CSRF Protection

Cross-Site Request Forgery (CSRF) is an attack that tricks the user into submitting a malicious request. Laravel provides built-in CSRF protection to prevent such attacks. Here's what you need to know about CSRF protection:

- **Definition**: CSRF protection in Laravel involves adding a CSRF token to each form and verifying it on form submission, ensuring that the request originated from the same application.
- **Implementation**: Laravel automatically generates a CSRF token for each session. You can include the token in your forms using the @csrf Blade directive, which adds a hidden input field with the token value.

```
<form method="POST" action="/submit-form">
    @csrf
    <!-- Rest of the form fields -->
</form>
```

Creating Forms

- **Form Creation**: Use Blade templates to create HTML forms.
- **Binding Data**: Easily bind form inputs to model data for editing.
- The **`old`** function repopulates the form with previous input upon validation failure

```
<form method="POST" action="/post/store">
    @csrf
    <label for="title">Title:</label>
    <input type="text" id="title" name="title"
value="{{ old('title') }}">
    <button type="submit">Submit</button>
</form>
```

Validation Techniques

Laravel provides a comprehensive validation system that allows you to validate user input before processing it.

- **Validation Rules**: Laravel offers a wide range of validation rules, such as `required`, `email`, `numeric`, `min`, `max`, `unique`, and many more, to validate different types of data.
- **Form Request Validation**: Laravel's Form Request validation allows you to define validation rules in a separate class, improving code organization and reusability.
- **Error Messages**: Laravel automatically handles error messages for failed validation, allowing you to display appropriate error messages to the user.
- https://laravel.com/docs/10.x/validation

```php
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|email|unique:users',
        'password' => 'required|min:8',
    ]);

    // Process the validated data
}
```

Eloquent ORM

Eloquent is Laravel's built-in Object-Relational Mapping (ORM) system, which provides a convenient and expressive way to interact with databases. Here's what you need to know about Eloquent:

- **Object-Relational Mapping**: Eloquent maps database tables to PHP objects, allowing developers to work with database records as objects, making database interactions more intuitive.
- **Model-View-Controller (MVC) Architecture**: Eloquent follows the MVC architectural pattern, where models represent data structures and handle database interactions, views handle presentation, and controllers handle application logic.
- **Query Building**: Eloquent simplifies the process of querying the database by providing a fluent query builder API, allowing you to chain methods to build complex queries.

Database Migrations

Database migrations are a key component of Laravel and Eloquent. Migrations provide an easy way to manage database schema changes and version control. Here's how migrations work with Eloquent:

- **Migration Files**: Migration files are stored in the `database/migrations` directory. Each migration file contains instructions for creating or modifying database tables.
- **Migration Methods**: Migration files contain `up` and `down` methods. The `up` method defines the changes to be made to the database schema, while the `down` method defines how to revert those changes.
- **Running Migrations**: Migrations can be executed using the `php artisan migrate` command. This command applies any pending migrations, ensuring that your database schema is up to date.
- **Reference:** https://laravel.com/docs/10.x/migrations

Working with Migrations

1 - Creating a Migration:
`php artisan make:migration create_posts_table`

2 - Writing the Migration

```php
public function up(): void
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('content');
        $table->timestamps(); // created_at, updated_at
    });
}

public function down(): void
{
    Schema::dropIfExists('posts');
}
```

3- Running the Migration
`php artisan migrate`

4- Rolling Back
`php artisan migrate:rollback`

Defining Models

Models in Eloquent represent database tables and handle database interactions.

https://laravel.com/docs/10.x/eloquent

- **Model Creation**

  Models are typically stored in the `app/Models` directory. You can create a model using the `php artisan make:model` command.

- **Table Mapping**

  By default, Eloquent assumes that the table name associated with a model is the plural form of the model's name. However, you can specify a different table name by defining the `$table` property in the model.

- **Model Relationships**

  Eloquent provides expressive methods to define relationships between models, such as one-to-one, one-to-many, and many-to-many relationships.

Model Example

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'users';

    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Eloquent ORM Usage

```php
// Retrieving Records
$users = User::where('active', true)->get();

// Creating Records
$user = new User;
$user->name = 'John Doe';
$user->email = 'john@example.com';
$user->save();

// Updating Records
$user = User::find(1);
$user->name = 'Jane Doe';
$user->save();

// Deleting Records
$user = User::find(1);
$user->delete();
```

Database Relationships

- **Eloquent ORM**: Laravel's Eloquent ORM makes it easy to interact with database relationships.
- **Types of Relationships**:
  - **One to One**: Each record in one table is linked to one record in another table.
  - **One to Many**: A single record in one table is associated with multiple records in another table.
  - **Many to Many**: Records in one table are related to multiple records in another table, typically using a pivot table.
- **Defining Relationships**: Relationships are defined as methods in Eloquent model classes.
- **Reference**: https://laravel.com/docs/10.x/eloquent-relationships

One to One Relationship

Scenario: Each User has one Profile.

```php
// User Model
class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class);
    }
}

// Profile Model
class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}

// Accessing the Relationship
$userProfile = User::find(1)->profile;
```

Maharah
Code Your Future

One to Many Relationship

Scenario: A **Post** model, where each **User** can have multiple **Posts**.

```php
// User Model
class User extends Model
{
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}

// Post Model
class Post extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}

// Accessing the Relationship
$userPosts = User::find(1)->posts;
```

Maharah
Code Your Future

Many to Many Relationship

Scenario: **User** and **Role** models, with a **role_user** pivot table.

```php
// User Model
class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}

// Role Model
class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}

// Accessing the Relationship:
$userRoles = User::find(1)->roles;
```

Maharah
Code Your Future

Eloquent Collections

Eloquent Collections in Laravel provide a powerful way to work with sets of Eloquent models and perform various operations on them.

https://laravel.com/docs/10.x/eloquent-collections

- **Definition**

  An Eloquent Collection is an object that wraps an array of Eloquent models, allowing you to perform operations on the entire set of models.

- **Common Operations**

  Eloquent Collections provide a wide range of methods for manipulating and filtering data, such as `map`, `filter`, `pluck`, `groupBy`, `sum`, and many more.

- **Chaining Methods**

  Eloquent Collections allow method chaining, enabling you to combine multiple operations to perform complex transformations on the data.

Eloquent Collections Example

```php
$users = User::where('active', true)->get();

// Perform operations on the collection
$filteredUsers = $users->filter(function ($user) {
    return $user->age > 25;
});

$names = $filteredUsers->pluck('name');

$groupedUsers = $filteredUsers->groupBy('role');

$totalAge = $filteredUsers->sum('age');
```

Advanced Queries

Laravel's Query Builder provides a fluent interface for building complex database queries.

- **Joins**

  The Query Builder allows you to perform various types of joins, such as inner joins, left joins, right joins, and cross joins, to fetch data from multiple tables.

- **Subqueries**

  You can use subqueries in your queries to retrieve data from a subset of records based on certain conditions.

- **Raw Expressions**

  The Query Builder supports raw expressions, which allow you to write raw SQL statements within your queries for advanced operations.

Advanced Queries Example

```
$users = DB::table('users')
    ->join('posts', 'users.id', '=', 'posts.user_id')
    ->select('users.*', 'posts.title')
    ->where('users.active', true)
    ->orWhere('posts.published', true)
    ->orderBy('users.created_at', 'desc')
    ->get();
```

Eager Loading

Eager loading is a technique in Laravel that allows you to load relationships between models upfront, reducing the number of database queries.

- **N+1 Problem**: Without eager loading, accessing related models within a loop can result in the N+1 problem, where each iteration triggers an additional database query, leading to performance issues.
- **Eager Loading with Relationships**: Eloquent provides the `with` method to specify relationships that should be eager loaded, improving performance by fetching related data in a single query.

```php
$users = User::with('posts')->get();

foreach ($users as $user) {
    foreach ($user->posts as $post) {
        // Access the related posts without additional queries
    }
}
```

Exercise: Build Forms and Save to Database

02

Mission

Expand the book rental library application to include functionality for users to view a list of books, and rent a book. This involves setting up a database, creating models and relationships, and handling form data securely.