---+ Package ==

=head1 NAME

CGI::Session::Tutorial - Extended CGI::Session manual

=head1 STATE MAINTENANCE OVERVIEW

Since HTTP is a stateless protocol, each subsequent click to a web site is treated as new request by the Web server. The server does not relate a visit with a previous one, thus all the state information from the previous requests are lost. This makes creating such applications as shopping carts, web sites requiring users to authenticate, impossible. So people had to do something about this despair situation HTTP was putting us in.

For our rescue come such technologies as I and Is that help us save the users' session for a certain period. Since I and Is alone cannot take us too far (B), several other libraries have been developed to extend their capabilities and promise a more reliable solution. L<CGI::Session|CGI::Session> is one of them.

Before we discuss this library, let's look at some alternative solutions.

=head2 COOKIE

Cookie is a piece of text-information that a web server is entitled to place in the user's hard disk, assuming a user agent (such as Internet Explorer, Mozilla, etc) is compatible with the specification. After the cookie is placed, user agents are required to send these cookies back to the server as part of the HTTP request. This way the server application (CGI, for example) will have a way of relating previous requests by the same user agent, thus overcoming statelessness of HTTP.

Although I seem to be promising solution for the statelessness of HTTP, they do carry certain limitations, such as limited number of cookies per domain and per user agent and limited size on each cookie. User Agents are required to store at least 300 cookies at a time, 20 cookies per domain and allow 4096 bytes of storage for each cookie. They also rise several Privacy and Security concerns, the lists of which can be found on the sections B<6-"Privacy"> and B<7-"Security Considerations"> of B.

=head2 QUERY STRING

Query string is a string appended to URL following a question mark (?) such as:

http://my.dot.com/login.cgi?user=sherzodr;password=top-secret

As you probably guessed, it can also help you pass state information from a click to another, but how secure is it do you think, considering these URLs tend to get cached by most of the user agents and also logged in the servers access log, to which everyone can have access.

=head2 HIDDEN FIELDS

Hidden field is another alternative to using query strings and they come in two flavors: hidden fields used in POST methods and the ones in GET. The ones used in GET methods will turn into a true query strings once submitted, so all the disadvantages of QUERY_STRINGs apply. Although POST requests do not have limitations of its sister-GET, the pages that hold them get cached by Web browser, and are available within the source code of the page (obviously). They also become unwieldily to manage when one has oodles of state information to keep track of (for instance, a shopping cart or an advanced search engine).

Query strings and hidden fields are also lost easily by closing the browser, or by clicking the browser's "Back" button.

=head2 SERVER SIDE SESSION MANAGEMENT

This technique is built upon the aforementioned technologies plus a server-side storage device, which saves the state data on the server side. Each session has a unique id associated with the data in the server. This id is also associated with the user agent either in the form of a I, a I, hidden field or any combination of the above. This is necessary to make the connection with the client and his data.

Advantages:

=over 4

=item *

We no longer need to depend on User Agent constraints in cookie size.

=item *

Sensitive data no longer need to be traveling across the network at each request (which is the case with query strings, cookies and hidden fields). The only thing that travels is the unique id generated for the session (B, for instance), which should make no sense to third parties.

=item *

User will not have sensitive data stored in his/her computer in unsecured file (which is a cookie file).

=item *

It's possible to handle very big and even complex data structures transparently (which I do not handle).

=back

That's what CGI::Session is all about - implementing server side session management. Now is a good time to get feet wet.

=head1 PROGRAMMING STYLE

Server side session management system might be seeming awfully convoluted if you have never dealt with it. Fortunately, with L<CGI::Session|CGI::Session> all the complexity is handled by the library transparently. This section of the manual can be treated as an introductory tutorial to both logic behind session management, and to CGI::Session programming style.

All applications making use of server side session management rely on the following pattern of operation regardless of the way the system is implemented:

=over 4

=item 1

Check if the user has session cookie dropped in his computer from previous request

=item 2

If the cookie does not exist, create a new session identifier, and drop it as cookie to the user's computer.

=item 3

If session cookie exists, read the session ID from the cookie and load any previously saved session data from the server side storage. If session had any expiration date set it's useful to re-drop the same cookie to the user's computer so its expiration time will be reset to be relative to user's last activity time.

=item 4

Store any necessary data in the session that you want to make available for the next HTTP request.

=back

CGI::Session will handle all of the above steps. All you have to do is to choose what to store in the session.

=head2 GETTING STARTED

To make L<CGI::Session|CGI::Session>'s functionality available in your program do either of the following somewhere on top of your program file:

use CGI::Session; # or require CGI::Session;

Whenever you're ready to create a new session in your application, do the following:

$session = new CGI::Session() or die CGI::Session->errstr;

Above line will first try to re-initialize an existing session by consulting cookies and necessary QUERY_STRING parameters. If it fails will create a brand new session with a unique ID, which is normally called I, I for short, and can be accessed through L<id()|CGI::Session/id()> - object method.

We didn't check for any session cookies above, did we? No, we didn't, but CGI::Session did. It looked for a cookie called C, and if it found it tried to load existing session from server side storage (B in our case). If cookie didn't exist it looked for a QUERY_STRING parameter called C. If all the attempts to recover session ID failed, it created a new session.

NOTE: For the above syntax to work as intended your application needs to have write access to your computer's I folder, which is usually F in UNIX. If it doesn't, or if you wish to store this application's session files in a different place, you may pass the third argument like so:

$session = new CGI::Session(undef, undef, {Directory=>'../tmp/sessions'});

Now it will store all the newly created sessions in (and will attempt to initialize requested sessions from) that folder. Don't worry if the directory hierarchy you want to use doesn't already exist. It will be created for you. For details on how session data are stored refer to L<CGI::Session::Driver::file|CGI::Session::Driver::file>, which is the default driver used in our above example.

There is one small, but very important thing your application needs to perform after creating CGI::Session object as above. It needs to drop Session ID as an I into the user's computer. CGI::Session will use this cookie to identify the user at his/her next request and will be able to load his/her previously stored session data.

To make sure CGI::Session will be able to read your cookie at next request you need to consult its C<name()> method for cookie's suggested name:

$cookie = $query->cookie(-name => $session->name, -value => $session->id); print $query->header(-cookie=>$cookie);

C<name()> returns C by default. If you prefer a different cookie name, you can change it as easily too, but you have to do it before CGI::Session object is created:

CGI::Session->name("SID"); $session = new CGI::Session();

Baking the cookie wasn't too difficult, was it? But there is an even easier way to send a cookie using CGI::Session:

print $session->header();

The above will create the cookie using L<CGI::Cookie|CGI::Cookie> and will return proper http headers using L<CGI.pm|CGI>'s L<CGI|CGI/header()> method. Any arguments to L<CGI::Session|CGI::Session/header()> will be passed to L<CGI::header()|CGI/header()>.

Of course, this method of initialization will only work if client is accepting cookies. If not you would have to pass session ID in each URL of your application as QUERY_STRING. For CGI::Session to detect it the name of the parameter should be the same as returned by L<name()|CGI::Session/name()>:

printf ("click me", $session->name, $session->id);

If you already have session id to be initialized you may pass it as the only argument, or the second argument of multi-argument syntax:

$session = new CGI::Session($sid); $session = new CGI::Session("serializer:freezethaw", $sid); $session = new CGI::Session("driver:mysql", $sid, {Handle=>$dbh});

By default CGI::Session uses standard L<CGI|CGI> to parse queries and cookies. If you prefer to use a different, but compatible object you can pass that object in place of $sid:

$cgi = new CGI::Simple(); $session = new CGI::Session ($cgi); $session = new CGI::Session("driver:db_file;serializer:storable", $cgi); # etc

See L<CGI::Simple|CGI::Simple>

=head2 STORING DATA

L<CGI::Session|CGI::Session> offers L<param() method|CGI::Session/param()>, which behaves exactly as L<CGI.pm's param()|CGI/param()> with identical syntax. L<param()|CGI::Session/param()> is used for storing data in session as well as for accessing already stored data.

Imagine your customer submitted a login form on your Web site. You, as a good host, wanted to remember the guest's name, so you can a) greet him accordingly when he visits your site again, or b) to be helpful by filling out I part of his login form, so the customer can jump right to the I field without having to type his username again.

my $name = $cgi->param('username'); $session->param('username', $name);

Notice, we're grabbing I value of the field using CGI.pm's (or another compatible library's) C<param()> method, and storing it in session using L<CGI::Session|CGI::Session>'s L<param()|CGI::Session/param()> method.

If you have too many stuff to transfer into session, you may find yourself typing the above code over and over again. I've done it, and believe me, it gets very boring too soon, and is also error-prone. So we introduced the following handy method:

$session->save_param(['name']);

If you wanted to store multiple form fields just include them all in the second list:

$session->save_param(['name', 'email']);

If you want to store all the available I parameters you can omit the arguments:

$session->save_param();

See L<save_param()|CGI::Session/save_param> for more details.

When storing data in the session you're not limited to strings. You can store arrays, hashes and even most objects. You will need to pass them as references (except objects).

For example, to get all the selected values of a scrolling list and store it in the session:

my @fruits = $cgi->param('fruits'); $session->param('fruits', \@fruits);

For parameters with multiple values save_param() will do the right thing too. So the above is the same as:

$session->save_param($cgi, ['fruits']);

All the updates to the session data using above methods will not reflect in the data store until your application exits, or C<$session> goes out of scope. If, for some reason, you need to commit the changes to the data store before your application exits you need to call L<flush()|CGI::Session/flush()> method:

$session->flush();

I've written a lot of code, and never felt need for using C<flush()> method, since CGI::Session calls this method at the end of each request. There are, however, occasions I can think of one may need to call L<flush()|CGI::Session/flush()>.

=head2 ACCESSING STORED DATA

There's no point of storing data if you cannot access it. You can access stored session data by using the same L<param() method|CGI::Session/param()> you once used to store them. Remember the Username field from the previous section that we stored in the session? Let's read it back so we can partially fill the Login form for the user:

$name = $session->param("name"); printf "", $name;

To retrieve previously stored @fruits do not forget to de reference it:

@fruits = @{ $session->param('fruits') };

Very frequently, you may find yourself having to create pre-filled and pre-selected forms, like radio buttons, checkboxes and drop down menus according to the user's preferences or previous action. With text and textareas it's not a big deal - you can simply retrieve a single parameter from the session and hard code the value into the text field. But how would you do it when you have a group of radio buttons, checkboxes and scrolling lists? For this purpose, CGI::Session provides L<load_param()|CGI::Session/load_param()> method, which loads given session parameters to a CGI object (assuming they have been previously saved with L<save_param()|CGI::Session/save_param()> or alternative):

$session->load_param($cgi, ["fruits"]);

Now when you say:

print $cgi->checkbox_group(fruits=>['apple', 'banana', 'apricot']);

See L<load_param()|CGI::Session/load_param()> for details.

Generated checkboxes will be pre-filled using previously saved information. To see example of a real session-powered application consider http://handalak.com/cgi-bin/subscriptions.cgi

If you're making use of L<HTML::Template|HTML::Template> to separate the code from the skin, you can as well associate L<CGI::Session|CGI::Session> object with HTML::Template and access all the parameters from within HTML files. We love this trick!

$template = new HTML::Template(filename=>"some.tmpl", associate=>$session); print $template->output();

Assuming the session object stored "first_name" and "email" parameters while being associated with HTML::Template, you can access those values from within your "some.tmpl" file now:

Hello !

See L<HTML::Template's online manual|HTML::Template> for details.

=head2 CLEARING SESSION DATA

You store session data, you access session data and at some point you will want to clear certain session data, if not all. For this purpose L<CGI::Session|CGI::Session> provides L<clear()|CGI::Session/clear()> method which optionally takes one argument as an arrayref indicating which session parameters should be deleted from the session object:

$session->clear(["~logged-in", "email"]);

Above line deletes "~logged-in" and "email" session parameters from the session. And next time you say:

$email = $session->param("email");

it returns undef. If you omit the argument to L<clear()|CGI::Session/clear()>, be warned that all the session parameters you ever stored in the session object will get deleted. Note that it does not delete the session itself. Session stays open and accessible. It's just the parameters you stored in it gets deleted

See L<clear()|CGI::Session/clear()> for details.

=head2 DELETING A SESSION

If there's a start there's an end. If session could be created, it should be possible to delete it from the disk for good:

$session->delete();

The above call to L<delete()|CGI::Session/delete()> deletes the session from the disk for good. Do not confuse it with L<clear()|CGI::Session/clear()>, which only clears certain session parameters but keeps the session open.

See L<delete()|CGI::Session/delete()> for details.

=head2 EXPIRATION

L<CGI::Session|CGI::Session> provides limited means to expire sessions. Expiring a session is the same as deleting it via delete(), but deletion takes place automatically. To expire a session, you need to tell the library how long the session would be valid after the last access time. When that time is met, CGI::Session refuses to retrieve the session. It deletes the session and returns a brand new one. To assign expiration ticker for a session, use L<expire()|CGI::Session/expire()>:

$session->expire(3600); # expire after 3600 seconds $session->expire('+1h'); # expire after 1 hour $session->expire('+15m'); # expire after 15 minutes $session->expire('+1M'); # expire after a month and so on.

When session is set to expire at some time in the future, but session was not requested at or after that time has passed it will remain in the disk. When expired session is requested CGI::Session will remove the data from disk, and will initialize a brand new session.

See L<expire()|CGI::Session/expire()> for details.

Before CGI::Session 4.x there was no way of intercepting requests to expired sessions. CGI::Session 4.x introduced new kind of constructor, L<load()|CGI::Session/load()>, which is identical in use to L<new()|CGI::Session/new()>, but is not allowed to create sessions. It can only load them. If session is found to be expired, or session does not exist it will return an empty CGI::Session object. And if session is expired, in addition to being empty, its status will also be set to expired. You can check against these conditions using L<empty()|CGI::Session/empty()> and L<is_expired()|CGI::Session/is_expired()> methods. If session was loaded successfully object returned by C<load()> is as good a session as the one returned by C<new()>:

$session = CGI::Session->load() or die CGI::Session->errstr; if ($session->is_expired) { die "Your session expired. Please refresh your browser to re-start your session"; } if ($session->is_empty) { $session = $session->new(); }

Above example is worth an attention. Remember, all expired sessions are empty sessions, but not all empty sessions are expired sessions. Following this rule we have to check with C<is_expired()> before checking with C<is_empty()>. There is another thing about the above example. Notice how its creating new session when un existing session was requested? By calling C<new()> as an object method! Handy thing about that is, when you call C<new()> on a session object new object will be created using the same configuration as the previous object.

For example:

$session = CGI::Session->load("driver:mysql;serializer:storable", undef, {Handle=>$dbh}); if ($session->is_expired) { die "Your session is expired. Please refresh your browser to re-start your session"; } if ($session->is_empty) { $session = $session->new(); }

Initial C<$session> object was configured with B as the driver, B as the serializer and B<$dbh> as the database handle. Calling C< new() > on this object will return an object of the same configuration. So C< $session > object returned from C< new() > in the above example will use B as the driver, B as the serializer and B<$dbh> as the database handle.

See L<is_expired()|CGI::Session/is_expired()>, L<is_empty()|CGI::Session/is_empty()>, L<load()|CGI::Session/load()> for details.

Sometimes it makes perfect sense to expire a certain session parameter, instead of the whole session. I usually

do this in my login enabled sites, where after the user logs in successfully, I set his/her "_logged_in" session parameter to true, and assign an expiration ticker on that flag to something like 30 minutes. It means, after 30 idle minutes CGI::Session will L<clear|CGI::Session/clear()> "_logged_in" flag, indicating the user should log in over again. I agree, the same effect can be achieved by simply expiring() the session itself, but by doing this we would loose other session parameters, such as user's shopping cart, session-preferences and the like.

This feature can also be used to simulate layered authentication, such as, you can keep the user's access to his/her personal profile information for as long as 60 minutes after a successful login, but expire his/her access to his credit card information after 5 idle minutes. To achieve this effect, we will use L<expire()|CGI::Session/expire()> method again:

$session->expire(_profile_access, '1h'); $session->expire(_cc_access, '5m');

With the above syntax, the person will still have access to his personal information even after 5 idle hours. But when he tries to access or update his/her credit card information, he may be displayed a "login again, please" screen.

See L<expire()|CGI::Session/expire()> for details.

This concludes our discussion of CGI::Session programming style. The rest of the manual covers some L<"SECURITY"> issues. Driver specs from the previous manual were moved to L<CGI::Session::Driver|CGI::Session::Driver>.

=head1 SECURITY

"How secure is using CGI::Session?", "Can others hack down people's sessions using another browser if they can get the session id of the user?", "Are the session ids easy to guess?" are the questions I find myself answering over and over again.

=head2 STORAGE

Security of the library does in many aspects depend on the implementation. After making use of this library, you no longer have to send all the information to the user's cookie except for the session id. But, you still have to store the data in the server side. So another set of questions arise, can an evil person get access to session data in your server, even if he does, can he make sense out of the data in the session file, and even if he can, can he reuse the information against a person who created that session. As you see, the answer depends on yourself who is implementing it.

=over 4

=item *

First rule of thumb, do not store users' passwords or other sensitive data in the session, please. If you have to, use one-way encryption, such as md5, or SHA-1-1. For my own experience I can assure you that in properly implemented session-powered Web applications there is never a need for it.

=item *

Default configuration of the driver makes use of L<Data::Dumper|Data::Dumper> class to serialize data to make it possible to save it in the disk. Data::Dumper's result is a human readable data structure, which, if opened, can be interpreted easily. If you configure your session object to use either L<Storable|CGI::Session::Serialize::storable> or L<FreezeThaw|CGI::Session::Serialize::freezethaw> as a serializer, this would make it more difficult for bad guys to make sense out of session data. But don't use this as the only precaution. Since evil fingers can type a quick program using L<Storable|Storable> or

L<FreezeThaw|FreezeThaw> to decipher session files very easily.

=item *

Do not allow anyone to update contents of session files. If you're using L serialized data string needs to be eval()ed to bring the original data structure back to life. Of course, we use L<Safe|Safe> to do it safely, but your cautiousness does no harm either.

=item *

Do not keep sessions open for very long. This will increase the possibility that some bad guy may have someone's valid session id at a given time (acquired somehow). To do this use L<expire()|CGI::Session/expire()> method to set expiration ticker. The more sensitive the information on your Web site is, the sooner the session should be set to expire.

=back

=head2 SESSION IDs

Session ids are not easily guessed (unless you're using L)! Default configuration of CGI::Session uses L<Digest::MD5|CGI::Session::ID::md5> to generate random, 32 character long identifier. Although this string cannot be guessed as easily by others, if they find it out somehow, can they use this identifier against the other person?

Consider the scenario, where you just give someone either via email or an instant messaging a link to a Web site where you're currently logged in. The URL you give to that person contains a session id as part of a query string. If the site was initializing the session solely using query string parameter, after clicking on that link that person now appears to that site as you, and might have access to all of your private data instantly.

Even if you're solely using cookies as the session id transporters, it's not that difficult to plant a cookie in the cookie file with the same id and trick the web browser to send that particular session id to the server. So key for security is to check if the person who's asking us to retrieve a session data is indeed the person who initially created the session data.

One way to help with this is by also checking that the IP address that the session is being used from is always same. However, this turns out not to be practical in common cases because some large ISPs (such as AOL) use proxies which cause each and every request from the same user to come from different IP address.

If you have an application where you are sure your users' IPs are constant during a session, you can consider enabling an option to make this check:

use CGI::Session ('-ip_match');

For backwards compatibility, you can also achieve this by setting $CGI::Session::IP_MATCH to a true value. This makes sure that before initializing a previously stored session, it checks if the ip address stored in the session matches the ip address of the user asking for that session. In which case the library returns the session, otherwise it dies with a proper error message.

=head1 LICENSING

For support and licensing see L<CGI::Session|CGI::Session>

Note: Please contribute updates to this topic on TWiki.org at TWiki:TWiki.CGISessionTutorialDotPm.