HTTP Session

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as Memcached, Redis, and databases is included out of the box.

Configuration

The session configuration file is stored at `config/session.php`. Be sure to review the options available to you in this file. By default, Laravel is configured to use the `file` session driver, which will work well for many applications. In production applications, you may consider using the `memcached` or `redis` drivers for even faster session performance.

The session `driver` configuration option defines where session data will be stored for each request. Laravel ships with several great drivers out of the box:

- `file` - sessions are stored in `storage/framework/sessions`.
- `cookie` - sessions are stored in secure, encrypted cookies.
- `database` - sessions are stored in a relational database.
- `memcached` / `redis` - sessions are stored in one of these fast, cache based stores.
- `array` - sessions are stored in a PHP array and will not be persisted.

The array driver is used during testing and prevents the data stored in the session from being persisted.

Driver Prerequisites

Database

When using the `database` session driver, you will need to create a table to contain the session items. Below is an example `Schema` declaration for the table:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

You may use the `session:table` Artisan command to generate this migration:

```
php artisan session:table

php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer. You may configure your Redis connections in the

`database` configuration file. In the `session` configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

Using The Session

Retrieving Data

There are two primary ways of working with session data in Laravel: the global `session` helper and via a `Request` instance. First, let's look at accessing the session via a `Request` instance, which can be type-hinted on a controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function show(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

When you retrieve an item from the session, you may also pass a default value as the second argument to the `get` method. This default value will be returned if the specified key does not exist in the session. If you pass a `Closure` as the default value to the `get` method and the requested key does not exist, the `Closure` will be executed and its result returned:

```php
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

The Global Session Helper

You may also use the global `session` PHP function to retrieve and store data in the session. When the `session` helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

```php
Route::get('home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

There is little practical difference between using the session via an HTTP request instance versus using the global `session` helper. Both methods are testable via the `assertSessionHas` method which is available in all of your test cases.

Retrieving All Session Data

If you would like to retrieve all the data in the session, you may use the `all` method:

```php
$data = $request->session()->all();
```

Determining If An Item Exists In The Session

To determine if an item is present in the session, you may use the `has` method. The `has` method returns `true` if the item is present and is not `null`:

```
if ($request->session()->has('users')) {
    //
}
```

To determine if an item is present in the session, even if its value is `null`, you may use the `exists` method. The `exists` method returns `true` if the item is present:

```
if ($request->session()->exists('users')) {
    //
}
```

Storing Data

To store data in the session, you will typically use the `put` method or the `session` helper:

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global helper...
session(['key' => 'value']);
```

Pushing To Array Session Values

The `push` method may be used to push a new value onto a session value that is an array. For example, if the `user.teams` key contains an array of team names, you may push a new value onto the array like so:

```
$request->session()->push('user.teams', 'developers');
```

Retrieving & Deleting An Item

The `pull` method will retrieve and delete an item from the session in a single statement:

```
$value = $request->session()->pull('key', 'default');
```

Flash Data

Sometimes you may wish to store items in the session only for the next request. You may do so using the `flash` method. Data stored in the session using this method will only be available during the subsequent HTTP request, and then will be deleted. Flash data is primarily useful for short-lived status messages:

```
$request->session()->flash('status', 'Task was successful!');
```

If you need to keep your flash data around for several requests, you may use the `reflash` method, which will keep all of the flash data for an additional request. If you only need to keep specific flash data, you may use the `keep` method:

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

Deleting Data

The `forget` method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the `flush` method:

```
// Forget a single key...
$request->session()->forget('key');

// Forget multiple keys...
$request->session()->forget(['key1', 'key2']);

$request->session()->flush();
```

Regenerating The Session ID

Regenerating the session ID is often done in order to prevent malicious users from exploiting a session fixation attack on your application.

Laravel automatically regenerates the session ID during authentication if you are using the built-in `LoginController`; however, if you need to manually regenerate the session ID, you may use the `regenerate` method.

```
$request->session()->regenerate();
```

Adding Custom Session Drivers

Implementing The Driver

Your custom session driver should implement the `SessionHandlerInterface`. This interface contains just a few simple methods we need to implement. A stubbed MongoDB implementation looks something like this:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

Laravel does not ship with a directory to contain your extensions. You are free to place them anywhere you like. In this example, we have created an `Extensions` directory to house the `MongoSessionHandler`.

Since the purpose of these methods is not readily understandable, let's quickly cover what each of the methods do:

- The `open` method would typically be used in file based session store systems. Since Laravel ships with a `file` session driver, you will almost never need to put anything in this method. You can leave it as an empty stub. It is a fact of poor interface design (which we'll discuss later) that PHP requires us to implement this method.
- The `close` method, like the `open` method, can also usually be disregarded. For most drivers, it is not needed.
- The `read` method should return the string version of the session data associated with the given `$sessionId`. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
- The `write` method should write the given `$data` string associated with the `$sessionId` to some persistent storage system, such as MongoDB, Dynamo, etc. Again, you should not perform any serialization - Laravel will have already handled that for you.
- The `destroy` method should remove the data associated with the `$sessionId` from persistent storage.
- The `gc` method should destroy all session data that is older than the given `$lifetime`, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

Registering The Driver

Once your driver has been implemented, you are ready to register it with the framework. To add additional drivers to Laravel's session backend, you may use the `extend` method on the `Session` facade. You should call the `extend` method from the `boot` method of a service provider. You may do this from the existing `AppServiceProvider` or create an entirely new provider:

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
```

```
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionHandler;
        });
    }
}
```

Once the session driver has been registered, you may use the `mongo` driver in your `config/session.php` configuration file.

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.