Using Laravel with GraphQL: A Beginner’s Guide

Using Laravel with GraphQL: A Beginner’s Guide

REST is common for APIs, but GraphQL is increasingly popular because it lets clients ask for exactly the data they need. Laravel integrates with GraphQL using community packages like `rebing/graphql-laravel`. In this guide, you’ll install GraphQL, define schemas and resolvers, and build a small UI to query data directly from your Laravel app.

1 – Install GraphQL Package

We’ll use the `rebing/graphql-laravel` package, a mature GraphQL server implementation for Laravel.

```bash
composer require rebing/graphql-laravel

php artisan vendor:publish --
provider="Rebing\GraphQL\GraphQLServiceProvider"

php artisan migrateCode language: Bash (bash)
```

This installs the package, publishes the config file (`config/graphql.php`), and prepares migrations if you plan to store persisted queries or cache.

2 – Create a GraphQL Type

GraphQL types describe what fields are available. Here's a `UserType` that maps to our `User` model.

```
// app/GraphQL/Types/UserType.php
namespace App\GraphQL\Types;

use App\Models\User;
use GraphQL\Type\Definition\Type;
use Rebing\GraphQL\Support\Type as GraphQLType;

class UserType extends GraphQLType
{
    protected $attributes = [
        'name' => 'User',
        'description' => 'A user object',
        'model' => User::class,
    ];

    public function fields(): array
    {
        return [
            'id' => [ 'type' => Type::nonNull(Type::int()) ],
            'name' => [ 'type' => Type::string() ],
            'email' => [ 'type' => Type::string() ],
            'created_at' => [ 'type' => Type::string() ],
        ];
    }
}Code language: PHP (php)
```

This type tells GraphQL that a `User` object has fields like `id`, `name`, and `email`. Types map directly to models or DTOs.

3 – Create a Query Resolver

Resolvers tell GraphQL how to fetch data. Let’s make a query to fetch users with optional limits.

```php
// app/GraphQL/Queries/UsersQuery.php
namespace App\GraphQL\Queries;

use App\Models\User;
use GraphQL\Type\Definition\Type;
use Rebing\GraphQL\Support\Facades\GraphQL;
use Rebing\GraphQL\Support\Query;

class UsersQuery extends Query
{
    protected $attributes = [
        'name' => 'users',
    ];

    public function type(): Type
    {
        return Type::listOf(GraphQL::type('User'));
    }

    public function args(): array
    {
        return [
            'limit' => [ 'type' => Type::int() ],
        ];
    }

    public function resolve($root, $args)
    {
        return User::query()
            ->limit($args['limit'] ?? 10)
            ->get();
    }
}
```
Code language: PHP (php)

This query returns a list of `User` objects. If `limit` is passed, it restricts the number of results; otherwise defaults to 10.

4 - Register Schema

Now wire the type and query into GraphQL's schema config so they are available to clients.

```php
// config/graphql.php (snippet)
'types' => [
    'User' => App\GraphQL\Types\UserType::class,
],

'schemas' => [
    'default' => [
        'query' => [
            'users' => App\GraphQL\Queries\UsersQuery::class,
        ],
    ],
],
```
Code language: PHP (php)

Types are registered by name, and queries point to their resolver classes. The default schema now has a `users` query available.

5 - Testing the GraphQL Endpoint

GraphQL endpoints are usually mounted at `/graphql`. Let's test with a simple query:

query { users(limit: 5) { id name email } }

This query fetches five users with only the requested fields (`id`, `name`, `email`). GraphQL won’t fetch extra columns unless you ask for them.

6 - Quick UI: GraphQL Explorer

Many GraphQL packages include a built-in UI like GraphiQL or Playground. You can enable it in dev, or build a minimal query tester:

```
<!-- resources/views/graphql/test.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container">
  <h1>GraphQL Tester</h1>

  <textarea id="query" class="form-control mb-3" rows="6">
{ users(limit: 3) { id name email } }
  </textarea>

  <button class="btn btn-theme mb-3" onclick="runQuery()">Run
Query</button>
  <pre id="result"></pre>
</div>

<script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script>
function runQuery() {
  const q = document.getElementById('query').value;
  axios.post('/graphql', { query: q })
    .then(res => {
      document.getElementById('result').textContent =
        JSON.stringify(res.data, null, 2);
    })
    .catch(err => {
      document.getElementById('result').textContent = err;
    });
}
</script>
@endsectionCode language: PHP (php)
```

This test page lets you run raw GraphQL queries and see results in JSON format, making development much faster without switching tools.

Wrapping Up

GraphQL provides a flexible alternative to REST. With Laravel and `rebing/graphql-laravel`, you can define types, queries, and resolvers that let clients request only the data they need. You also built a small query tester UI to try it out quickly. This approach reduces over-fetching and speeds up client development.

What's Next

- How to Build a Multi-Auth API with Laravel & Sanctum
- How to Add JWT Authentication to Laravel APIs
- Integrating Laravel with Third-Party APIs (Mail, SMS, Payment)