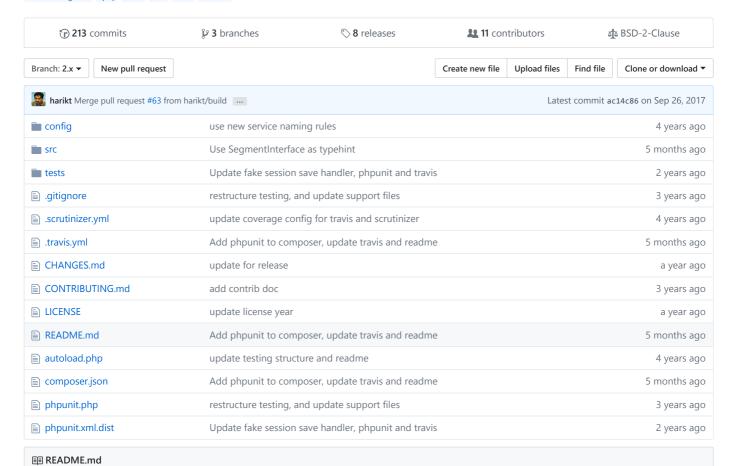
auraphp / Aura.Session

Tools for managing sessions, including session segments and read-once messages

#session-segment #php #flash #csrf #aura #session



Aura Session

Provides session management functionality, including lazy session starting, session segments, next-request-only ("flash") values, and CSRF tools.

Foreword

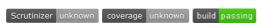
Installation

This library requires PHP 5.3 or later; we recommend using the latest available version of PHP as a matter of principle. It has no userland dependencies.

It is installable and autoloadable via Composer as aura/session.

Alternatively, download a release or clone this repository, then require or include its autoload.php file.

Quality



To run the unit tests at the command line, issue composer install and then vendor/bin/phpunit at the package root. This requires Composer to be available as composer.

This library attempts to comply with PSR-1, PSR-2, and PSR-4. If you notice compliance oversights, please send a patch via pull request.

Community

To ask questions, provide feedback, or otherwise communicate with the Aura community, please join our Google Group, follow @auraphp on Twitter, or chat with us on #auraphp on Freenode.

Getting Started

Instantiation

The easiest way to get started is to use the SessionFactory to create a Session manager object.

```
<?php
$session_factory = new \Aura\Session\SessionFactory;
$session = $session_factory->newInstance($_COOKIE);
?>
```

We can then use the *Session* instance to create *Segment* objects to manage session values and flashes. (In general, we should not need to manipulate the *Session* manager directly -- we will work mostly with *Segment* objects.)

Segments

In normal PHP, we keep session values in the \$_SESSION array. However, when different libraries and projects try to modify the same keys, the resulting conflicts can result in unexpected behavior. To resolve this, we use *Segment* objects. Each *Segment* addresses a named key within the \$_SESSION array for deconfliction purposes.

For example, if we get a Segment for Vendor\Package\ClassName, that Segment will contain a reference to \$_SESSION['Vendor\Package\ClassName']. We can then set() and get() values on the Segment, and the values will reside in an array under that reference.

```
<?php
// get a _Segment_ object
$segment = $session->getSegment('Vendor\Package\ClassName');
// try to get a value from the segment;
// if it does not exist, return an alternative value
echo $segment->get('foo'); // null
echo $segment->get('baz', 'not set'); // 'not set'
// set some values on the segment
$segment->set('foo', 'bar');
$segment->set('baz', 'dib');
// the $ SESSION array is now:
// $_SESSION = array(
// 'Vendor\Package\ClassName' => array(
       'tou -.
'baz' => 'dib',
           'foo' => 'bar',
//
// ),
// );
// try again to get a value from the segment
echo $segment->get('foo'); // 'bar'
// because the segment is a reference to $_SESSION, we can modify
// the superglobal directly and the segment values will also change
$_SESSION['Vendor\Package\ClassName']['zim'] = 'gir'
echo $segment->get('zim'); // 'gir'
?>
```

The benefit of a session segment is that we can deconflict the keys in the \$_SESSION superglobal by using class names (or some other unique name) for the segment names. With segments, different packages can use the \$_SESSION superglobal without stepping on each other's toes.

To clear all the values on a Segment, use the clear() method.

Flash Values

Segment values persist until the session is cleared or destroyed. However, sometimes it is useful to set a value that propagates only through the next request, and is then discarded. These are called "flash" values.

Setting And Getting Flash Values

To set a flash value on a Segment, use the setFlash() method.

```
<?php
$segment = $session->getSegment('Vendor\Package\ClassName');
$segment->setFlash('message', 'Hello world!');
?>
```

Then, in subsequent requests, we can read the flash value using getFlash():

```
<?php
$segment = $session->getSegment('Vendor\Package\ClassName');
$message = $segment->getFlash('message'); // 'Hello world!'
?>
```

N.b. As with <code>get()</code>, we can provide an alternative value if the flash key does not exist. For example, <code>getFlash('foo', 'not set')</code> will return 'not set' if there is no 'foo' key available.

Using setFlash() makes the flash value available only in the *next* request, not the current one. To make the flash value available immediately as well as in the next request, use setFlashNow(\$key, \$val).

Using getFlash() returns only the values that are available now from having been set in the previous request. To read a value that will be available in the next request, use getFlashNext(\$key, \$alt).

Keeping and Clearing Flash Values

Sometimes we will want to keep the flash values in the current request for the next request. We can do so on a per-segment basis by calling the *Segment* keepFlash() method, or we can keep all flashes for all segments by calling the *Session* keepFlash() method.

Similarly, we can clear flash values on a per-segment basis or a session-wide bases. Use the clearFlash() method on the Segment to clear flashes just for that segment, or the same method on the Session to clear all flash values for all segments.

Lazy Session Starting

Merely instantiating the *Session* manager and getting a *Segment* from it does *not* call session_start() . Instead, session_start() occurs only in certain circumstances:

- If we read from a Segment (e.g. with get()) the Session looks to see if a session cookie has already been set. If so, it will call session_start() to resume the previously-started session. If not, it knows there are no previously existing \$_SESSION values, so it will not call session_start().
- If we write to a Segment (e.g. with set()) the Session will always call session_start(). This will resume a previous session if it exists, or start a new one if it does not.

This means we can create each *Segment* at will, and session_start() will not be invoked until we actually interact with a *Segment* in a particular way. This helps to conserve the resources involved in starting a session.

Of course, we can force a session start or reactivation by calling the Session start() method, but that defeats the purpose of lazy-loaded sessions.

Saving, Clearing, and Destroying Sessions

N.b.: These methods apply to all session data and flashes across all segments.

To save the session data and end its use during the current request, call the commit() method on the Session manager:

```
<?php
$session->commit();
?>
```

N.b.: Per http://php.net/manual/en/session.examples.basic.php, "Sessions normally shutdown automatically when PHP is finished executing a script, but can be manually shutdown using the session_write_close() function." The commit() method is the equivalent of session_write_close().

To clear all session data, but leave the session active during the current request, use the clear() method on the Session manager.

```
<?php
$session->clear();
?>
```

To clear all flash values on a segment, use the clearFlash() method:

To clear the data *and* terminate the session for this and future requests, thereby destroying it completely, call the destroy() method:

```
<?php
$session->destroy(); // equivalent of session_destroy()
?>
```

Calling destroy() will also delete the session cookie via setcookie(). If we have an alternative means by which we delete cookies, we should pass a callable as the second argument to the SessionFactory method newInstance(). The callable should take three parameters: the cookie name, path, and domain.

Session Security

Session ID Regeneration

Any time a user has a change in privilege (that is, gaining or losing access rights within a system) be sure to regenerate the session ID:

```
<?php
$session->regenerateId();
?>
```

N.b.: The regenerateId() method also regenerates the CSRF token value.

Cross-Site Request Forgery

A "cross-site request forgery" is a security issue where the attacker, via malicious JavaScript or other means, issues a request in-the-blind from a client browser to a server where the user has already authenticated. The request *looks* valid to the server, but in fact is a forgery, since the user did not actually make the request (the malicious JavaScript did).

http://en.wikipedia.org/wiki/Cross-site_request_forgery

Defending Against CSRF

To defend against CSRF attacks, server-side logic should:

- 1. Place a token value unique to each authenticated user session in each form; and
- $\hbox{2. Check that all incoming POST/PUT/DELETE (i.e., "unsafe") requests contain that value.}\\$

N.b.: If our application uses GET requests to modify resources (which incidentally is an improper use of GET), we should also check for CSRF on GET requests from authenticated users.

For this example, the form field name will be __csrf_value . In each form we want to protect against CSRF, we use the session CSRF token value for that field:

When processing the request, check to see if the incoming CSRF token is valid for the authenticated user:

```
<?php
* @var Vendor\Package\User $user A user-authentication object.
* \underline{\text{@}}var Aura\Session\Session $session A session management object.
$unsafe = $_SERVER['REQUEST_METHOD'] == 'POST'
       | $_SERVER['REQUEST_METHOD'] == 'PUT'
       || $_SERVER['REQUEST_METHOD'] == 'DELETE';
if ($unsafe && $user->auth->isValid()) {
   $csrf_value = $_POST['__csrf_value'];
    $csrf_token = $session->getCsrfToken();
    if (! $csrf_token->isValid($csrf_value)) {
        echo "This looks like a cross-site request forgery.";
    } else {
       echo "This looks like a valid request.";
    }
} else {
    echo "CSRF attacks only affect unsafe requests by authenticated users.";
}
?>
```

CSRF Value Generation

For a CSRF token to be useful, its random value must be cryptographically secure. Using things like <code>mt_rand()</code> is insufficient. Aura. Session comes with a <code>Randval</code> class that implements a <code>RandvalInterface</code>. It uses the <code>random_bytes()</code> function preferentially, then <code>openssl</code>, or finally <code>mcrypt</code> to generate a random value. If you do not have one of these installed, you will need your own random-value implementation of the <code>RandvalInterface</code>. We suggest a wrapper around <code>RandomLib</code>.

Session Lifetime

We can set the session lifetime to as long (or as short) as we like using the setCookieParams on Session object. The lifetime is in seconds. To set the session cookie lifetime to two weeks:

```
<?php
$session->setCookieParams(array('lifetime' => '1209600'));
?>
```

N.b: The setCookieParams method calls session_set_cookie_params internally.