CET218

Advanced Web Programming

10 - Eloquent ORM & Database Interaction

Dr. Ahmed Said

Start →

Database Seeding

- Database seeding is the process of populating a database with initial data.
- Laravel provides a simple way to seed your database using the `db:seed` Artisan command.
- Seeders are classes that contain the logic to insert data into your database.
- Seeders are typically stored in the `database/seeders` directory.
  - Use the Artisan command to create a seeder:

```bash
php artisan make:seeder ProductSeeder
```

  - This creates a file like `database/seeders/ProductSeeder.php`.
- You can run all seeders using:

```bash
php artisan db:seed
```

- You can also run a specific seeder using:

```
php artisan db:seed --class=ProductSeeder
```

Database Seeder: Example

- Example: `database/seeders/ProductSeeder.php`

```php
namespace Database\Seeders;
...
class ProductSeeder extends Seeder
{
    use HasFactory;

    public function run()
    {
        // Using Eloquent Model
        Product::factory()->count(10)->create();

        // Using DB Facade
        DB::table('products')->insert([
            'name' => 'Sample Product',
            'description' => 'This is a sample product.',
            'price' => 19.99,
            'stock' => 100,
            'created_at' => now(),
            'updated_at' => now(),
        ]);
    }
}
```

Models & Eloquent ORM

Models & Eloquent ORM

- Eloquent is Laravel's Object-Relational Mapper - makes database interaction intuitive.

What is an ORM?

- Maps database tables to PHP classes (Models).
- Maps table rows to object instances.
- Provides an object-oriented way to query and manipulate data, abstracting away raw SQL (mostly).

Creating a Model

- Models are typically stored in the `app/Models` directory.
- By convention, model names are singular and table names are plural.
- Models should extend `Illuminate\Database\Eloquent\Model`.
- Use the Artisan command:

```bash
# Create a Product model
php artisan make:model Product

# Create a model and its migration file simultaneously
php artisan make:model Category -m

# Create a model, migration, factory, and resource controller
php artisan make:model Post -mfcr
```

- Creates a file like `app/Models/Product.php`.

Eloquent Conventions

- Eloquent makes assumptions to simplify configuration:
- **Table Name:** Plural, snake_case version of the class name (e.g., `Product` model → `products` table).
  - Override with: `protected $table = 'my_products';`
- **Primary Key:** Assumed to be `id`.
  - Override with: `protected $primaryKey = 'product_uuid';`
- **Incrementing PK:** Assumed to be `true`.
  - Set to `false` for non-incrementing keys (like UUIDs): `public $incrementing = false;`
- **Primary Key Type:** Assumed to be `int`.
  - Override with: `protected $keyType = 'string';`
- **Timestamps:** Assumes `created_at` and `updated_at` columns exist.
  - Disable with: `public $timestamps = false;`

Eloquent Conventions, cont'd

- **Soft Deletes:** Automatically manages soft deletes with `deleted_at` column.
- **Mass Assignment** is a security feature to prevent accidentally updating sensitive columns.
- **Mass Assignment Protection:** Protects against mass assignment vulnerabilities.
  - Use `$fillable` or `$guarded` properties to specify which attributes are mass assignable.
  - Example: `protected $fillable = ['name', 'description'];`
  - Example: `protected $guarded = ['id'];` (all except `id`).
- **Recommendation:** Use `$fillable` for clarity and security.
  - **Note:** `$guarded` is the opposite of `$fillable` - specify which attributes are NOT mass assignable.
  - **Caution:** Using an empty array `[]` for `$guarded` makes all attributes mass assignable, which can be risky.
  - **Best Practice:** Always define `$fillable` to explicitly state which attributes can be mass assigned.

Eloquent Conventions, cont'd

- **Soft Deletes:** Automatically manages soft deletes with `deleted_at` column.
- **Mass Assignment** is a security feature to prevent accidentally updating sensitive columns.
- **Mass Assignment Protection:** Protects against mass assignment vulnerabilities.
  - Use `$fillable` or `$guarded` properties to specify which attributes are mass assignable.
  - Example: `protected $fillable = ['name', 'description'];`
  - Example: `protected $guarded = ['id'];` (all except `id`).
- **Recommendation:** Use `$fillable` for clarity and security.
  - **Note:** `$guarded` is the opposite of `$fillable` - specify which attributes are NOT mass assignable.
  - **Caution:** Using an empty array `[]` for `$guarded` makes all attributes mass assignable, which can be risky.
  - **Best Practice:** Always define `$fillable` to explicitly state which attributes can be mass assigned.

Eloquent: Mass Assignment

- A security feature to prevent accidentally updating sensitive columns.
- When using `create()` or `update()` with an array of data (like `$request->all()` or `$request->validated()`), Eloquent needs to know which attributes are "safe" to be mass-assigned.
- **Option 1: `$fillable` (Allow list)**
  - Specify which attributes can be mass-assigned.

```php
// In App\Models\Product.php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;

class Product extends Model {
    /* The attributes that are mass assignable.
     * @var array<int, string>
     */
    protected $fillable = [
        'name', 'description', 'price', 'stock', 'is_active',
        // Add other safe attributes here
    ];

    // ... rest of the model
}
```

Eloquent: Mass Assignment

Option 2: `$guarded` (Block list)

Specify which attributes cannot be mass-assigned. All others are fillable. **Use `$fillable` OR `$guarded`, not both.**

```php
// In App\Models\Product.php

class Product extends Model
{
    /**
     * The attributes that aren't mass assignable.
     * Use an empty array to make ALL attributes mass assignable (use with caution!).
     *
     * @var array<int, string>
     */
    protected $guarded = [
        'id', // Usually guarded
        'is_admin', // Example of a sensitive field
    ];

    // ... rest of the model
}
```

- **Recommendation:** Prefer `$fillable` as it's more explicit and secure by default.

Eloquent: Mass Assignment

Option 2: `$guarded` (Block list)

Specify which attributes cannot be mass-assigned. All others are fillable. **Use `$fillable` OR `$guarded`, not both.**

```php
// In App\Models\Product.php

class Product extends Model
{
    /**
     * The attributes that aren't mass assignable.
     * Use an empty array to make ALL attributes mass assignable (use with caution!).
     *
     * @var array<int, string>
     */
    protected $guarded = [
        'id', // Usually guarded
        'is_admin', // Example of a sensitive field
    ];

    // ... rest of the model
}
```

- **Recommendation:** Prefer `$fillable` as it's more explicit and secure by default.

Eloquent: Basic CRUD Operations

- CRUD = Create, Read, Update, Delete

```php
...

// READ: Get a single product by ID
public function show(string $id) {
    // find() returns null if not found
    // $product = Product::find($id);
    // findOrFail() throws a ModelNotFoundException (results in 404) if not found
    $product = Product::findOrFail($id);
    return view('products.show', compact('product'));
}

...
```

Eloquent: Basic CRUD Operations

- CRUD = Create, Read, Update, Delete

```php
...

// READ: Get a single product by ID
public function show(string $id) {
    // find() returns null if not found
    // $product = Product::find($id);
    // findOrFail() throws a ModelNotFoundException (results in 404) if not found
    $product = Product::findOrFail($id);
    return view('products.show', compact('product'));
}

...
```

Eloquent: Basic CRUD Operations

```php
    // UPDATE: Updating an existing product
    public function update(Request $request, string $id) {
        $product = Product::findOrFail($id);
        $validatedData = $request->validate([ /* ... validation rules ... */ ]);

        // $product->update($validatedData); // Mass update fillable attributes

        // OR: Update individual properties
        $product->name = $validatedData['name'];
        // ... update other properties ...
        $product->save(); // Persist changes

        return redirect()->route('products.show', $product->id)
                         ->with('success', 'Product updated successfully!');
    }

    // DELETE: Removing a product
    public function destroy(string $id) {
        $product = Product::findOrFail($id);
        $product->delete();

        return redirect()->route('products.index')
                         ->with('success', 'Product deleted successfully!');
    }
}
```

Advanced Controllers: Resource Controllers

- Quickly scaffold controllers for typical CRUD operations.
- Pairs well with resourceful routing.

1. Create Resource Controller:

```bash
php artisan make:controller PhotoController --resource
# For API (omits create/edit methods returning views)
php artisan make:controller Api/PhotoController --resource --api
```

2. Define Resource Route:

```php
// In routes/web.php or routes/api.php
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
// For API routes, often better:
// Route::apiResource('photos', Api\PhotoController::class);
```

- This single line defines routes for `index`, `create`, `store`, `show`, `edit`, `update`, `destroy`.
- Run `php artisan route:list` to see them.

Advanced Controllers: Resource Controllers

Dependency Injection

- Laravel automatically resolves dependencies type-hinted in controller methods (and constructors).
- Makes code more testable and decoupled.

```php
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request; // Injected Request object
use App\Models\Product;      // Injected Model (Route Model Binding)
use App\Services\ReportingService; // Example custom service (must be registered)

class ProductController extends Controller
{
    // Inject service via constructor
    protected $reportingService;
    public function __construct(ReportingService $service) {
        $this->reportingService = $service;
    }

    // Inject Request and use Route Model Binding
    public function update(Request $request, Product $product) {
```

Controllers: Request Validation

Essential for security and data integrity.

Option 1: Validate in Controller (`$request->validate()`)

- Simple, good for basic validation.
- If validation fails, automatically throws `ValidationException` and redirects back (web) or returns JSON error (API).

php

```php
public function store(Request $request)
{
    $validated = $request->validate([
        'name' => 'required|unique:products|max:255', // Must exist, unique in products table, max 255 chars
        'description' => 'nullable|string',
        'price' => ['required', 'numeric', 'min:0.01'], // Array syntax for rules
        'category_id' => 'required|exists:categories,id', // Must exist in categories table's id column
        'image' => 'nullable|image|mimes:jpeg,png,jpg,gif|max:2048', // Image file validation
        'tags' => 'nullable|array', // Expect an array
        'tags.*' => 'integer|exists:tags,id' // Validate each item in the tags array
    ]);

    // $validated contains only the validated data
    $product = Product::create($validated);
```

Controllers: Returning Responses

Controllers must return a response.

Common Response Types:

- **Views:** Render Blade templates.

```php
return view('products.index', compact('products'));
// Pass data to the view
```

- **JSON:** For APIs.

```php
return response()->json(['data' => $products, 'status' => 'success']);
// Customize status code
return response()->json(['message' => 'Product not found'], 404);
// Eloquent Collections/Models automatically convert to JSON
return Product::findOrFail($id);
```

- **Redirects:** Send user to a different URL.

```php
// Redirect to a specific path
return redirect('/dashboard');
// Redirect to a named route
return redirect()->route('products.show', $product->id);
// Redirect back to the previous page (often after form submission)
return back()->withInput(); // Send old input back
```

1. Create Model & Migration

```bash
php artisan make:model Product -m
```

2. Define Migration

(`database/migrations/...create_products_table.php`)

```php
// ... inside up() method ...
Schema::create('products', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->text('description')->nullable();
    $table->decimal('price', 8, 2)->default(0.00);
    $table->integer('stock')->default(0);
    $table->timestamps();
});
```

3. Run Migration

```bash
# Make sure .env is configured!
php artisan migrate
```

4. Configure Model (`app/Models/Product.php`)

5. Update Controller

(`app/Http/Controllers/ProductController.php`)

```php
<?php
namespace App\Http\Controllers;
use App\Models\Product; // Use the actual model
use Illuminate\Http\Request;
use Illuminate\View\View;
use Illuminate\Http\RedirectResponse; // For redirects

class ProductController extends Controller {
    // Use Eloquent to get products
    public function index(): View {
        $products = Product::orderBy('name')->get(); // Get from
        return view('products.index', compact('products'));
    }

    // Use Route Model Binding & Eloquent
    public function show(Product $product): View {
        // $product is automatically fetched by Laravel
        return view('products.show', compact('product'));
    }

    // Add a create form view route (if needed)
    public function create(): View {
        return view('products.create');
    }
```

1. Create Model & Migration

```bash
php artisan make:model Product -m
```

2. Define Migration

(`database/migrations/...create_products_table.php`)

```php
// ... inside up() method ...
Schema::create('products', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->text('description')->nullable();
    $table->decimal('price', 8, 2)->default(0.00);
    $table->integer('stock')->default(0);
    $table->timestamps();
});
```

3. Run Migration

```bash
# Make sure .env is configured!
php artisan migrate
```

4. Configure Model (`app/Models/Product.php`)

5. Update Controller

(`app/Http/Controllers/ProductController.php`)

```php
<?php
namespace App\Http\Controllers;
use App\Models\Product; // Use the actual model
use Illuminate\Http\Request;
use Illuminate\View\View;
use Illuminate\Http\RedirectResponse; // For redirects

class ProductController extends Controller {
    // Use Eloquent to get products
    public function index(): View {
        $products = Product::orderBy('name')->get(); // Get from
        return view('products.index', compact('products'));
    }

    // Use Route Model Binding & Eloquent
    public function show(Product $product): View {
        // $product is automatically fetched by Laravel
        return view('products.show', compact('product'));
    }

    // Add a create form view route (if needed)
    public function create(): View {
        return view('products.create');
    }
```

Putting It All Together (Cont.)

6. Update Routes (`routes/web.php`)

```php
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\ProductController;

Route::get('/', function () { return view('welcome'); });

// Use resource routing for simplicity
Route::resource('products', ProductController::class);
// This defines routes for index, create, store, show, edit, update, destroy

// If not using resource routing, ensure routes match controller methods:
// Route::get('/products', [ProductController::class, 'index'])->name('products.index');
// Route::get('/products/create', [ProductController::class, 'create'])->name('products.create');
// Route::post('/products', [ProductController::class, 'store'])->name('products.store');
// Route::get('/products/{product}', [ProductController::class, 'show'])->name('products.show'); // Uses Route Model Binding
// Route::get('/products/{product}/edit', [ProductController::class, 'edit'])->name('products.edit');
// Route::put('/products/{product}', [ProductController::class, 'update'])->name('products.update');
// Route::delete('/products/{product}', [ProductController::class, 'destroy'])->name('products.destroy');
```

7. Update Views (e.g., `resources/views/products/index.blade.php`)

```
@extends('layouts.app')
@section('title', 'Products')
```

Summary

- **Blade:** Powerful templating engine (`{{ }}`, `@foreach`, layouts, components `<x-name>`).
- **Migrations:** Version control for database schema (`php artisan make:migration`, `migrate`, `rollback`).
- **Eloquent:** ORM for easy database interaction (`php artisan make:model`, `::all()`, `::find()`, `::create()`, `->save()`, `->delete()`). Remember Mass Assignment (`$fillable` / `$guarded`).
- **Controllers:** Handle logic, use Dependency Injection, validate requests (`$request->validate()` or Form Requests), return responses (`view()`, `redirect()`, `response()`).
- **Resource Routing:** Simplifies CRUD route definitions (`Route::resource()`).

Next Steps

- **Eloquent Relationships:** One-to-One, One-to-Many, Many-to-Many.
- **Database Seeding & Factories:** Populating your database with test data.
- **Middleware:** Filtering HTTP requests (Authentication, Logging).
- **Authentication & Authorization:** User Login, Gates & Policies.
- **Testing:** Writing Unit and Feature tests (PHPUnit/PEST).
- **Queues & Jobs:** Offloading time-consuming tasks.
- **Events & Listeners:** Decoupling parts of your application.

Next Steps

- **Eloquent Relationships:** One-to-One, One-to-Many, Many-to-Many.
- **Database Seeding & Factories:** Populating your database with test data.
- **Middleware:** Filtering HTTP requests (Authentication, Logging).
- **Authentication & Authorization:** User Login, Gates & Policies.
- **Testing:** Writing Unit and Feature tests (PHPUnit/PEST).
- **Queues & Jobs:** Offloading time-consuming tasks.
- **Events & Listeners:** Decoupling parts of your application.

Questions?

- **Object-Relational Mapping**
- A technique that lets you query and manipulate data from a database using an object-oriented paradigm.
- Essentially, it translates data between incompatible type systems in relational databases and object-oriented programming languages.

````
text
<CodeBlockWrapper v-bind="{}" :ranges='[]'>

```
Advantages:
*   Increased productivity
*   Improved code readability
*   Reduced code duplication
*   Easier maintenance
```

</CodeBlockWrapper>
````

What is an ORM?

Eloquent: Laravel’s ORM

- Eloquent is Laravel’s ORM.
- Provides a simple ActiveRecord implementation for working with your database.
- Each database table has a corresponding "Model" that is used to interact with that table.

```php
// Example: Retrieving all users
use App\Models\User;

$users = User::all();

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

Database Configuration

- Laravel's database configuration is located in the `.env` file and `config/database.php`.
- Make sure your database connection details are correctly configured before using Eloquent.

```text
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=your_database
DB_USERNAME=your_username
DB_PASSWORD=your_password
```

- Models typically live in the `app/Models` directory.
- You can generate a model using the `make:model` Artisan command.
- By convention, model names are singular and table names are plural.

```bash
php artisan make:model User
```

```php
// Example: App\Models\User.php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    // Table name (optional)
    // protected $table = 'users';

    // Primary key (optional)
    // protected $primaryKey = 'id';
}
```

Defining Eloquent Models

- **Create (Insert):** Create new records in the database.
- **Read (Select):** Retrieve records from the database.
- **Update:** Modify existing records in the database.
- **Delete:** Remove records from the database.

```php
// Create
$user = new User;
$user->name = 'John Doe';
$user->email = 'john.doe@example.com';
$user->password = bcrypt('secret');
$user->save();

// Read
$user = User::find(1); // Find by primary key
echo $user->name;

// Update
$user->email = 'new.email@example.com';
$user->save();

// Delete
$user->delete();
```

Basic CRUD Operations

Eloquent Relationships

- Eloquent supports various types of relationships:
  - One-to-One
  - One-to-Many
  - Many-to-Many
  - Has One Through
  - Has Many Through
  - Polymorphic Relationships
- Relationships are defined as methods on your Eloquent model classes.

```php
// Example: One-to-Many (User has many Posts)
// In App\Models\User.php
public function posts()
{
    return $this->hasMany(Post::class);
}

// Accessing the relationship
$user = User::find(1);
foreach ($user->posts as $post) {
    echo $post->title;
}
```

Eloquent Collections

- Eloquent's `all()` and `get()` methods return instances of `Illuminate\Database\Eloquent\Collection`.
- Provides a variety of helpful methods for working with your data.

```php
// Example: Using collections
$users = User::all();

// Filtering
$activeUsers = $users->filter(function ($user) {
    return $user->is_active;
});

// Mapping
$emails = $users->map(function ($user) {
    return $user->email;
});
```

Eloquent Accessors & Mutators

- **Accessors:** Transform Eloquent attributes values when you retrieve them.
- **Mutators:** Transform Eloquent attribute values when you set them.

```php
//Example: Accessor and Mutator
//In App\Models\User.php
public function getNameAttribute($value)
{
    return ucfirst($value);
}

public function setPasswordAttribute($value)
{
    $this->attributes['password'] = bcrypt($value);
}

// Usage
$user = User::find(1);
echo $user->name; // Output: John Doe (if name was john doe in th

$user->password = 'new_password'; // Password will be hashed
$user->save();
```

- Allow you to define common sets of constraints that you can easily re-use throughout your application.
- Scopes allow you to keep your models clean and DRY (Don’t Repeat Yourself).

```php
// Example: Query Scope
// In App\Models\User.php
public function scopeActive($query)
{
    return $query->where('is_active', true);
}

// Usage
$activeUsers = User::active()->get();
```

Eloquent Query Scopes

Mass Assignment

- Mass assignment allows you to create or update a model using an array of attributes.
- For security, Eloquent protects against mass-assignment vulnerabilities by default.
- You must define either a `$fillable` or `$guarded` property on your model.
  - `$fillable` : An array of attributes that are mass assignable.
  - `$guarded` : An array of attributes that are not mass assignable. (Use an empty array `[]` to make all attributes mass assignable - use with caution).

```php
// In App\Models\User.php
class User extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name', 'email', 'password'];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    // protected $guarded = ['is_admin'];
}

// Usage
$user = User::create([
    'name' => 'Jane Doe',
    'email' => 'jane@example.com',
    'password' => bcrypt('password'),
]);
```

- **Retrieving Multiple Models:**
  - `where()`, `orWhere()`, `orderBy()`, `limit()`, `offset()`
  - `latest()`, `oldest()`
- **Chunking Results:**
  - `chunk()` and `lazy()` for processing large datasets.
- **Aggregates:**
  - `count()`, `sum()`, `avg()`, `max()`, `min()`
- **"Or Fail" Methods:**
  - `findOrFail()`, `firstOrFail()`: Throw an exception if no model is found.

```php
// Get users with more than 100 votes, ordered by name
$users = User::where('votes', '>', 100)
              ->orderBy('name', 'desc')
              ->get();

// Get the first user with more than 100 votes
$user = User::where('votes', '>', 100)->first();

// Find a user by ID or throw an exception
$user = User::findOrFail(1);

// Count active users
$count = User::where('active', 1)->count();

// Process users in chunks
User::chunk(100, function ($users) {
    foreach ($users as $user) {
        // Process the user
    }
});
```

Advanced Retrieval

- `create()` **method:** (Requires mass assignment to be configured)
- `update()` **method:** (Requires mass assignment to be configured)
- `firstOrCreate()` / `firstOrNew()` :
  - `firstOrCreate` : Attempt to find a model matching attributes, or create it if not found.
  - `firstOrNew` : Attempt to find a model, or instantiate a new model instance (without persisting).
- `updateOrCreate()` :
  - Attempt to find a model matching attributes, then update it. If not found, create it.
- `upsert()` :
  - Perform multiple "upserts" in a single query (insert or update if exists).

```php
// Create (mass assignment)
$user = User::create(['name' => 'New User', 'email' => 'new@examp

// Update (mass assignment)
$user = User::find(1);
$user->update(['name' => 'Updated Name']);

// Find user by email, or create if not exists
$user = User::firstOrCreate(
    ['email' => 'unique@example.com'],
    ['name' => 'Unique User', 'password' => bcrypt('secret')]
);

// Find user by email, update if exists, or create if not
$user = User::updateOrCreate(
    ['email' => 'another@example.com'],
    ['name' => 'Another User', 'votes' => 10]
);
```

Soft Deleting

- Instead of actually removing records from your database, soft deleting "marks" them as deleted.
- To enable soft deletes, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to your model and add the `deleted_at` column to your table.
- Now, when you call the `delete` method, the `deleted_at` column will be set.
- Queries will automatically exclude soft-deleted models.

```php
// In App\Models\Post.php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Post extends Model
{
    use SoftDeletes; // Enable soft deletes

    // The 'deleted_at' column will be automatically managed
}

// Deleting a model
$post = Post::find(1);
$post->delete(); // Sets deleted_at timestamp

// Querying with soft deleted models
$posts = Post::withTrashed()->get(); // Include soft-deleted
$trashedPosts = Post::onlyTrashed()->get(); // Only soft-deleted

// Restoring a soft-deleted model
$post->restore();

// Permanently deleting
$post->forceDelete();
```

Eager Loading (N+1 Problem)

- When accessing Eloquent relationships as properties, the relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property.
- This can lead to the "N+1 query problem" if you load a parent model and then loop through its children, executing one query for the parent and N additional queries for each child.
- **Eager loading** loads the relationship data at the same time as the parent model.

```php
// N+1 Problem:
// $books = App\Models\Book::all(); // 1 query
// foreach ($books as $book) {
//     echo $book->author->name; // N queries (one for each autho
// }

// Solution: Eager Loading with `with()`
$books = App\Models\Book::with('author')->get(); // 2 queries

foreach ($books as $book) {
    echo $book->author->name; // No additional queries
}

// Eager load multiple relationships
$books = App\Models\Book::with(['author', 'publisher'])->get();

// Eager load nested relationships
$books = App\Models\Book::with('author.contacts')->get();
```

- Eloquent models fire several events, allowing you to hook into various points in the model's lifecycle:
  - `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleted`, `restoring`, `restored`, `replicating`.
- Events allow you to easily execute code each time a specific model class is saved or updated in the database.
- You can define event listeners in your model's `booted` method or use dedicated observer classes.

```php
// In App\Models\User.php
use Illuminate\Support\Str;

class User extends Model
{
    protected static function booted()
    {
        static::creating(function ($user) {
            $user->uuid = (string) Str::uuid(); // Generate UUID
        });

        static::deleted(function ($user) {
            // Logic after a user is deleted
            Log::info("User {$user->id} deleted.");
        });
    }
}

// Using Observers (php artisan make:observer UserObserver --mode
// In App\Observers\UserObserver.php
class UserObserver
{
    public function created(User $user): void
    {
        // ...
    }
}
// Register observer in AppServiceProvide
```