by way of a destructor on the `spud` class. This is typically called when a script finished running or when no code is referencing the SPUD. Because of this, data is automatically saved and the developer using SPUD doesn't have to worry about calling a save function after making changes.

Although the maximum size of SPUD data is large (16 MB), it should be remembered that the entire SPUD row is loaded every time the user hits a SPUD-enabled page. As such, custom values should be kept small to prevent excessive load.