

Provides a unified interface to local and remote authentication systems.

[#aura](#) [#authentication](#) [#authenticator](#) [#login](#) [#oauth2](#) [#pdo](#) [#standalone-php-library](#)

321 commits

2 branches

4 releases

17 contributors

BSD-2-Clause

Branch: 2.x ▾


New pull request

Create new file

Upload files

Find file

Clone or download ▾

 pmjones	Merge pull request #93 from auraphp/harikt-patch-2 ...	Latest commit 99c16a1 on Sep 12, 2017
config	Use more meaningful and useful values.	8 months ago
src	Improv Exception message string.	5 months ago
tests	* Added phpunit to require-dev of composer	11 months ago
.gitignore	update testing and autoloading	2 years ago
.scrutinizer.yml	update coverage config for travis and scrutinizer	4 years ago
.travis.yml	Travis upload coverage report only for PHP 5.6	5 months ago
CHANGELOG.md	Removed changes.md, added changelog.md	11 months ago
CONTRIBUTING.md	add contrib file	2 years ago
LICENSE	updated license year and composer.json as release script	11 months ago
README.md	* Added phpunit to require-dev of composer	11 months ago
TODO.md	update todo	4 years ago
autoload.php	update autoloader	2 years ago
composer.json	updated license year and composer.json as release script	11 months ago
phpunit.php	update testing and autoloading	2 years ago
phpunit.xml.dist	Fix Travis / Phpunit errors.	5 months ago

README.md

Aura.Auth

Provides authentication functionality and session tracking using various adapters; currently supported adapters are:

- Apache httpasswd files
- SQL tables via the [PDO](#) extension
- IMAP/POP/NNTP via the [imap](#) extension
- LDAP and Active Directory via the [ldap](#) extension
- OAuth via customized adapters

Note that the purpose of this package is only to authenticate user credentials. It does not currently, and probably will not in the future, handle user account creation and management. That is more properly the domain of application-level functionality, or at least a separate Aura bundle.

Foreword

Installation

This library requires PHP 5.5 or later, and has no userland dependencies. (As a special consideration, this library is compatible with PHP 5.3 and 5.4 when the [ircmaxell/password-compat](#) package is installed.)

It is installable and autoloadable via Composer as [aura/auth](#).

Alternatively, [download a release](#) or clone this repository, then require or include its *autoload.php* file.

Quality

Scrutinizer unknown coverage unknown build **passing**

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

To track authentication state and related information, create an *Auth* object using the *AuthFactory*.

```
<?php
$auth_factory = new \Aura\Auth\AuthFactory($_COOKIE);
$auth = $auth_factory->newInstance();
?>
```

You can retrieve authentication information using the following methods on the *Auth* instance:

- `getUserName()` : returns the authenticated username string
- `getUserData()` : returns the array of optional arbitrary user data
- `getFirstActive()` : returns the Unix time of first activity (login)
- `getLastActive()` : return the Unix time of most-recent activity (generally that of the current request)
- `getStatus()` : returns the current authentication status constant. These constants are:
 - `Status::ANON` -- anonymous/unauthenticated
 - `Status::IDLE` -- the authenticated session has been idle for too long
 - `Status::EXPIRED` -- the authenticated session has lasted for too long in total
 - `Status::VALID` -- authenticated and valid
- `isAnon()`, `isIdle()`, `isExpired()`, `isValid()` : these return true or false, based on the current authentication status.

You can also use the `set*()` variations of the `get*()` methods above to force the *Auth* object to whatever values you like. However, because the values are stored in a `$_SESSION` segment, the values will not be retained if a session is not running.

To retain values in a session, you can start a session by force with `session_start()` on your own. Alternatively, it would be better to use one of the Aura.Auth package services to handle authentication and session-state management for you.

Services

This package comes with three services for dealing with authentication phases:

- *LoginService* to log in and start (or resume) a session,
- *LogoutService* to log out and remove the username and user data in the session (note that this **does not** destroy the session), and
- *ResumeService* to resume a previously-started session.

You can create each by using the *AuthFactory*. For now, we will look at how to force login and logout; later, we will show how to have the service use a credential adapter.

Forcing Login

You can force the *Auth* object to a logged-in state by calling the *LoginService* `forceLogin()` method with a user name and optional arbitrary user data.

```
<?php
// the authentication status is currently anonymous
echo $auth->getStatus(); // ANON

// create the login service
$login_service = $auth_factory->newLoginService();

// use the service to force $auth to a logged-in state
$username = 'boshag';
$userdata = array(
    'first_name' => 'Bolivar',
    'last_name' => 'Shagnasty',
    'email' => 'boshag@example.com',
);
$login_service->forceLogin($auth, $username, $userdata);

// now the authentication status is valid
echo $auth->getStatus(); // VALID
?>
```

Using `forceLogin()` has these side effects:

- it starts a new session if one has not already been started, or resumes a previous session if one exists
- it regenerates the session ID

The specified user name and user data will be stored in a `$_SESSION` segment, along with an authentication status of `Status::VALID`.

Note that `forceLogin()` does not check any credential sources. You as the application owner are forcing the *Auth* object to a logged-in state.

Forcing Logout

You can force the *Auth* object to a logged-out state by calling the *LogoutService* `forceLogout()` method.

```
<?php
// the authentication status is currently valid
echo $auth->getStatus(); // VALID

// create the logout service
$logout_service = $auth_factory->newLogoutService();

// use the service to force $auth to a logged-out state
$logout_service->forceLogout($auth);

// now the authentication status is anonymous/invalid
echo $auth->getStatus(); // ANON
?>
```

Using `forceLogout()` has these side effects:

- it clears any existing username and user data from the `$_SESSION` segment
- it regenerates the session ID

Note that `forceLogout()` does not check any credential sources. You as the application owner are forcing the *Auth* object to a logged-out state.

Note also that this **does not** destroy the session. This is because you may have other things you need in the session memory, such as flash messages.

Resuming A Session

When a PHP request ends, PHP saves the `$_SESSION` data for you. However, on the next request, PHP does not automatically start a new session for you, so `$_SESSION` is not automatically available.

You could start a new session yourself to repopulate `$_SESSION`, but that will incur a performance overhead if you don't actually need the session data. Similarly, there may be no need to start a session when there was no session previously (and thus no data to repopulate into `$_SESSION`). What we need is a way to start a session if one was started previously, but avoid starting a session if none was started previously.

The *ResumeService* exists to address this problem. When you call the `resume()` method on the *ResumeService*, it examines `$_COOKIE` to see if a session cookie is present:

- If the cookie is not present, it will not start a session, and return to the calling code. This avoids starting a session when there is no `$_SESSION` data to be populated.
- If the cookie is present, the *ResumeService* will start a session, thereby repopulating `$_SESSION`. Then it will update the authentication status depending on how long the session has been in place:
 - If the session has been idle for too long (i.e., too much time has passed since the last request), the *ResumeService* will log the user out automatically and return to the calling code.
 - If the session session has expired (i.e., the total logged-in time has been too long), the *ResumeService* will likewise log the user out automatically and return to the calling code.
 - Otherwise, the *ResumeService* will update the last-active time on the *Auth* object and return to the calling code.

Generally, you will want to invoke the *ResumeService* at the beginning of your application cycle, so that the session data becomes available at the earliest opportunity.

```
<?php
// create the resume service
$resume_service = $auth_factory->newResumeService();

// use the service to resume any previously-existing session
$resume_service->resume($auth);

// $_SESSION has now been repopulated, if a session was started previously,
// meaning the $auth object is now populated with its previous values, if any
?>
```

Adapters

Forcing the *Auth* object to a particular state is fine for when you want to exercise manual control over the authentication status, user name, user data, and other information. However, it is more often the case that you will want to check user credential input (username and password) against a credential store. This is where the *Adapter* classes come in.

To use an *Adapter* with a *Service*, you first need to create the *Adapter*, then pass it to the *AuthFactory* `new*Service()` method.

Htpasswd Adapter

Instantiation

To create an adapter for Apache htpasswd files, call the *AuthFactory* `newHtpasswdAdapter()` method and pass the file path of the Apache htpasswd file.

```
<?php
$htpasswd_adapter = $auth_factory->newHtpasswdAdapter(
    '/path/to/accounts.htpasswd'
);
?>
```

This will automatically use the *HtpasswdVerifier* to check DES, MD5, and SHA passwords from the htpasswd file on a per-user basis.

Service Integration

You can then pass the *Adapter* to each *Service* factory method like so:

```
<?php
$login_service = $auth_factory->newLoginService($htpasswd_adapter);
$logout_service = $auth_factory->newLogoutService($htpasswd_adapter);
```

```
$resume_service = $auth_factory->newResumeService($htpasswd_adapter);
?>
```

To attempt a user login, pass an array with `username` and `password` elements to the *LoginService* `login()` method along with the *Auth* object:

```
<?php
$login_service->login($auth, array(
    'username' => 'boshag',
    'password' => '12345'
));
?>
```

For more on *LoginService* idioms, please see the [Service Idioms](#) section. (The *LogoutService* and *ResumeService* do not need credential information.)

IMAP/POP/NNTP Adapter

Instantiation

To create an adapter for IMAP/POP/NNTP servers, call the *AuthFactory* `newImapAdapter()` method and pass the mailbox specification string, along with any appropriate option constants:

```
<?php
$imap_adapter = $auth_factory->newImapAdapter(
    '{mail.example.com:143/imap/secure}',
    OP_HALFOPEN
);
?>
```

N.b.: See the [imap_open\(\)](#) documentation for more variations on mailbox specification strings.

Service Integration

You can then pass the *Adapter* to each *Service* factory method like so:

```
<?php
$login_service = $auth_factory->newLoginService($imap_adapter);
$logout_service = $auth_factory->newLogoutService($imap_adapter);
$resume_service = $auth_factory->newResumeService($imap_adapter);
?>
```

To attempt a user login, pass an array with `username` and `password` elements to the *LoginService* `login()` method along with the *Auth* object:

```
<?php
$login_service->login($auth, array(
    'username' => 'boshag',
    'password' => '12345'
));
?>
```

For more on *LoginService* idioms, please see the [Service Idioms](#) section. (The *LogoutService* and *ResumeService* do not need credential information.)

LDAP Adapter

Instantiation

To create an adapter for LDAP and Active Directory servers, call the *AuthFactory* `newLdapAdapter()` method and pass the server name with a distinguished name (DN) format string:

```
<?php
$ldap_adapter = $auth_factory->newLdapAdapter(
    'ldaps://ldap.example.com:636',
    'ou=Company Name,dc=Department Name,cn=users,uid=%s'
);
```

```
);  
?>
```

N.b.: The username will be escaped and then passed to the DN format string via `sprintf()`. The completed DN will be used for binding to the server after connection.

Service Integration

You can then pass the *Adapter* to each *Service* factory method like so:

```
<?php  
$login_service = $auth_factory->newLoginService($ldap_adapter);  
$logout_service = $auth_factory->newLogoutService($ldap_adapter);  
$resume_service = $auth_factory->newResumeService($ldap_adapter);  
?>
```

To attempt a user login, pass an array with `username` and `password` elements to the *LoginService* `login()` method along with the *Auth* object:

```
<?php  
$login_service->login($auth, array(  
    'username' => 'boshag',  
    'password' => '12345'  
));  
?>
```

For more on *LoginService* idioms, please see the [Service Idioms](#) section. (The *LogoutService* and *ResumeService* do not need credential information.)

PDO Adapter

Instantiation

To create an adapter for PDO connections to SQL tables, call the *AuthFactory* `newPdoAdapter()` method and pass these parameters in order:

- a *PDO* connection instance
- a indication of how passwords are hashed in the database:
 - if a `PASSWORD_*` constant from PHP 5.5 and up, it is treated as `password_hash()` algorithm for a *PasswordVerifier* instance (this is the preferred method)
 - if a string, it is treated as a `hash()` algorithm for a *HashVerifier* instance
 - otherwise, it is expected to be an implementation of *VerifierInterface*
- an array of column names: the first element is the username column, the second element is the hashed-password column, and additional columns are used as extra user information to be selected and returned from the database
- a `FROM` specification string to indicate one or more table names, with any other `JOIN` clauses you wish to add
- an optional `WHERE` condition string; use this to add extra conditions to the `SELECT` statement built by the adapter

Here is a legacy example where passwords are MD5 hashed in an accounts table:

```
<?php  
$pdo = new \PDO(...);  
$hash = new PasswordVerifier('md5');  
$cols = ('username', 'md5password');  
$from = 'accounts';  
$pdo_adapter = $auth_factory->newPdoAdapter($pdo, $hash, $cols, $from);  
?>
```

Here is a modern, more complex example that uses bcrypt instead of md5, retrieves extra user information columns from joined tables, and filters for active accounts:

```

<?php
$pdo = new \PDO(...);
$hash = new PasswordVerifier(PASSWORD_BCRYPT);
$cols = array(
    'accounts.username', // "AS username" is added by the adapter
    'accounts.bcryptpass', // "AS password" is added by the adapter
    'accounts.uid AS uid',
    'userinfo.email AS email',
    'userinfo.uri AS website',
    'userinfo.fullname AS display_name',
);
$from = 'accounts JOIN profiles ON accounts.uid = profiles.uid';
$where = 'accounts.active = 1';
$pdo_adapter = $auth_factory->newPdoAdapter($pdo, $hash, $cols, $from, $where);
?>

```

(The additional information columns will be retained in the session data after successful authentication.)

Service Integration

You can then pass the *Adapter* to each *Service* factory method like so:

```

<?php
$login_service = $auth_factory->newLoginService($pdo_adapter);
$logout_service = $auth_factory->newLogoutService($pdo_adapter);
$resume_service = $auth_factory->newResumeService($pdo_adapter);
?>

```

To attempt a user login, pass an array with `username` and `password` elements to the *LoginService* `login()` method along with the *Auth* object:

```

<?php
$login_service->login($auth, array(
    'username' => 'boshag',
    'password' => '12345'
));
?>

```

For more on *LoginService* idioms, please see the [Service Idioms](#) section. (The *LogoutService* and *ResumeService* do not need credential information.)

Custom Adapters

Although this package comes with multiple *Adapter* classes, it may be that none of them fit your needs.

You may wish to extend one of the existing adapters to add login/logout/resume behaviors. Alternatively, you can create an *Adapter* of your own by implementing the *AdapterInterface* on a class of your choosing:

```

<?php
use Aura\Auth\Adapter\AdapterInterface;
use Aura\Auth\Auth;
use Aura\Auth\Status;

class CustomAdapter implements AdapterInterface
{
    // AdapterInterface::login()
    public function login(array $input)
    {
        if ($this->isLegit($input)) {
            $username = ...;
            $userdata = array(...);
            $this->updateLoginTime(time());
            return array($username, $userdata);
        } else {
            throw CustomException('Something went wrong.');
```

```

        $this->updateLogoutTime($auth->getUsername(), time());
    }

    // AdapterInterface::resume()
    public function resume(Auth $auth)
    {
        $this->updateActiveTime($auth->getUsername(), time());
    }

    // custom support methods not in the interface
    protected function isLegit($input) { ... }

    protected function updateLoginTime($time) { ... }

    protected function updateActiveTime($time) { ... }

    protected function updateLogoutTime($time) { ... }
}
?>

```

You can then pass an instance of the custom adapter when creating services through the *AuthFactory* methods:

```

<?php
$custom_adapter = new CustomAdapter;
$login_service = $auth_factory->newLoginService($custom_adapter);
$logout_service = $auth_factory->newLogoutService($custom_adapter);
$resume_service = $auth_factory->newResumeService($custom_adapter);
?>

```

OAuth Adapters

Should you desire to handle your authentication through a 3rd party service that uses OAuth 2.0, you'll need to write an adapter that implements `Aura\Auth\Adapter\AdapterInterface` and provide your own implementation for fetching the access token and user information. Your implementation can be something you've written yourself or it can be an existing OAuth2 client.

The following example will demonstrate how you'd go about creating this adapter using the [PHP League's OAuth2 Client](#). We'll also be using Github as the service provider for this example.

```

<?php
namespace OAuth2\Adapter;

use Aura\Auth\Adapter\AdapterInterface;
use Aura\Auth\Exception;
use Aura\Auth\Auth;
use Aura\Auth>Status;
use League\OAuth2\Client\Provider\AbstractProvider;

class LeagueOAuth2Adapter implements AdapterInterface
{
    /**
     * @var \League\OAuth2\Client\Provider\IdentityProvider
     * The identity provider that the adapter will use
     */
    protected $provider;

    public function __construct(AbstractProvider $provider)
    {
        $this->provider = $provider;
    }

    /**
     * @param $input an input containing the OAuth 2 code
     * @return array the username and details for the user
     * @throws \Aura\Auth\Exception
     * This method must be implemented to fulfill the contract
     * with AdapterInterface
     */
    public function login(array $input)
    {
        if (!isset($input['code'])) {
            throw new Exception('Authorization code missing.');
```



```

    }

    $token = $this->provider->getAccessToken(
        'authorization_code',
        array('code' => $input['code'])
    );

    $details = $this->provider->getResourceOwner($token);
    $data = [
        'name' => $details->getName(),
        'email' => $details->getEmail(),
    ];
    $data['token'] = $token;
    $username = $data['email'];
    return [$username, $data];
}

/**
 * @param Auth $auth
 * Logout method is required to fulfill the contract with AdapterInterface
 */
public function logout(Auth $auth, $status = Status::ANON)
{
    //nothing to do here
}

/**
 * @param Auth $auth
 * Resume method required to fulfill the contract with AdapterInterface
 */
public function resume(Auth $auth)
{
    // nothing to do here
}
}
?>

```

As you can see in the code, your adapter will be accepting a client as a parameter and using that client to fulfill the \Aura\Auth\Adapter\AdapterInterface contract. This adapter would be commonly used in an OAuth2 Callback process. Essentially, once you provide your credentials and authenticate with the 3rd Party service (in this case Github), you will be redirected back to a script on your server where you'll have to verify that you sent the request by sending an verification code back to the service. This is why it's a good idea to use a good OAuth2 client in lieu of writing your own. Below is an example of what this OAuth2 callback code might look like.

```

<?php
namespace OAuth2;

use Aura\Auth\AuthFactory;
use League\OAuth2\Client\Provider\Github;
use OAuth2\Adapter\LeagueOAuth2Adapter;
use Aura\Auth\Exception;

require_once 'vendor/autoload.php';

$auth_factory = new AuthFactory($_COOKIE);
$auth = $auth_factory->newInstance();

$github_provider = new Github(array(
    'clientId' => 'xxxxxxxxxxxxxxxx',
    'clientSecret' => 'xxxxxxxxxxxxxxxx',
    'redirectUri' => 'http://aura.auth.dev/'
));

if (!isset($_GET['code'])) {
    header('Location: ' . $github_provider->getAuthorizationUrl());
    exit;
} else {
    $oauth_adapter = new LeagueOAuth2Adapter($github_provider);
    $login_service = $auth_factory->newLoginService($oauth_adapter);
    try {
        // array is the username and an array of info and indicates successful
        // login
        $data = $login_service->login($auth, $_GET);
    } catch (Exception $e) {
        // handle the exception
    }
}

```

```

    }
}
?>

```

The fact that not every 3rd Party Service returns data the same way means it's not reasonable for Aura to try to handle every different data set out of the box. By writing this little bit of code, you can easily implement Aura Auth for your 3rd Party OAuth2 services.

Service Idioms

Resuming A Session

This is an example of the code needed to resume a pre-existing session. Note that the `echo` statements are intended to explain the different resulting states of the `resume()` call, and may be replaced by whatever logic you feel is appropriate. For example, you may wish to redirect to a login page when a session has idled or expired.

```

<?php
$auth = $auth_factory->newInstance();

$resume_service = $auth_factory->newResumeService(...);
$resume_service->resume($auth);

switch (true) {
    case $auth->isAnon():
        echo "You are not logged in.";
        break;
    case $auth->isIdle():
        echo "Your session was idle for too long. Please log in again.";
        break;
    case $auth->isExpired():
        echo "Your session has expired. Please log in again.";
        break;
    case $auth->isValid():
        echo "You are still logged in.";
        break;
    default:
        echo "You have an unknown status.";
        break;
}
?>

```

N.b.: Instead of creating the *Auth* and *ResumeService* objects by hand, you may wish to use a dependency injection container such as [Aura.Di](#) to retain them for shared use throughout your application.

Logging In

This is an example of the code needed to effect a login. Note that the `echo` and `$log` statements are intended to explain the different resulting states of the `login()` call, and may be replaced by whatever logic you feel is appropriate; in particular, you should probably not expose the exact nature of the failure, to help mitigate brute-force attempts.

```

<?php

class InvalidLoginException extends Exception {}

$auth = $auth_factory->newInstance();

$login_service = $auth_factory->newLoginService(...);

try {
    $login_service->login($auth, array(
        'username' => $_POST['username'],
        'password' => $_POST['password'],
    ));
    echo "You are now logged into a new session.";
} catch (\Aura\Auth\Exception\UsernameMissing $e) {

    $log->notice("The 'username' field is missing or empty.");
    throw new InvalidLoginException();
}

```

```

} catch (\Aura\Auth\Exception\PasswordMissing $e) {

    $log->notice("The 'password' field is missing or empty.");
    throw new InvalidLoginException();

} catch (\Aura\Auth\Exception\UsernameNotFound $e) {

    $log->warning("The username you entered was not found.");
    throw new InvalidLoginException();

} catch (\Aura\Auth\Exception\MultipleMatches $e) {

    $log->warning("There is more than one account with that username.");
    throw new InvalidLoginException();

} catch (\Aura\Auth\Exception\PasswordIncorrect $e) {

    $log->notice("The password you entered was incorrect.");
    throw new InvalidLoginException();

} catch (\Aura\Auth\Exception\ConnectionFailed $e) {

    $log->notice("Could not connect to IMAP or LDAP server.");
    $log->info("This could be because the username or password was wrong,");
    $log->info("or because the the connect operation itself failed in some way. ");
    $log->info($e->getMessage());
    throw new InvalidLoginException();

} catch (\Aura\Auth\Exception\BindFailed $e) {

    $log->notice("Could not bind to LDAP server.");
    $log->info("This could be because the username or password was wrong,");
    $log->info("or because the the bind operation itself failed in some way. ");
    $log->info($e->getMessage());
    throw new InvalidLoginException();

} catch (InvalidLoginException $e) {

    echo "Invalid login details. Please try again.";

}
?>

```

N.b.: Instead of creating the *Auth* and *LoginService* objects by hand, you may wish to use a dependency injection container such as [Aura.Di](#) to retain them for shared use throughout your application.

Alternatively, you may wish to use credentials from the HTTP `Authorization: Basic` headers instead of using `$_POST` or other form-related inputs. On Apache `mod_php` you might use the auto-populated `$_SERVER['PHP_AUTH_*']` values:

```

<?php
$authorization_basic = function () {
    return array(
        isset($_SERVER['PHP_AUTH_USER']) ? $_SERVER['PHP_AUTH_USER'] : null,
        isset($_SERVER['PHP_AUTH_PW']) ? $_SERVER['PHP_AUTH_PW'] : null,
    );
}

list($username, $password) = $authorization_basic();
$login_service->login($auth, array(
    'username' => $username,
    'password' => $password,
));
?>

```

On other servers you may need to extract the credentials from the `Authorization: Basic` header itself:

```

<?php
$authorization_basic = function () {

    $header = isset($_SERVER['HTTP_AUTHORIZATION'])
        ? $_SERVER['HTTP_AUTHORIZATION']
        : '';

    if (strtolower(substr($header, 0, 6)) !== 'basic ') {

```

```

        return array(null, null);
    }

    $encoded = substr($header, 6);
    $decoded = base64_decode($encoded);
    return explode(':', $decoded);
}

list($username, $password) = $authorization_basic();
$login_service->login($auth, array(
    'username' => $username,
    'password' => $password,
));
?>

```

Logging Out

This is an example of the code needed to effect a logout. Note that the `echo` statements are intended to explain the different resulting states of the `logout()` call, and may be replaced by whatever logic you feel is appropriate.

```

<?php
$auth = $auth_factory->newInstance();

$logout_service = $auth_factory->newLogoutService(...);

$logout_service->logout($auth);

if ($auth->isAnon()) {
    echo "You are now logged out.";
} else {
    echo "Something went wrong; you are still logged in.";
}
?>

```

N.b.: Instead of creating the *Auth* and *LogoutService* objects by hand, you may wish to use a dependency injection container such as [Aura.Di](#) to retain them for shared use throughout your application.

Custom Services

You are not restricted to the login, logout, and resume services provided by this package. However, if you build a service of your own, or if you extend one of the provided services, you will have to instantiate that customized service object manually, instead of using the *AuthFactory*. This can be tedious but is not difficult, especially when using a dependency injection container system such as [Aura.Di](#).

Session Management

The *Service* objects use a *Session* object to start sessions and regenerate session IDs. (Note that they **do not** destroy sessions.) The *Session* object uses the native PHP `session_*`() functions to manage sessions.

Custom Sessions

If you wish to use an alternative means of managing sessions, implement the *SessionInterface* on an object of your choice. One way to do this is by wrapping a framework-specific session object and proxying the *SessionInterface* methods to the wrapped object:

```

<?php
use Aura\Auth\Session\SessionInterface;

class CustomSession implements SessionInterface
{
    protected $fwsession;

    public function __construct(FrameworkSession $fwsession)
    {
        $this->fwsession = $fwsession;
    }

    public function start()
    {
        return $this->fwsession->startSession();
    }
}

```

```

    }

    public function resume()
    {
        if ($this->fwsession->isAlreadyStarted()) {
            return true;
        }

        if ($this->fwsession->canBeRestarted()) {
            return $this->fwsession->restartSession();
        }

        return false;
    }

    public function regenerateId()
    {
        return $this->fwsession->regenerateSessionId();
    }
}
?>

```

Then pass that custom session object to the *AuthFactory* instantiation:

```

<?php
use Aura\Auth\AuthFactory;

$custom_session = new CustomSession(new FrameworkSession);
$auth_factory = new AuthFactory($_COOKIE, $custom_session);
?>

```

The factory will pass your custom session object wherever it is needed.

Working Without Sessions

In some situations, such as with APIs where credentials are provided with every request, it may be beneficial to avoid sessions altogether. In this case, pass a *NullSession* and *NullSegment* to the *AuthFactory*:

```

<?php
use Aura\Auth\AuthFactory;
use Aura\Auth\Session\NullSession;
use Aura\Auth\Session\NullSegment;

>null_session = new NullSession;
>null_segment = new NullSegment;
$auth_factory = new AuthFactory($_COOKIE, $null_session, $null_segment);
?>

```

With the *NullSession*, no session will ever be started, and no session ID will be created or regenerated. Likewise, no session will ever be resumed, because it will never have been saved at the end of the previous request. Finally, PHP will never create a session cookie to send in the response.

Similarly, the *NullSegment* retains authentication information in an object property instead of in a `$_SESSION` segment. Unlike the normal *Segment*, which only retains data when `$_SESSION` is present, the *NullSegment* will always retain data that is set into it. When the request is over, all information retained in the *NullSegment* will disappear.

When using the *NullSession* and *NullSegment*, you will have to check credentials via the *LoginService* `login()` or `forceLogin()` method on each request, which in turn will retain the authentication information in the *Segment*. In an API situation this is often preferable to managing an ongoing session.

N.b. In an API situation, the credentials may be an API token, or passed as HTTP basic or digest authentication headers. Pass these to the adapter of your choice.

DI Configuration

Here are some hints regarding configuration of Aura.Auth via Aura.Di.

Aura\Auth\Adapter\HtpasswdAdapter

```
<?php
$di->params['Aura\Auth\Adapter\HtpasswdAdapter'] = array(
    'file' => '/path/to/htpasswdfile',
);
?>
```

Aura\Auth\Adapter\ImapAdapter

```
<?php
$di->params['Aura\Auth\Adapter\ImapAdapter'] = array(
    'mailbox' => '{mail.example.com:143/imap/secure}',
);
?>
```

Aura\Auth\Adapter\LdapAdapter

```
<?php
$di->params['Aura\Auth\Adapter\LdapAdapter'] = array(
    'server' => 'ldaps://ldap.example.com:636',
    'dnformat' => 'ou=Company Name,dc=Department Name,cn=users,uid=%s',
);
?>
```

Aura\Auth\Adapter\PdoAdapter

```
<?php
$di->params['Aura\Auth\Adapter\PdoAdapter'] = array(
    'pdo' => $di->lazyGet('your_pdo_connection_service'),
    'cols' => array(
        'username_column',
        'password_column',
    ),
    'from' => 'users_table',
    'where' => '',
);
?>
```