# Session



## # Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across requests. Laravel ships with a variety of session back-ends available for use through a clean, unified API. Support for popular back-ends 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 well documented 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` 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 database used by your application.
- `memcached` / `redis` - sessions are stored in one of these fast, cache based stores.
- `array` - sessions are stored in a simple PHP array and will not be persisted across requests.

**Note:** The array driver is typically used for running tests to prevent session data from persisting.

### Driver Prerequisites

**Database**

When using the `database` session driver, you will need to setup 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->text('payload');
    $table->integer('last_activity');
});
```

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

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

#### Redis

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

## Other Session Considerations

The Laravel framework uses the `flash` session key internally, so you should not add an item to the session by that name.

If you need all stored session data to be encrypted, set the `encrypt` configuration option to `true`.

# # Basic Usage

#### Accessing The Session

First, let's access the session. We can access the session instance via the HTTP request, which can be type-hinted on a controller method. Remember, controller method dependencies are injected via the Laravel service container:

```php
<?php

namespace App\Http\Controllers;

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

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

        //
    }
}
```

When you retrieve a value 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, the `Closure` will be executed and its result returned:

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

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

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

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

You may also use the global `session` PHP function to retrieve and store data in the session:

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

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

### Determining If An Item Exists In The Session

The `has` method may be used to check if an item exists in the session. This method will return `true` if the item exists:

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

#### Storing Data In The Session

Once you have access to the session instance, you may call a variety of functions to interact with the underlying data. For example, the `put` method stores a new piece of data in the session:

```
$request->session()->put('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 And Deleting An Item

The `pull` method will retrieve and delete an item from the session:

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

#### Deleting Items From The Session

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:

```
$request->session()->forget('key');

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

#### Regenerating The Session ID

If you need to regenerate the session ID, you may use the `regenerate` method:

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

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

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

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

# Adding Custom Session Drivers

To add additional drivers to Laravel's session back-end, you may use the `extend` method on the `Session` facade. You can call the `extend` method from the `boot` method of a service provider:

```
<?php

namespace App\Providers;

use Session;
use App\Extensions\MongoSessionStore;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore;
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Note that 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 MongoHandler 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) {}
}
```

Since these methods are not as readily understandable as the cache `StoreInterface`, 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 simply 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 `$sessionId` to some persistent storage system, such as MongoDB, Dynamo, etc.
- The `destroy` method should remove the data associated with the `$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.

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