Pla de Recuperació, Transformació i Resiliència
FPCV Formació Professional Comunitat Valenciana
Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur
GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura, Universitats i Ocupació
IES MARIA ENRIQUEZ

Unitat 5. Frameworks. Laravel

2n DAW - IES María Enríquez

Part 5

Resultats d’aprenentatge i criteris d’avaluació

RA 5. Desenvolupa aplicacions Web identificant i aplicant mecanismes per a separar el codi de presentació de la lògica de negoci.

Criteris d'avaluació:

a) S'han identificat els avantatges de separar la **lògica de negoci** dels aspectes de presentació de l'aplicació.

b) S'han analitzat **tecnologies i mecanismes** que permeten realitzar aquesta separació i les seues característiques principals.

c) S'han utilitzat **objectes i controls en el servidor** per a generar l'aspecte visual de l'aplicació web en el client.

d) S'han utilitzat **formularis generats de manera** dinàmica per a respondre als esdeveniments de l'aplicació Web.

e) S'han identificat i aplicat els **paràmetres relatius a la configuració** de l'aplicació Web.

f) S'han escrit aplicacions Web amb **manteniment d'estat i separació de la lògica** de negoci.

g) S'han aplicat els principis de la programació **orientada a objectes**.

h) S'ha provat i **documentat** el codi.

Resultats d'aprenentatge i criteris d'avaluació

RA8. Genera pàgines web dinàmiques analitzant i utilitzant tecnologies i frameworks del servidor web que afigen codi al llenguatge de marques.

Criteris d'avaluació:

a) S'han identificat les diferències entre l'**execució de codi en el servidor i en el client web**.

b) S'han reconegut els avantatges d'**unir totes dues tecnologies** en el procés de desenvolupament de programes.

c) S'han identificat les **tecnologies i frameworks relacionades amb la generació per part del servidor** de pàgines web amb guions embeguts.

d) S'han utilitzat aquestes **tecnologies i frameworks per a generar pàgines web** que incloguen **interacció** amb **l'usuari**.

e) S'han utilitzat aquestes tecnologies i frameworks, per a generar pàgines web que incloguen **verificació** de formularis.

f) S'han utilitzat aquestes tecnologies i frameworks per a generar pàgines web que incloguen **modificació dinàmica del seu contingut** i la seua estructura.

g) S'han aplicat aquestes tecnologies i frameworks en la **programació d'aplicacions web**.

Unitat 5. Frameworks. Laravel

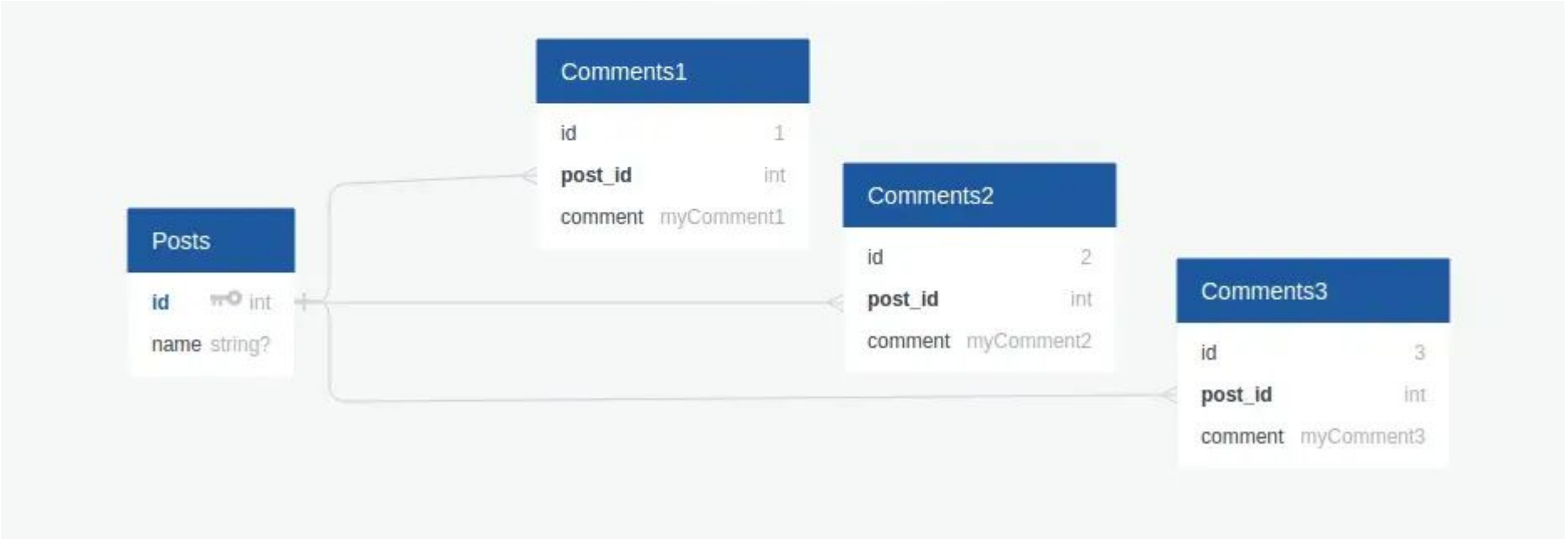
4. El model de dades
    - 4.4. Relacions entre models
    - 4.5. Seeders i factories
    - 4.6. Query Builders i ús de dates

5. El model de dades II

Relacions entre models

Eloquent permet definir relacions entre diferents tipus de taules.

Per exemple, entre pel·lícules i actors o post i comentaris.



Relacions un a un (one to one)

Imaginem els models `Usuario` i `Telefono` amb una relació un a un.

Al ser una relació 1:1, la clau d'una de les taules passa a l'altra. Per exemple, la taula `usuaris` té el camp `telefono_id`. És important que el camp tinga el nom `telefono_id`.

Per indicar-ho, afegim un mètode al model `Usuario` amb el mateix nom que el model que volem connectar, `telefono`. Dins utilitzarem el mètode `hasOne` per indicar que té un objecte de tipus `Telefono`.

```
class Usuario extends Model
{
    public function telefono()
    {
        return $this->hasOne(Telefono::class);
    }
}
```

Relacions un a un (one to one)

Guardar dades relacionades

```
$telefono = Telefono::findOrFail($idTelefono);
$usuario = new Usuario();
$usuario->nombre = "Pepe";
$usuario->email = "pepe@gmail.com";
$usuario->telefono_id = $telefono->id;
$usuario->save();
```

Relacions un a un (one to one)

Guardar dades relacionades

També és possible associar els dos objectes amb el mètode `associate`

```
$telefono = Telefono::findOrFail($idTelefono);
$usuario = new Usuario();
$usuario->nombre = "Pepe";
$usuario->email = "pepe@gmail.com";
$usuario->telefono()->associate($telefono);
$usuario->save();
```

Relacions un a molts (one to many)

Imaginem que tenim la relació `Autor` i `Libro`.

En aquest cas afegiríem l'id de l'autor al llibre.

Per implementar-ho, utilitzarem el mètode `hasMany()`.

```
class Autor extends Model
{
    public function libros()
    {
        return $this->hasMany(Libro::class);
    }
}
```

Relacions un a molts (one to many)

De la mateixa forma que abans, assumim que la taula libros té un camp id i que la clau aliena cap a autores és autor_id.

En cas contrari, haurem de passar més paràmetres en hasMany:

```
class Autor extends Model
{
    public function libros()
    {
        return $this->hasMany(Libro::class, 'id_autor');
    }
}
```

Relacions un a molts (one to many)

Per obtenir els llibres associats a un autor:

```
$libros = Autor::findOrFail($id)->libros();
```

Relacions un a molts (one to many)

També podem far el cas invers, a partir d'un llibre obtenir l'autor. Per fer-ho necessitem el mètode `belongsTo`:

```
class Libro extends Model
{
    public function autor()
    {
        return $this->belongsTo(Autor::class);
    }
}
```

Per obtenir el l'autor a partir del llibre:

```
$nombreAutor = Libro::findOrFail($id)->autor->nombre;
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

Afegim el camp autor_id en la taula libros

```
php artisan make:migration nuevo_campo_autor_libros --table=libros
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

```
class NuevoCampoAutorLibros extends Migration
{
    public function up()
    {
        Schema::table('libros', function(Blueprint $table) {
            $table->integer('autor_id');
        });
    }

    public function down()
    {
        Schema::table('libros', function(Blueprint $table) {
            $table->dropColumn('autor_id');
        });
    }
}
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

```
php artisan migrate
```

Creem el model, la migració i el controlador d'autors:

```
php artisan make:model Autor -mcr
```

Haurem de renombrar la migració a ma, perque per defecte Laravel crearà la taula **autors** i no **autores**. Caldra canviar tant el nom de la migració, com el nom de la taula en els mètodes up i down.

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

El mètode up quedaria així:

```
return new class extends Migration
{
    public function up()
    {
        Schema::create('autores', function(Blueprint $table) {
            $table->id();
            $table->string('nombre');
            $table->integer('nacimiento')->nullable();
            $table->timestamps();
        });
    }
}
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

A continuació:

```
 php artisan migrate
```

Modifiquem el model Autor:

```
class Autor extends Model
{
    protected $table = 'autores';
    ...

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

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

Afegim al model Libro el mètode `belongTo` per recuperar l'Autor

```
class Libro extends Model
{
    ...

    public function autor()
    {
        return $this->belongsTo(Autor::class);
    }
}
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

Per provar-ho anem a crear un llibre associat a l'autor 1. Definim la ruta i incorporem amb `use` els models `Autor` i `Libro`

```
Route::get('relacionPrueba', function() {
    $autor = Autor::findOrFail(1);
    $libro = new Libro();
    $libro->titulo = "Libro de prueba " . rand();
    $libro->editorial = "Editorial de prueba";
    $libro->precio = 5;
    $libro->autor()->associate($autor);
    $libro->save();

    return redirect()->route('libros.index');
});
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

Ara modifiquem la vista `libros/index.blade.php`

```
@forelse($libros as $libro)
    <li><a href="{{ route('libros.show', $libro) }}">
        {{ $libro->titulo }} ({{ $libro->autor->nombre }})
    </a></li>
@empty
    <li>No se encontraron libros</li>
@endforelse
```

Relacions un a molts (one to many)

Aplicació a l'exemple Autor - Libro

Provem el resultant accedint a:

```
http://localhost:8000/relacionPrueba
http://localhost:8000/libros
```

Relacions un a molts (one to many)

Accés eficient

En el exemple anterior, si obtenim una consulta amb 100 llibres, farem 100 consultes associades per obtenir l'autor.

Una forma de disminuir aquesta sobrecàrrega és utilitzar la tècnica ***eagler loading*** (càrrega anticipada).

Consisteix en utilitzar el mètode `with` per indicar la relació que volem deixar pre-carregada.

```
public function index()
{
    $libros = Libro::with('autor')->get();
    return view('libros.index', compact('libros'));
}
```

Relacions molts a molts (many to many)

Per fer aquest tipus de relacions, necessitem una tercera taula.

Imaginem la relació entre `Usuario` i `Rol`.

Definim un mètode en `Usuario` que utilitze el `belongsToMany` per indicar amb quin model es relaciona.

```
class Usuario extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Rol::class);
    }
}
```

Relacions molts a molts (many to many)

Per accedir als rols fem:

```
$roles = Usuario::findOrFail($id)->roles;
```

A l'altre costat tindríem:

```
class Rol extends Model
{
    public function usuarios()
    {
        return $this->belongsToMany(Usuario::class);
    }
}
```

Relacions molts a molts (many to many)

Per a que Eloquent establisca relacions automàtiques entre dos taules A i B, assumeix que hi ha una taula intermitja a_b amb els camps a_id i b_id.

En el nostre cas existeix una taula rol_usuari amb un camp rol_id i usuari_id.

Si volem accedir a altres camps de la taula intermediària farem ús de l'atribut `pivot`

```
$roles = Usuario::findOrFail($id)->roles;

for($roles as $rol)
{
    echo $rol->pivot->created_at;
}
```

Documentació

Es recomana molt la lectura de la documentació oficial per veure tot el potencial d'Eloquent i les relacions:

- Documentació oficial: eloquent-relationships
- En castellà

Seeders i factories

Introducció

Quan comencem una aplicació, necessitem **afegir dades** per poder realitzar proves.

A més, la part dels **formularis** sol deixar-se per al final.

Per solventar aquests problemes farem ús dels ***seeders i factories*** que ens permetran afegir dades de prova d'una manera senzilla.

Seeders

- Permeten "*sembrar*" contingut a l'aplicació.
- Per fer-ho executarem:

```
php artisan make:seeder NombreSeeder
```

- Açò crearà una classe amb el nom `NombreSeeder` en `database`/`seeders`.
- En el mètode `run` afegirem les dades que volem afegir a la base de dades.

Seeders

- En el nostre exemple de la biblioteca crearem:

```
php artisan make:seeder LibrosSeeder
php artisan make:seeder AutoresSeeder
```

-

Seeders

- En el mètode `run` de llibres, definim l'autor amb el llibre associat. Caldrà afegir prèviament `use` del model `Autor` i `Libro`.

```
public function run()
{
    $autor = new Autor();
    $autor->nombre = "Juan Seeder";
    $autor->nacimiento = 1960;
    $autor->save();
    $libro = new Libro();
    $libro->titulo = "El libro del Seeder";
    $libro->editorial = "Seeder S.A.";
    $libro->precio = 10;
    $libro->autor()->associate($autor);
    $libro->save();
}
```

Seeders

- En el mètode `run` d'autor, afegim un autor solt

```
public function run()
{
    $autor = new Autor();
    $autor->nombre = "Autor Suelto";
    $autor->nacimiento = 1951;
    $autor->save();
}
```

Seeders i factories

Afegint els seeders a l'aplicació

Ara, cal carregar aquests seeders en l'aplicació, per fer-ho, els donarem d'alta en el seeder general, `DatabaseSeeder`. És important l'ordre en el que afegim els seeders.

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        ...
        $this->call(AutoresSeeder::class);
        $this->call(LibrosSeeder::class);
    }
}
```

Seeders i factories

Llançar els seeders

Si volem llançar tots els seeders definits en `DatabaseSeeder`:

```
php artisan db:seed
```

Si només volem executar un en concret:

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

També pot ser necessari i convenient netejar la base de dades abans:

```
php artisan migrate:fresh --seed
```

Seeders i factories

Els factories

El seeders estan bé per afegir unes **poques dades**, però si volem una major quantitat de dades es queden un poc fluixos.

Per solventar aquest problema estan els factories que permeten afegir **dades per lots**.

Per crear-los utilitzem el següent comandament:

```
php artisan make:factory NombreFactory --model=NombreModelo
```

Seeders i factories

Generar factories i associar-los a un model

Quan vam crear els models, la classe es crea amb un use al *trait* `hasFactory`.

Un trait és bàsicament un conjunt de mètodes que es poden utilitzar des de qualsevol classe. És similar a poder heretar de diferents classes.

```
php artisan make:factory NombreFactory --model=NombreModelo
```

Seeders i factories

Generar factories i associar-los a un model

Anem a crear un factory per al model del nostre exemple `Libro` i `Autor`

```
php artisan make:factory AutorFactory --model=Autor
php artisan make:factory LibroFactory --model=Libro
```

Les classes generades seran així:

```
class AutorFactory extends Factory
{
    protected $model = Autor::class;

    ...
}
```

Seeders i factories

Generar factories i associar-los a un model

El mètode `definition` és el que es va executar quan utilitzem el factory per generar les dades del mètode associat..

Per exemple, podem retornar dades manualment generades:

```
public function definition()
{
    return [
        'nombre' => "Autor " . rand(1, 100),
        'nacimiento' => rand(1950, 1990)
    ];
}
```

Seeders i factories

Generar factories i associar-los a un model

Per crear autors des del seeder utilitzant el factory, en el seu mètode `run`:

```
class AutoresSeeder extends Seeder
{
    public function run()
    {
        return Autor::factory()->count(5)->create();
    }
}
```

Seeders i factories

Els fakers

Ara conseguim crear dades, però no són molt **reals**: Autor 1, Autor 2, ...

Laravel ens proporciona els ***fakers*** per generar dades a l'atzar. Així podem crear noms reals aleatoris, correus electrònics, frases, ...

Per fer-ho, utilitzarem la classe `Faker` que està incorporat en el factory, fent ús de la propietat `$this->faker`.

Algunes de les dades que podem generar són:

Seeders i factories

Els fakers

- `name`: admet el paràmetre opcional “male” o “female”
- `sentence`: genera una frase curta. Admet paràmetre número de paraules.
- `word`: genera una paraula aleatòria.
- `text`: genera un text llarg.
- `phoneNumber`: genera un telèfon.
- `email:` genera un e-mail aleatori.
- `randomNumber`: genera un número aleatori. Té `numberBetween`.

Més info

Seeders i factories

Els fakers

També té el mètode `unique()` que assegura que un camp no aparega repetit.

Anem a generar dades d'autor utilitzant el faker en el mètode `definition` del AutorFactory:

```
public function definition()
{
    return [
        'nombre' => $this->faker->name,
        'nacimiento' => $this->faker->numberBetween(1950, 1990)
    ];
}
```

Seeders i factories

Els fakers

En el LibroFactory

```
public function definition()
{
    return [
        'titulo' => $this->faker->sentence,
        'editorial' => $this->faker->sentence(2),
        'precio' => $this->faker->randomFloat(2, 5, 20)
    ];
}
```

Seeders i factories

Relacionant els models

En els seeders corresponents generem els autors

```
class AutoresSeeder extends Seeder
{
    public function run()
    {
        Autor::factory()->count(5)->create();
    }
}
```

Seeders i factories

Relacionant els models

Recorreguem els autors i generem dos llibres per cadascú d'ells:

```
class LibrosSeeder extends Seeder
{
    public function run()
    {
        $autores = Autor::all();
        $autores->each(function($autor) {
            Libro::factory()->count(2)->create([
                'autor_id' => $autor->id
            ]);
        });
    }
}
```

Seeders i factories

Important

Abans hem comentat que en el `DataSeeders` és important l'ordre.

Primer hem de crear els autors i després els llibres per poder crear els llibres associats als autors.

Query builders i ús de dates

A continuació veurem un parell d'eines que poden ser útils:

- **Query builder** per atacar la base de dades. És una alternativa a Eloquent.
- Com treballar amb **dades**

Query builders i ús de dates

Consultes

Per fer-les utilitzem la classe `DB`. Ubicat en `Illuminate\Support\Facades\DB`.

Internament utilitzarem el mètode `table` per seleccionar la taula i `get` per obtenir tots els registres:

```
use Illuminate\Support\Facades\DB;

...

$personas = DB::table('personas')->get();
```

Ens retornarà un array d'objectes, no un array associatiu.

Query builders i ús de dates

Consultes

Si volem recuperar un objecte pel seu id, ho farem de la següent forma:

```
$persona = DB::table('personas')->where('id', $id)->first();
```

Query builders i ús de dates

Actualitzacions

Per insertar un nou registre, utilitzem `insert`:

```
DB::table('personas')->insert([
    'nombre' => 'Juan',
    'edad' => 56
]);
```

Per modificar utilitzarem els mètodes `where` i `update`:

```
DB::table('personas')->where('id', $id)->update([
    'nombre' => 'Juan',
    'edad' => 56
]);
```

Query builders i ús de dates

Actualitzacions

Per eliminar, utilizarem `delete`

```
DB::table('personas')->where('id', $id)->delete();
```

Query builders i ús de dates

Ús de dates

En algunes ocasions utilitzem els camps *timestamp*.

Aquests camps són instàncies d'una llibreria de PHP amb el nom `Carbon`.

Un exemple d'ús seria:

```
<p>
Fecha creación:
{{ Carbon\Carbon::parse($persona->created_at)->format('d/m/Y') }}
</p>
```

Els camps `created_at` i `updated_at` es creen per defecte al fer les migracions.

Exercicis

Exercici 1

- Crea una relació un a molts entre el model d'`Usuari` i el model de `Post`, tots dos ja existents en l'aplicació, de manera que un post és d'un usuari, i un usuari pot tindre molts posts. Hauràs de definir una nova migració de modificació sobre la taula `posts` que afija un nou camp `usuari_id`, i establir a partir d'ell la relació, com hem fet en l'exemple amb autors i llibres.
- Crea des de phpMyAdmin una sèrie d'usuaris de prova en la taula usuaris, i associa alguns d'ells als posts que hi haja.
- Modifica la vista `posts/index.blade.php` perquè, al costat del títol de cada post, entre parèntesi, aparega el login de l'usuari que el va crear.

Exercicis

Exercici 2

- Crea un seeder anomenat `UsuariosSeeder`, amb un factory associat anomenat `UsuarioFactory` (canvia de nom el que ve per defecte `UserFactory` per a aprofitar-ho). Crea amb això 3 usuaris de prova, amb logins que siguen únics i d'una sola paraula (usa el faker), i passwords també d'una sola paraula, sense encriptar (per a poder-los identificar després, arribat el cas).
- Crea un altre seeder anomenat `PostsSeeder` amb un factory associat anomenat `PostFactory`. En el factory, defineix amb el faker títols aleatoris (frases) i continguts aleatoris (textos llargs). Usa el seeder per a crear 3 posts per a cadascun dels usuaris existents.

Exercicis

Exercici 2

- Utilitza l'opció `php artisan migrate:fresh --seed` per a esborrar tot contingut previ i poblar la base de dades amb aquests nous elements. Comprova després des de la pàgina del llistat de posts, i des de phpMyAdmin, que la informació que apareix és correcta.

Exercicis

Exercici 3

- Afig al projecte blog un nou model anomenat `Comentari`, juntament amb la seua migració i controlador associats. Cada comentari tindrà com a camp el contingut del comentari, i estarà relacionat un a molts amb el model `Usuari`, de manera que un usuari pot tindre molts comentaris, i cada comentari pertany a un usuari. També tindrà una relació un a molts amb el model `Post`, de manera que un comentari pertany a un post, i un post pot tindre molts comentaris. Per tant, la migració dels comentaris haurà de tindre com a camps addicionals la relació amb l'usuari (`usuari_id`) i amb el post al qual pertany (`post_id`).

Exercicis

Exercici 3

- Aplica la migració per a reflectir la nova taula en la base de dades, i utilitza un seeder i un factory per a crear 3 comentaris en cada post, amb l'usuari que siga. A l'hora d'aplicar tot això, esborra els continguts previs de la base de dades (`migrate:fresh --seed`).
- AJUDA: si vols triar un usuari a l'atzar com a autor de cada comentari, pots fer una cosa així:
  - `Usuari::inRandomOrder()->first();`
- En aquest cas, seria convenient que eixe usuari aleatori s'afija directament en el factory del comentari, i no en el seeder, ja que en cas contrari és possible que genere el mateix usuari per a tots els comentaris d'un post.

Exercicis

Exercici 3

- En la fitxa dels posts (vista `posts/show.blade.php`), afig el codi necessari per a mostrar el login de l'usuari que ha fet el post, i el llistat de comentaris associat al post, mostrant per a cadascun el login de l'usuari que el va fer, i el text del comentari en sí. Utilitza també la llibreria `Carbon` per a mostrar la data de creació del post (o la dels comentaris, com preferisques) en format `d/m/I`.

Atribucions

Curs de Nacho Iborra