**Interview Questions**

**1)What is a dependency class?**

What is a Dependency? **Whenever a class A uses another class or interface B, then A depends on B**. A cannot carry out it's work without B, and A cannot be reused without also reusing B. In such a situation the class A is called the "dependant" and the class or interface B is called the "dependency".

**What is dependency in Laravel?**

In Laravel, dependency injection is **the process of injecting class dependencies into a class through a constructor or setter method**. This allows your code to look clean and run faster. Dependency injection involves the use of a Laravel service container, a container that is used to manage class dependencies.

How is dependency injection implemented in Laravel?

The service container is one of the most important pieces of the framework. It is responsible for managing your class dependencies and allows you to perform dependency injection.

**Why should you use dependency injection?**

If you are new to Laravel / modern PHP frameworks you might wonder what "dependency injection" actually means, so let me give you an example.

In your PHP classes, no matter if its a controller, a "service" class, or anything else, you will at some point need to include "dependencies".

Lets say that your application has a **GoogleMaps** class, which needs some API credentials and you want to use that class in one of your controllers:

**class** **GoogleMaps**

{

// ...

}

**class** **LocationController**

{

**public** **function** **index**(Request $request)

{

$maps = **new** GoogleMaps();

$coordinates = $maps->getCoordinates($request->get('address'));

}

}

As you can see, inside the **index** method, we created a new instance of the **GoogleMaps** class.

So what's the problem with this, you might ask. Well, what if you want to use the same GoogleMaps class in another part of your application? Or maybe the GoogleMaps class has additional dependencies that you would need to pass.

This quickly gets cumbersome to manage. Because of this, we have the concept of dependency injection. In fact, in the code snippet above, we already use dependency injection. We inject the **Request** class into our controller method, without creating a new instance of it.

**Automatic dependency injection**

By default, Laravel allows you to use a completely configuration-free dependency injection system. All you need to do is type hint the class that you want to inject – for example in your controller, middleware, jobs, event listeners, etc.

The example above would become:

**class** **LocationController**

{

**public** **function** **index**(Request $request, GoogleMaps $maps)

{

$coordinates = $maps->getCoordinates($request->get('address'));

}

}

By simply type-hinting the GoogleMaps class in our controller method, Laravel is automatically looking up the **GoogleMaps** class and its dependencies, tries to create a new instance of the class and pass it to the controller method.

In a controller, you might also need a dependency throughout all of the controller methods. In this case, you can simply inject the class in the controllers' constructor:

**class** **LocationController**

{

**protected** GoogleMaps $maps;

**public** **function** **\_\_construct**(GoogleMaps $maps)

{

**$this**->maps = $maps;

}

**public** **function** **index**(Request $request)

{

$coordinates = **$this**->maps->getCoordinates($request->get('address'));

}

}

This zero-configuration dependency injection is amazing and you will most likely not need to manually configure dependency injection. There are, however, certain situations where you might want to handle dependency injection manually.

=====================================================================================================================END===========================================

**2)What is relation in Laravel?**

To define a relationship, **we need first to define the post() method in User model**. In the post() method, we need to implement the hasOne() method that returns the result. Let's understand the one to one relationship through an example. First, we add the new column (user\_id) in an existing table named as posts.

Visit this link-

<https://www.w3resource.com/laravel/eloquent-relationships.php#:~:text=You%20can%20query%20the%20posts,builder%20methods%20on%20the%20relationship>.

===============================================================================================================END=================================================

**3)Role of model in Laravel**

Laravel is an **MVC** based PHP framework. In MVC architecture, ‘**M**’ stands for ‘**Model**’. A **Model** is basically a way for querying data to and from the table in the database. Laravel provides a simple way to do that using **Eloquent ORM (Object-Relational Mapping)**. Every table has a **Model** to interact with the table.

**Create a Model:** To create an **Eloquent Model**, Laravel provides an Artisan Command as follows:

php artisan make:model Article

After running the command above, a file is created with the name **Article.php** in the **app** directory. The file content looks as follows:

|  |
| --- |
| <?php    **namespace** App;    **use** Illuminate\Database\Eloquent\Model;    **class** Article **extends** Model {    //  } |

**Retrieve Data:** To retrieve data from the database, we can use two methods as described below:

* Here, the **all()** method will return all the data in the table in the form of an array and store it in the **$articles** variable.
* public function index() {
* $articles = \App\Article::all();
* return view('gfg')->with('articles', $articles);

}

* We can also get a particular record by using ‘**where()**’ as shown below:
* public function index() {
* $articles = \App\Article::all()->where('id', 1);
* return view('gfg')->with('articles', $articles);

}

**Insert Data:** To insert data in the database, we will use **save()** method as shown below:

* Here, we create a new Article and store it in the **$article** variable. The **topic** and **content** attribute are used to store the data specified in the respective columns in the table. And then ‘save()’ method is used to insert the data in the database. Then a successful message will be shown if the insert is successful.
* public function insert() {
* $article = new Article;
* $article->topic = "View in Laravel";
* $article->content = "View is the data display at the user end.";
* $article->save();
* echo "Insert Successful!";

}

**Note:** The *created\_at* and *updated\_at* field in the table will be automatically be inserted with the time stamp.

**Update Data:** To update data in the database, we will again use **save()** method as shown below:

* Here, the **find()** method is used to specify the record that we want to update in the database. The number in the parenthesis is of **id** i.e. primary key. Now a new value is given to the **topic** field of the record which will change the old one. And then ‘**save()**’ method is used to insert the data in the database. Then a successful message will be shown if the update is successful.
* public function update() {
* $article = \App\Article::find(1);
* $article->topic = "Laravel";
* $article->save();
* echo "Update Successful!";

}

**Note:** The *updated\_at* field in the table will be automatically be inserted with the time stamp.

**Delete Data:** To delete data in the database, we will use **delete()** method as shown below:

* Here, the **find()** method is used to specify the record that we want to delete from the database. The number in the parenthesis is of **id** i.e. primary key. Then the **delete()** method is used. Then a successful message will be shown if the delete is successful.
* public function delete() {
* $article = \App\Article::find(1);
* $article->delete();
* echo "Delete Successful!";

}

Below example illustrates each of them:

**Example:**

1. Create and Connect to a **MySQL Database**.
2. Create a migration using the following Artisan command:

php artisan make:migration create\_articles\_table

And then write the below code in the **up()** function in the migration file created at **database/migrations** directory.

Schema::create('articles', function (Blueprint $table) {

$table->bigIncrements('id');

$table->string('topic');

$table->string('content');

$table->timestamps();

});

1. Now run the migrate command to create migrations:

php artisan migrate

1. Now create a model using the below Artisan command:

php artisan make:model Article

The **Article.php** file, which is created in the **app** directory, should look like as follows:

|  |
| --- |
| <?php    **namespace** App;    **use** Illuminate\Database\Eloquent\Model;    **class** Article **extends** Model {      //  } |

1. Now create a controller using the below Artisan command:

php artisan make:controller ArticleController

The **ArticleController.php** file, which is created in the **app/Http/Controllers** directory, should look like as follows:

|  |
| --- |
| <?php    **namespace** App\Http\Controllers;    **use** Illuminate\Http\Request;    **class** ArticleController **extends** Controller {      //  } |

1. Now, you will have to change the controller file as below code or copy the code below and paste it in that controller file which was created in the previous step.

|  |
| --- |
| <?php    **namespace** App\Http\Controllers;    **use** App\Article;  **use** Illuminate\Http\Request;    **class** ArticleController **extends** Controller {        // Retrieve function  **public** **function** index() {            $articles = \App\Article::all();    **return** view('gfg')->with('articles', $articles);        }        // Insert function  **public** **function** insert() {        $article = **new** Article;        $article->topic = "View in Laravel";      $article->content = "View is the data display at the user end.";        $article->save();        echo "Insert Successful!";        }        // Update function  **public** **function** update() {            $article = \App\Article::find(1);            $article->topic = "Laravel";            $article->save();            echo "Update Successful!";        }        // Delete function  **public** **function** delete() {            $article = \App\Article::find(1);            $article->delete();            echo "Delete Successful!";        }  } |

1. Now, create a view with the name ‘**gfg.blade.php**’ in the ‘resources/views’ directory and the below code in that file.

|  |
| --- |
| <!DOCTYPE html>  <**html**>  <**head**>      <**title**>GeeksforGeeks</**title**>      <**style**>          body {              font-size: 20px;          }      </**style**>  </**head**>  <**body**>        <**h2**>Articles Topics</**h2**>      <**ol**>          @foreach($articles as $article)              <**li**>{{ $article->topic }}</**li**>          @endforeach      </**ol**>    </**body**>  </**html**> |

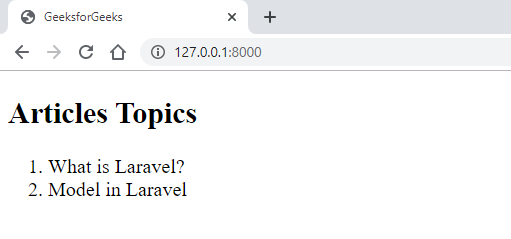
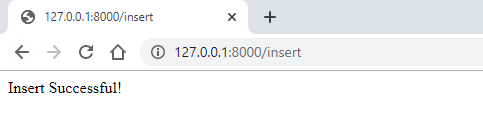
1. Now, create the routes by writing the following in the **web.php** file in the **routes** directory.

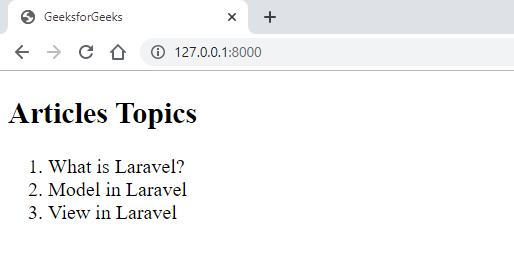
**Note:** Comment or remove any previous routes from the file.

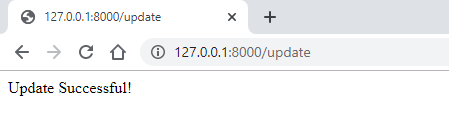
1. Route::get('/', 'ArticleController@index');
2. Route::get('/insert', 'ArticleController@insert');
3. Route::get('/update', 'ArticleController@update');
4. Route::get('/delete', 'ArticleController@delete');
5. Now run the Laravel app using the following Artisan command:

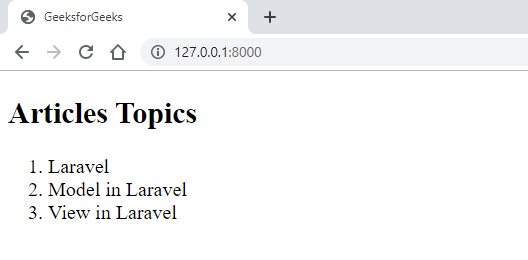
php artisan serve

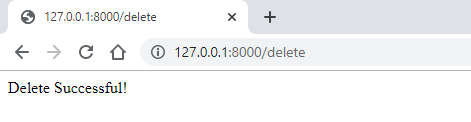
**Output:**

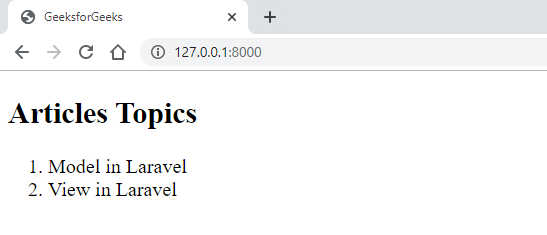
1. **For Index Function:**  
   
2. **For Insert Function:**  
   



1. **For Update Function:**  
   



1. **For Delete Function:**  
   



=========================================================================================================================END=======================================

**4)Localization in Laravel**

Localization feature of Laravel supports different language to be used in application. You need to store all the strings of different language in a file and these files are stored at **resources/views** directory. You should create a separate directory for each supported language. All the language files should return an array of keyed strings as shown below.

<?php

return [

'welcome' => 'Welcome to the application'

];

## Example

**Step 1** − Create 3 files for languages − **English, French**, and **German**. Save English file at **resources/lang/en/lang.php**

<?php

return [

'msg' => 'Laravel Internationalization example.'

];

?>

**Step 2** − Save French file at **resources/lang/fr/lang.php**.

<?php

return [

'msg' => 'Exemple Laravel internationalisation.'

];

?>

**Step 3** − Save German file at **resources/lang/de/lang.php**.

<?php

return [

'msg' => 'Laravel Internationalisierung Beispiel.'

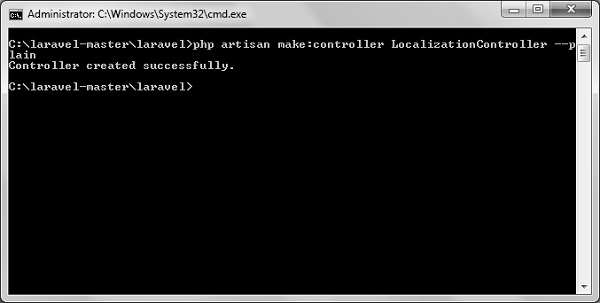
];

?>

**Step 4** − Create a controller called **LocalizationController** by executing the following command.

php artisan make:controller LocalizationController --plain

**Step 5** − After successful execution, you will receive the following output −



**Step 6** − Copy the following code to file

**app/Http/Controllers/LocalizationController.php**

**app/Http/Controllers/LocalizationController.php**

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

use App\Http\Controllers\Controller;

class LocalizationController extends Controller {

public function index(Request $request,$locale) {

//set’s application’s locale

app()->setLocale($locale);

//Gets the translated message and displays it

echo trans('lang.msg');

}

}

**Step 7** − Add a route for LocalizationController in **app/Http/routes.php** file. Notice that we are passing {locale} argument after localization/ which we will use to see output in different language.

**app/Http/routes.php**

Route::get('localization/{locale}','LocalizationController@index');

**Step 8** − Now, let us visit the different URLs to see all different languages. Execute the below URL to see output in English language.

http://localhost:8000/localization/en

**Step 9** − The output will appear as shown in the following image.

Laravel Internationalization

**Step 10** − Execute the below URL to see output in French language.

http://localhost:8000/localization/fr

**Step 11** − The output will appear as shown in the following image.

French Example

**Step 12** − Execute the below URL to see output in German language

http://localhost:8000/localization/de

**Step 13** − The output will appear as shown in the following image.

German Example

===========================================================================================================================END=====================================

**5)What is Collection in Laravel?**

Defining Laravel collections  
  
Laravel collections can be regarded as **modified versions of PHP arrays**. They are located in the Illuminate\Support\Collection directory and provide a wrapper to work with data arrays.

What is eloquent Collection?

Eloquent collections are **an extension of Laravel's Collection class with some handy methods for dealing with query results**. The Collection class itself, is merely a wrapper for an array of objects, but has a bunch of other interesting methods to help you pluck items out of the array.

What is difference between array and Collection?

**Arrays can hold only homogeneous data types elements.** **Collection can hold both homogeneous and and heterogeneous elements**. There is no underlying data structure for arrays and hence ready made method support is not available.

====================================================================================================================END============================================

**6)Composer in Laravel**

Composer in Laravel is **used to manage their dependencies**. Once Composer is installed, download the framework and extract its contents into a directory on your server. The primary task you should do after installing Laravel is to set your application key to a random string.