**1)Laravel eloquent Lazy Vs. Eager Loaded**

The Eloquent (ORM) in laravel is working amazingly and provide very simple ways for accessing to the database. In this article we will understand about what is lazy and eager loading in Laravel eloquent and how it’s works in the background.

#> Eloquent Relation

The first step we need to define relationship between models. In this example I will use two models, **house** and **city**. House is belong to one city, and city has many houses. Let’s see the relationship in model **House**:

**class** House **extends** Model  
{  
 **protected** $fillable = [  
 'title', 'description', 'price'  
 ];  
  
 **public function** city() {  
 **return** $this->belongsTo('App\City');  
 }  
}

*\*\*\* Note: for many to one relationship (house is belong to one city) the function name is singular. [****public function city()****]*

**class** City **extends** Model  
{  
 **protected** $fillable = [  
 'name', 'code'  
 ];  
  
 **public function** houses() {  
 **return** $this->hasMany('App\House');  
 }  
}

*\*\*\* Note: for one to many relationship (city has many houses) the function name is plural. [****public function houses()****]*

$houses = House::*all*();

By default, accessing data in eloquent is **“Lazy loaded”**, in the above code, we get all data in the house table, the query running behind that is:

select \* from `houses`

In this step the data in the relationship table (city) is not retrieved yet, if we want to access the data in the relationship table, we can access like this:

**foreach** ($houses **as** $house) {  
 **echo** $house->city->name; // this is lazy loaded  
}

the query running behind that is:

*select* \* *from* `cities` *where* `cities`.`id` = ? *limit* 1  
...

The process will loop all records in houses and execute 1 query for each loop to get the city data, example that we have 20 records of house, the query to get the data in the relationship table will execute 20 times + 1 original query for getting the house data, the time of query for **“Lazy loaded”** is **N+1.**

#> Eager Loading

Sometimes it is useful to use eager loading in your application, for example, you are calling the data by using Ajax, in this case we have to use eager loading to prepare all data include the data in its relationship table before response result to ajax. To use eager loading, just add the***with*** method to your eloquent.

$houses = House::*with*('city')->get();

Now all data in house and data in the relationship table loading in the same time, the query running behind that is:

*select* \* *from houses  
select* \* *from cities where id* in (1, 2, 3, 4, 5, ...)

There are only 2 queries execute when using eager loading. As you can see the queries above, one query is to get all data in houses table, and other query is get all data in cities table that have **id** matched to the **city\_id** in houses table using **IN** operation.

#> Lazy & Eager Loading

In some case that is useful to eager loading dynamically, we can decide weather the data in relationship table need to load.

$houses = House::*all*();  
  
**if**($isLoad) {  
 $houses->load('city');   
}

we can use **load** method to load the relationship data under specific condition

#> Conclusion

Now you understand the process and hopefully, it can help to understand the basic of lazy loaded and eager loading.

$houses = House::*all*();  
  
**if**($isLoad) {  
 $houses->load('city');   
}

**2)How to use Active Record in Laravel**

Active Record Implementation is an architectural pattern found in software engineering that stores in-memory object data in relational databases. Active Record facilitates the creation and use of business objects whose data is required to persistent in the database. Laravel implements Active Records by Eloquent ORM. Below is sample usage of Active Records Implementation is Laravel.

$product = new Product;

$product->title = 'Iphone 6s';

$product->save();

Active Record style ORMs map an object to a database row. In the above example, we would be mapping the Product object to a row in the products table of database.

3) **What are Closures in Laravel?**

Closure is a PHP feature so you can use it anywhere, not just Laravel.

Laravel uses Closures quite a bit and the reason is mostly boils down to flexibility.

Just imagine you are the author of Eloquent builder and you want a function named ‘with’ which fetches all related models (eg Post::with(‘comments’))

Now how will you go about it, maybe like following?

<?php

function with($relation) {

return allModelsOf($relation);

}

This is an oversimplification but you get the idea. This function will return all the related models (All comments of a post).

But! What if user wants only the last 2 days comments, or the comments with most up-votes, or maybe collapsed comments.

This means user want to supply his own custom logic so there should be any way to execute user’s logic rather than running our own logic and end up with same data everytime. This function is not flexible.

So you can use closures in this situation and provide user with a tool to pass his custom logic to pass to the actual function like this.

<?php

Post::with('comments', function ($builder) {

// Get me all collapsed comments

return $builder->whereCollapsed(true);

});

So a simplified implementation will look like following

<?php

function with($relation, Closure $closure = null) {

if(!is\_null($closure)) {

// means user has supplied his own logic so use that

return $closure($this);

}

// otherwise just do as before

return allModelsOf($relation);

}

4) What is a Dependency injection in Laravel?

If you have been using a PHP framework with a Service Container for a while now, you may have noted that you can use Dependency Injection on almost anywhere, easing most of the hard-wiring in a constant manner. In Laravel, you note this when instantiating classes like Controllers, Middlewares, Jobs, Listeners, and even Notifications.

Controllers methods are also called using type-hinting, which means you can put the class you need in the method and the Service Container will resolve it automatically. Pretty much like you do when you receive a Request and you want to validate some input:

public function post(Request $request)

{

$request->validate([

// ...

]);

// .. do something

}

Okay. Wouldn’t be nice if you could do the same with your code?

Well, the Service Container allows this out of the box.

Call this method with magic

The Service Container can be called using the app() method, since this is basically a helper that returns the Application instance, which extends the original Container class. Most of the time you’re dealing with this instance when asking something to the application, like the environment its on.

Going back to the magic, the call() method is what we’re looking for, which in return calls the BoundMethod class — a bunch of static helpers to check what and how to call something. The documentation in the source code says you can push a Closure or a class name with the method to call and it will automatically execute it while injecting its dependencies.

I’m gonna put this in sections, so you don’t get lost looking for a particular way of using this method.

Using a simple string

So, for example, we will call a class method using the class name and method like this:

$result = app()->call('App\MyClass@myMethod');

Wait a moment… Is this like defining a Route? Well, yeah, the Router uses the same syntaxis, but weirdly it doesn’t use the call method directly but uses its own way to do it. Anyway, that’s for another article.

The call() method will automatically resolve the class, inject dependencies if it asking for them, and do the same to the method itself. No need to instance the class manually:

class MyClass

{

protected $foo;

public function \_\_construct(Foo $foo)

{

$this->foo = $foo;

}

public function myMethod(Bar $bar)

{

return $foo->doSomethingWith($bar);

}

}

Why would you do this? Well, this can become very convenient to not clog your class full of dependencies in the \_\_construct(), specially when you need only some some in particular places, like when it happens when using Controllers methods. Also, a method may need only one thing, while in the construct it will be always instantiated no matter what method is hit later.

The “problem” with this technique is that is just for one-time only uses. In other words, once the container class fires the method for you, no class instance will be saved inside the Service Container, but instead, you will get the method result. The only exception for not-saving-the class is when you registered the class as “shared” (singleton) beforehand, like it happens inside Service Providers:

public function register()

{

// When someone calls for this class, anywhere, ensure

// the instance is saved inside the container so we

// can use it in other parts of the application.

$this->app->singleton(\App\MyClass::class);

}

When you need the class instance to do something more, you’re better of instantiating the class separately and then calling the method manually, or asking the Service Container to do it for you.

Using an instance

Luckily, if you already have your class instanced, then you can just tell the container to use the instance and call the method by putting them both inside an array.

$instance = new MyClass($foo);

$result = app()->call([$instance, 'myMethod']);

$instance->doSomethingMore();

Okay, that’s nice, the container will inject whatever the method needs automatically.

Again, instantiating the class is up to you. Personally, I would avoid calling the Service Container unless strictly necessary.

Adding parameters

What about parameters? What if you need to put something that the Service Container wouldn’t guess. Easy, we can just issue them as an array:

$result = app()->call('App\MyClass@myMethod', [

'red',

'cool',

]);

Hey, but what if we need to add a class that uses Dependency Injection? No problem, you don’t need to instance the class manually or call the Service Container to resolve it beforehand. Just add it and the call() method will automatically call the container to resolve the dependencies of that class if needed.

$result = app()->call('App\MyClass@myMethod', [

'status' => 'good',

'cool' => true,

'service' => SuperService::class,

]);

Some parameter shenanigans: Always try to use $key => $value for setting parameters. The framework may not guess properly what goes where if you have optional parameters.

Obviously, it will respect the default variable value only if it can’t be instanced. For example, if the a method expects an instance of something, but it’s null, and you don’t pass neither an instance or the class name, the Service Container will try to resolve it. If you really want to pass null, just pass null.

app()->call('App\MyClass@myMethod', [

'service' => null,

]);

Default method

The call method accepts a third parameter which is a default method to call. In other words, if you dare to only call a class without pointing the method name, a default will be used.

$result = app()->call('App\MyClass', [

// ... may be some arguments

], 'defaultMethod');

This default method is good when you don’t have control on what the developer will put there, like just the class name, or you may expect an Invokable class (more on that later). In the latter case, the third method it’s a way to cover your ass if you’re making a package.

Calling closure, callbacks and others callables

Wait, what? You can call Closures? You can inject dependencies on a Closure? WHAT SORCERY IS THIS? Okay, calm down, you can and its awesome.

When the method receives callable, like a Closure or an invokable class, it will use the service container to resolve the arguments of it. Let’s start with the first.

Methods and Closures

Callables are very useful when you need to inject executable logic for later. The Service Container can resolve it like it was nothing.

$closure = function (Foo $foo) {

return $foo->doSomething();

};

$result = app()->call($closure);

If the closure needs parameters, you can use them with no problem:

$closure = function (Foo $foo, string $bar) {

return $foo->doSomethingWith($bar);

};

$result = app()->call($closure, [

'bar' => 'rofl',

]);

Obviously, when using a Closure or any other callable, you won’t be able to use a default method, but in that case you may want to replace it for your own default callable if none is received before the actual call.

Invokables

Invokables Classes, those that implement an \_\_invoke() method, are sort of left in limbo. If you issue an invokable class name, you will receive an exception:

ReflectionException : Function MyInvokableClass() does not exist

This is because the Class name is mistaken by a function name, so when its called, PHP will try to get the function name which doesn’t exists. Don’t get me wrong, but if you plan to call an invokable class, it’s recommended to put your dependencies in the method signature,

<?php

namespace App;

use \Illuminate\Contracts\Config\Repository as Config;

class MyInvokableClass

{

public function \_\_invoke(Config $config)

{

return $config->get('app.env');

}

}

and then call it by just instancing it:

app()->call(new MyInvokableClass);

What is an Observer?

Laravel framework comes with lots of awesome features, the outstanding one for me is the model observers.

According to the Laravel framework’s documentation:

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observers classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the model as their only argument. The make:observer Artisan command is the easiest way to create a new observer class.

The observers helps me to declutter my controller of clean-up codes that I might have to run before or after making a model event, and also gives me a way to plug into the model’s event’s lifecycle and run any logic I might see fit.

The model events that can be observed are spread across the model’s CRUD and includes:

Retrieved

Creating

Created

Updating

Updated

Saving

Saved

Deleting

Deleted

Restoring

Restored

The above events can be observed for every model in the Laravel Framework and business logic attached to it, you can also dispatch custom events from the observer and listen to it from other parts of your application.

You might not see yourself using the observer class just yet and that is fine, having fair knowledge about it too, is okay.

Before that, to create an observer class, run:

php artisan make:observer <observerName>

(replace observerName with the name of the model you are observing).

This will create a folder in your application’s app directory called Observers and store the observer class.

<?php

namespace App\Observers;

class BankObserver

{

// an empty observer class

}

From here you can populate the classes with methods that match events I listed earlier, it is worthy to note that you can attach a model to an observer when creating the observer, like:

php artisan make:observer <observerName> -m=<ModelName>

This will create the class with some method filled in by default, see below.

<?php

namespace App\Observers;

use App\Task;

class Task

{

/\*\*

\* Handle the task "created" event.

\*

\* @param \App\Task $task

\* @return void

\*/

public function created(Task $task)

{

//

}

/\*\*

\* Handle the task "updated" event.

\*

\* @param \App\Task $task

\* @return void

\*/

public function updated(Task $task)

{

//

}

/\*\*

\* Handle the task "deleted" event.

\*

\* @param \App\Task $task

\* @return void

\*/

public function deleted(Task $task)

{

//

}

/\*\*

\* Handle the task "restored" event.

\*

\* @param \App\Task $task

\* @return void

\*/

public function restored(Task $task)

{

//

}

/\*\*

\* Handle the task "force deleted" event.

\*

\* @param \App\Task $task

\* @return void

\*/

public function forceDeleted(Task $task)

{

//

}

}

You might be wondering what model action triggers what observer action, I’ll briefly explain the different methods and what triggers them.

Retrieved — This observer method is called when a model record is retrieved from the database.

Model::findOrFail($id); //this triggers the retrieved method in the observer class

Creating — This observer method is called when a model record is in the process of creation, and not yet stored into the database, this is before the id, and default timestamps are generated for the model, at this point you can dynamically check for and assign a default value to a missing column.

Created — This observer method is called after a model record is created successfully. If there is an error in the process of creation, say a missing column data, this method doesn’t get called.

Model::create([]); //this triggers the creating method first, then created method in the observer class.

Updating — This observer method is called when a model record is in the updating process, at this point, the updates has not yet been persisted to the database.

Updated — This observer method is called after a model record is updated successfully. If there is an error in the process of updating, this method doesn’t get called.

Model::update([]); //this triggers the creating method first, then created method in the observer class.

Saving and Saved — These model observer methods might seem a bit like a swiss army knife, it gets called before and after any event that requires persistence of data to the database, so if you’re creating a new model record, the saving method runs first, then the creating method, then the created method and finally the saved method, the same routine applies when updating a model, saving, updating, updated, saved.

Deleting — This observer method is called when a model record is in the deletion process, at this point, the record has not yet been deleted from the database, and using its id to retrieve it from the database will return appropriate data.

Deleted — This observer method is called after a model record is successfully deleted, at this point, the record has been deleted from the database.

Model::destroy($id);

Restoring and Restored — These observer methods are called when a deleted model record is restored (using soft deletes implementation)

Important things to note:

The updating and updated methods only run when the update changes a column of the model in the database, as such, if the update request does not effect a change, the updating and updated observers don’t trigger, only the saving and saved methods get triggered.

When restoring a deleted record, series of methods gets triggered one after the other, retrieved, restoring, saving, updating, updated, saved, then restored.

In any case where you want to make a model event without triggering any observer method, you can save it without observer events. An example of a method to use when creating a model without triggering any of the events:

public function saveQuietly(array $options = [])

{

return static::withoutEvents(function () use ($options) {

return $this->save($options);

});

}

Note: This method should be added in the respective model.

You can refactor it to suit other model events, as necessary.

Finally, the last part, binding the observer to a particular model.

This can be done in the boot method of the AppServiceProvider’s class:

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Model::observe(Observer::class);

}

Model is the model to be observed and observer is the observer class

6) Explain What are the Default packages Laravel?

Below are some official packages provided by Laravel

Dusk: Laravel Dusk provides an expressive, easy-to-use browser automation and testing API. By default, Dusk does not require you to install JDK or Selenium on your machine.

Envoy: Laravel Envoy provides a clean, minimal syntax for defining common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems.

Horizon: Horizon provides a beautiful dashboard and code-driven configuration for your Laravel powered Redis queues. Horizon allows you to easily monitor key metrics of your queue system such as job throughput, runtime, and job failures.

Cashier: Laravel Cashier provides an expressive, fluent interface to Stripe's and Braintree's subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading writing. In addition to basic subscription management, Cashier can handle coupons, swapping subscription, subscription "quantities", cancellation grace periods, and even generate invoice PDFs.

Envoy: Laravel Envoy provides a clean, minimal syntax for defining common tasks you run on your remote servers. Using Blade style syntax, you can easily setup tasks for deployment, Artisan commands, and more. Currently, Envoy only supports the Mac and Linux operating systems.

Passport: Laravel makes API authentication a breeze using Laravel Passport, which provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the League OAuth2 server that is maintained by Alex Bilbie.

Scout: Laravel Scout provides a simple, driver based solution for adding full-text search to your Eloquent models. Using model observers, Scout will automatically keep your search indexes in sync with your Eloquent records.

Socialite: Laravel Socialite provides an expressive, fluent interface to OAuth authentication with Facebook, Twitter, Google, LinkedIn, GitHub and Bitbucket. It handles almost all of the boilerplate social authentication code you are dreading writing.

Telescope: Laravel Telescope is an elegant debug assistant for the Laravel framework. Telescope provides insight into the requests coming into your application, exceptions, log entries, database queries, queued jobs, mail, notifications, cache operations, scheduled tasks, variable dumps and more. Telescope makes a wonderful companion to your local Laravel development environment.

7) What is Route caching?

Performance optimization within our PHP code isn't always something at the forefront of our minds, but our backend code--especially our more complex operations--can impact our sites' request times to the tune of tens and maybe even hundreds of milliseconds. It may not sound like much, but a few hundred milliseconds can mean a huge difference in the perceived quickness of your applications.

Whether or not you were aware of it, the routing logic in Laravel 4 and earlier--especially as you have more and more routes in your application--were one such place for performance bottlenecks. A site with just a few hundred routes could, in the past, lose up to a half second just for the framework to register those routes. Fear no more, as Laravel 5 introduces route caching, optimizing the performance of your routes (except Closure routes, so it's time to move them all to controllers).

Using Route Caching

There's not a lot to using this feature, honestly. There's a new Artisan command, route:cache, which serializes the results of your routes.php file--it's performing the operation of parsing the routes once and then storing those result. Sort of like pre-compiling a Handlebars template, if you've ever done that before.

That's it! Now your routes are being parsed from the cached file, not your routes file. You can make all the changes you want to routes.php and the routing of your app won't change until you re-cache.

8)What are Jobs and Middleware?

This adds an easy way to have job specific middleware for queued jobs. Global job middleware were actually already possible by calling Bus::pipeThrough([]) in a service provider during the application boot process…These middleware provide a convenient location to wrap jobs in some logic before they are executed.

You define middleware by specifying a middleware() method on the job class which returns an array of middleware objects. From the pull request, here’s an example:

public function middleware()

{

return [new SomeMiddleware];

}

And here’s an example of the middleware class:

class SomeMiddleware

{

public function handle($command, $next)

{

// Do something...

return $next($command);

}

}

You can also specify middleware when dispatching a job:

SomeJob::dispatch()->through([new SomeMiddleware]);

9)What are Deferred Providers in laravel?

**Deferred Providers in laravel** are every service provider class that will be packed for your implementation. Although more of these are “deferred” providers, explaining that they will be unloaded on all requests, only during the facilities their activities are required. Hence, if one's provider is only recording bindings in the Service Container, one can wish to defer its record until one of the recorded bindings is truly required. This will enhance the performance of the applications as it is not filled from the filesystem on all requests. To defer the filling of a provider, execute the \Illuminate\Contracts\Support\DeferrableProvider interface and explain a Provides procedure. The Provides process should send back the facility container bindings recorded by the provider.

10) What is namespace in Laravel?

Namespaces can be defined as a class of elements in which each element has a unique name to that associated class. It may be shared with elements in other classes.

Declaration of namespace

The use keyword allows the developers to shorten the namespace.

use <namespace-name>;

The default namespace used in Laravel is App, however a user can change the namespace to match with web application. Creating user defined namespace with artisan command is mentioned as follows −

php artisan app:name SocialNet

Select Git

The namespace once created can include various functionalities which can be used in controllers and various classes.

SQL

11) Explain Joins with an example  
Write a complex query?

Problem

Consider the sample code below, which use the Product and Category table to return the category name.

$product = Product::where('id', $productId)->first();

$productCategory = Category::where('id', $product->category\_id)->pluck('name')->first();

Above code example is simple enough to understand but it uses two separate calls to the database. The first query must return before the second query can be triggered

12) Write CRUD in Laravel Query Builder?

Using A Raw Expression

$users = DB::table('users')

->select(DB::raw('count(\*) as user\_count, status'))

->where('status', '<>', 1)

->groupBy('status')

->get();

Inserts

Inserting Records Into A Table

DB::table('users')->insert(

['email' => 'john@example.com', 'votes' => 0]

);

Inserting Records Into A Table With An Auto-Incrementing ID

If the table has an auto-incrementing id, use insertGetId to insert a record and retrieve the id:

$id = DB::table('users')->insertGetId(

['email' => 'john@example.com', 'votes' => 0]

);

Note: When using PostgreSQL the insertGetId method expects the auto-incrementing column to be named "id".

Inserting Multiple Records Into A Table

DB::table('users')->insert([

['email' => 'taylor@example.com', 'votes' => 0],

['email' => 'dayle@example.com', 'votes' => 0]

]);

Updates

Updating Records In A Table

DB::table('users')

->where('id', 1)

->update(['votes' => 1]);

Incrementing or decrementing a value of a column

DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);

You may also specify additional columns to update:

DB::table('users')->increment('votes', 1, ['name' => 'John']);

Deletes

Deleting Records In A Table

DB::table('users')->where('votes', '<', 100)->delete();

Deleting All Records From A Table

DB::table('users')->delete();

Truncating A Table

DB::table('users')->truncate();

13) What are Accessors and Mutators in Eloquent and why should you use them?

In order to format the Eloquent Attributes Accessors and Mutators are used. Laravel Accessors and Mutators are custom, user defined methods.

Accessors are used to format the attributes when you retrieve them from database.

Whereas, Mutators are used to format the attributes before saving them into the database.

How To Define An Accessor ?

The syntax used for defining an Accessor is very simple, getFooAttribute(). Here Foo is the “studly” cased name of the column you wish to access.

Let’s take an example. In this example, we’ll define an accessor for the first\_name attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the first\_name attribute:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Get the user's first name.

\*

\* @param string $value

\* @return string

\*/

public function getFirstNameAttribute($value)

{

return ucfirst($value);

}

}

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Get the user's first name.

\*

\* @param string $value

\* @return string

\*/

public function getFirstNameAttribute($value)

{

return ucfirst($value);

}

}

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may access the first\_name attribute on a model instance:

$user = App\User::find(1);

$firstName = $user->first\_name;

$user = App\User::find(1);

$firstName = $user->first\_name;

Accessor can also be used to return new, computed values from existing attributes:

/\*\*

\* Get the user's full name.

\*

\* @return string

\*/

public function getFullNameAttribute()

{

return "{$this->first\_name} {$this->last\_name}";

}

/\*\*

\* Get the user's full name.

\*

\* @return string

\*/

public function getFullNameAttribute()

{

return "{$this->first\_name} {$this->last\_name}";

}

How To Define A Mutator ?

To define a mutator, define a setFooAttribute method on your model where Foo is the “studly” cased name of the column you wish to access. So, again, let’s define a mutator for the first\_name attribute. This mutator will be automatically called when we attempt to set the value of the first\_name attribute on the model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Set the user's first name.

\*

\* @param string $value

\* @return void

\*/

public function setFirstNameAttribute($value)

{

$this->attributes['first\_name'] = strtolower($value);

}

}

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* Set the user's first name.

\*

\* @param string $value

\* @return void

\*/

public function setFirstNameAttribute($value)

{

$this->attributes['first\_name'] = strtolower($value);

}

}

The mutator will receive the value that is being set on the attribute, allowing you to manipulate the value and set the manipulated value on the Eloquent model’s internal $attributes property. So, for example, if we attempt to set the first\_name attribute to Sally:

$user = App\User::find(1);

$user->first\_name = 'Sally';

$user = App\User::find(1);

$user->first\_name = 'Sally';

In this example, the setFirstNameAttribute function will be called with the value Sally. The mutator will then apply the strtolower function to the name and set its resulting value in the internal $attributes array.

14)What is CSRF and JWT token?

CSRF:

Laravel makes it easy to protect your application from [cross-site request forgery](https://en.wikipedia.org/wiki/Cross-site_request_forgery) (CSRF) attacks. Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user.

Laravel automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application.

Anytime you define an HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the @csrf Blade directive to generate the token field:

JWT:

JSON Web Token (JWT) is an open standard that allows two parties to securely send data and information as JSON objects. This information can be verified and trusted because it is digitally signed.

JWT authentication has aided the wider adoption of stateless API services. It makes it convenient to authorise and verify clients accessing API resources. It is a critical part of the authentication system in javascript powered applications