Database: Getting Started

Introduction

Laravel makes interacting with databases extremely simple across a variety of database backends using either raw SQL, the fluent query builder, and the Eloquent ORM. Currently, Laravel supports four databases:

- MySQL 5.6+ (Version Policy)
- PostgreSQL 9.4+ (Version Policy)
- SQLite 3.8.8+
- SQL Server 2017+ (Version Policy)

Configuration

The database configuration for your application is located at `config/database.php`. In this file you may define all of your database connections, as well as specify which connection should be used by default. Examples for most of the supported database systems are provided in this file.

By default, Laravel's sample environment configuration is ready to use with Laravel Homestead, which is a convenient virtual machine for doing Laravel development on your local machine. You are free to modify this configuration as needed for your local database.

SQLite Configuration

After creating a new SQLite database using a command such as `touch database/database.sqlite`, you can easily configure your environment variables to point to this newly created database by using the database's absolute path:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

To enable foreign key constraints for SQLite connections, you should add the `foreign_key_constraints` option to your `config/database.php` configuration file:

```
'sqlite' => [
    // ...
    'foreign_key_constraints' => true,
],
```

Configuration Using URLs

Typically, database connections are configured using multiple configuration values such as `host`, `database`, `username`, `password`, etc. Each of these configuration values has its own corresponding environment variable. This means that when configuring your database connection information on a production server, you need to manage several environment variables.

Some managed database providers such as Heroku provide a single database "URL" that contains all of the connection information for the database in a single string. An example database URL may look something like the following:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

These URLs typically follow a standard schema convention:

```
driver://username:password@host:port/database?options
```

For convenience, Laravel supports these URLs as an alternative to configuring your

database with multiple configuration options. If the `url` (or corresponding `DATABASE_URL` environment variable) configuration option is present, it will be used to extract the database connection and credential information.

Read & Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
         ],
    ],
    'sticky'    => true,
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix'    => '',
],
```

Note that three keys have been added to the configuration array: `read`, `write` and `sticky`. The `read` and `write` keys have array values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` array.

You only need to place items in the `read` and `write` arrays if you wish to override the values from the main array. So, in this case, `192.168.1.1` will be used as the host for the "read" connection, while `192.168.1.3` will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` array will be shared across both connections.

The `sticky` Option

The `sticky` option is an *optional* value that can be used to allow the immediate reading of records that have been written to the database during the current request cycle. If the `sticky` option is enabled and a "write" operation has been performed against the database during the current request cycle, any further "read" operations will use the "write" connection. This ensures that any data written during the request cycle can be immediately read back from the database during that same request. It is up to you to decide if this is the desired behavior for your application.

Using Multiple Database Connections

When using multiple connections, you may access each connection via the `connection` method on the `DB` facade. The `name` passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file:

```
$users = DB::connection('foo')->select(...);
```

You may also access the raw, underlying PDO instance using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Running Raw SQL Queries

Once you have configured your database connection, you may run queries using the `DB` facade. The `DB` facade provides methods for each type of query: `select`, `update`, `insert`, `delete`, and `statement`.

Running A Select Query

To run a basic query, you may use the `select` method on the `DB` facade:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);

        return view('user.index', ['users' => $users]);
    }
}
```

The first argument passed to the `select` method is the raw SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the `where` clause constraints. Parameter binding provides protection against SQL injection.

The `select` method will always return an `array` of results. Each result within the array will be a PHP `stdClass` object, allowing you to access the values of the results:

```php
foreach ($users as $user) {
    echo $user->name;
}
```

Using Named Bindings

Instead of using `?` to represent your parameter bindings, you may execute a query using named bindings:

```php
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Running An Insert Statement

To execute an `insert` statement, you may use the `insert` method on the `DB` facade. Like `select`, this method takes the raw SQL query as its first argument and bindings as its second argument:

```php
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Running An Update Statement

The `update` method should be used to update existing records in the database. The number of rows affected by the statement will be returned:

```php
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

Running A Delete Statement

The `delete` method should be used to delete records from the database. Like `update`, the number of rows affected will be returned:

```php
$deleted = DB::delete('delete from users');
```

Running A General Statement

Some database statements do not return any value. For these types of operations, you may use the `statement` method on the `DB` facade:

```php
DB::statement('drop table users');
```

Listening For Query Events

If you would like to receive each SQL query executed by your application, you may use the `listen` method. This method is useful for logging queries or debugging. You may register your query listener in a service provider:

```php
<?php
```

```php
namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql
            // $query->bindings
            // $query->time
        });
    }
}
```

Database Transactions

You may use the `transaction` method on the `DB` facade to run a set of operations within a database transaction. If an exception is thrown within the transaction `Closure`, the transaction will automatically be rolled back. If the `Closure` executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the `transaction` method:

```php
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

Handling Deadlocks

The `transaction` method accepts an optional second argument which defines the number of times a transaction should be reattempted when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

```php
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
}, 5);
```

Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the `beginTransaction` method on the `DB` facade:

```php
DB::beginTransaction();
```

You can rollback the transaction via the `rollBack` method:

```php
DB::rollBack();
```

Lastly, you can commit a transaction via the `commit` method:

```php
DB::commit();
```

The `DB` facade's transaction methods control the transactions for both the query builder and Eloquent ORM.

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization

Resources

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

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova

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.

Authorization
Artisan Console
Database
Eloquent ORM
Testing

Jobs
Certification
Forums

Cabet
DevSquad
Ideil
Cyber-Duck
ABOUT YOU
Become A Partner

Nova
Echo
Valet
Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite