

[Using JSON Columns in Laravel Eloquent \(Practical Guide\)](#)

Using JSON Columns in Laravel Eloquent (Practical Guide)

JSON columns help you store flexible, semi-structured data without exploding your schema. In this practical guide, you'll create JSON columns, cast them in Eloquent, query nested paths efficiently, validate JSON payloads, update nested keys, and build a small UI to edit JSON-based user settings.

1 - Creating JSON Columns

Define JSON columns with the schema builder. On MySQL 5.7+/8 and PostgreSQL, JSON is a native type; on SQLite it's stored as text but works fine for most cases.

```
//
database/migrations/2025_08_27_000000_add_profile_json_to_users_table.
php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->json('profile')->nullable();    // e.g.
            {"country": "TR", "preferences": {"theme": "dark"}}
        });
    }
};
```

```
}

public function down(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('profile');
    });
}
};Code language: PHP (php)
```

This migration adds a nullable `profile` column intended to hold a JSON object with free-form keys such as `country` and `preferences`. Removing it in `down()` keeps the migration reversible.

Indexing tip: If you frequently filter by a nested property, consider indexing via a generated column (MySQL) or a GIN index (PostgreSQL JSONB).

```
-- MySQL: generated (virtual) column + index for profile->country
ALTER TABLE users
  ADD COLUMN country_code VARCHAR(2)
    GENERATED ALWAYS AS (JSON_UNQUOTE(JSON_EXTRACT(profile,
'$>country')))) VIRTUAL,
  ADD INDEX idx_users_country_code (country_code);

-- PostgreSQL: convert to JSONB and index
-- ALTER TABLE users ADD COLUMN profile JSONB;
CREATE INDEX idx_users_profile_gin ON users USING GIN ((profile));Code
language: SQL (Structured Query Language) (sql)
```

MySQL example projects `$>country` into a virtual column to index it. PostgreSQL's GIN index accelerates containment and path queries on JSONB.

2 - Casting JSON to Arrays/Objects in Eloquent

Use attribute casting so you can read/write JSON naturally as arrays. Eloquent will automatically JSON-encode/decode on save/fetch.

```
// app/Models/User.php
namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    protected $fillable = ['name', 'email', 'password', 'profile'];

    protected $casts = [
        'profile' => 'array', // or 'collection' if you prefer
        Illuminate\Support\Collection
    ];
}Code language: PHP (php)
```

Setting 'profile' => 'array' means \$user->profile is an array in PHP. Assigning an array automatically serializes back to JSON when the model is saved.

3 - Querying JSON Columns

Laravel supports vendor-agnostic JSON path syntax like `where('profile->country', 'TR')`, plus helpers such as `whereJsonContains` for arrays/objects.

```
use App\Models\User;

// Path operator (country == 'TR')
```

```
$turkish = User::where('profile->country', 'TR')->get();

// Array contains (roles includes 'admin')
$admins = User::whereJsonContains('profile->roles', 'admin')->get();

// Object contains (preferences has {"theme":"dark"})
$darkTheme = User::whereJsonContains('profile->preferences', ['theme'
=> 'dark'])->get();

// Null checks inside JSON
$missingPhone = User::whereNull('profile->phone')->get();
```

Code language: PHP (php)

where('profile->country', 'TR') targets a nested key under profile.
whereJsonContains is ideal when filtering arrays (roles) or matching object fragments (preferences contains theme: dark).

4 - Validating and Persisting JSON Input

Validate incoming JSON as an array and validate nested keys with dot notation. Then assign the validated array to the casted attribute.

```
// app/Http/Controllers/ProfileController.php
namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class ProfileController extends Controller
{
    public function update(Request $request)
    {
```

```
$data = $request->validate([
    'profile' => ['required','array'],
    'profile.country' => ['required','string','size:2'],
    'profile.phone' => ['nullable','string','max:30'],
    'profile.roles' => ['nullable','array'],
    'profile.roles.*' => ['string'],
    'profile.preferences' => ['nullable','array'],
    'profile.preferences.theme' =>
['nullable','in:light,dark'],
    'profile.preferences.notifications' =>
['nullable','array'],
    'profile.preferences.notifications.email' => ['boolean'],
    'profile.preferences.notifications.sms' => ['boolean'],
]);

$user = Auth::user();
$user->profile = $data['profile']; // cast handles JSON
$user->save();

return back()->with('status','Profile saved.');
```

}
}Code language: PHP (php)

The validator ensures `profile` is an array and that expected nested keys conform to your rules. Assigning the array to `$user->profile` triggers casting and JSON persistence.

5 - Updating Individual Nested Keys

You can update a single nested path using the JSON path key syntax. This avoids fetching and rewriting the whole object for simple toggles.

```
use Illuminate\Support\Facades\DB;
use App\Models\User;
```

```
// Toggle email notifications ON for a user
User::whereKey($id)->update([
    "profile->preferences->notifications->email" => true,
]);

// Append item into a JSON array (DB-specific alternative)
DB::table('users')
    ->where('id', $id)
    ->update([
        'profile' => DB::raw("JSON_SET(profile, '$.roles',
JSON_ARRAY_APPEND(JSON_EXTRACT(profile, '$.roles'), '$', 'editor'))")
    ]);Code language: PHP (php)
```

The first example uses Laravel's dotted JSON path to set a boolean inside nested objects. The second shows a DB-level array append using raw SQL (syntax varies by engine).

6 - UI: Editing JSON Settings in a Form

Here's a small UI: a controller method to show the form and a Blade template that binds to the JSON fields. Submissions hit `ProfileController@update` from above.

```
// app/Http/Controllers/ProfileController.php (add show() action)
public function show()
{
    $user = auth()->user();
    return view('profile.edit', ['user' => $user]);
}Code language: PHP (php)
```

The `show()` action passes the current user to the view so you can pre-populate the form from the JSON column.

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

```
@section('content')
<div class="container">
    <h1 class="mb-4">Profile Settings</h1>

    @if(session('status'))
        <div class="alert alert-success">{{ session('status') }}</div>
    @endif

    <form method="POST" action="{{ route('profile.update') }}">
        @csrf
        @method('PUT')

        <div class="row g-3">
            <div class="col-md-3">
                <label class="form-label">Country Code (ISO2)</label>
                <input name="profile[country]" class="form-control"
                    value="{{ data_get($user->profile, 'country') }}"
maxlength="2"/>
            </div>

            <div class="col-md-6">
                <label class="form-label">Phone</label>
                <input name="profile[phone]" class="form-control"
                    value="{{ data_get($user->profile, 'phone') }}" />
            </div>

            <div class="col-md-12">
                <label class="form-label">Roles (comma separated)</label>
                <input name="roles_csv" class="form-control"
                    value="{{ implode(',', (array) data_get($user->profile,
'roles', [])) }}" />
            </div>

            <div class="col-md-4">
                <label class="form-label">Theme</label>
                <select name="profile[preferences][theme]" class="form-
select">
                    @php $theme = data_get($user->profile, 'preferences.theme');
```

```
@endphp
        <option value="" {{ $theme ? '' : 'selected'
    }}>System</option>
        <option value="light" {{ $theme==='light' ? 'selected' : ''
    }}>Light</option>
        <option value="dark" {{ $theme==='dark' ? 'selected' : ''
    }}>Dark</option>
    </select>
</div>

<div class="col-md-4 form-check mt-4">
    @php $emailOn = (bool)
data_get($user->profile,'preferences.notifications.email'); @endphp
    <input id="notif_email" type="checkbox" class="form-check-
input"
        name="profile[preferences][notifications][email]"
value="1"
        {{ $emailOn ? 'checked' : '' }} />
    <label for="notif_email" class="form-check-label">Email
Notifications</label>
</div>

<div class="col-md-4 form-check mt-4">
    @php $smsOn = (bool)
data_get($user->profile,'preferences.notifications.sms'); @endphp
    <input id="notif_sms" type="checkbox" class="form-check-input"
        name="profile[preferences][notifications][sms]"
value="1"
        {{ $smsOn ? 'checked' : '' }} />
    <label for="notif_sms" class="form-check-label">SMS
Notifications</label>
</div>
</div>

<button class="btn btn-theme mt-3">Save Settings</button>
</form>
</div>
@endsectionCode language: PHP (php)
```


This form posts nested arrays using PHP's bracket notation (e.g., `profile[preferences][theme]`). In the controller, the validator receives a structured array, which you assign to `$user->profile`. The optional `roles_csv` field is a UX shortcut—if you want to support it, convert to an array before validation or within a form request.

7 - Performance & Gotchas

JSON columns are flexible but keep these in mind: index frequently filtered paths, avoid storing huge blobs that you rewrite often, and prefer stable columns for high-cardinality filters. When schema hardens, consider migrating hot properties into first-class columns.

Wrapping Up

You created JSON columns, cast them in Eloquent, queried nested paths, validated and saved structured input, and built a UI for editing JSON settings. With careful indexing and path updates, JSON gives you agility without sacrificing query power.

What's Next

- [How to Use Eloquent API Resources for Clean APIs](#)
- [Soft Deletes: Restore, Force Delete, and Prune Data](#)
- [Filtering and Searching with Eloquent Query Builder](#)