Creación de una Red Social con Laravel y TDD

Autor: Angel Cánovas

Asignatura: Desarrollo Web en Entorno Servidor

Curso online realizado:
https://aprendible.com/series/red-social-laravel-tdd

Página para acceder a la aplicación:
http://social.local/

Github del proyecto:
https://github.com/AngelCanovas/social

Leyenda para el código

Color:	Utilidad:
verde	Nuevo
magenta	Modificación
amarillo	Reestructuración
azul	Explicación / Comentario
naranja	Aviso
rojo	Importante / Fallos

-> Guia para instalar el proyecto (Windows) <-

Instalamos los siguientes programas:

- VirtualBox: https://www.virtualbox.org/wiki/Downloads
- Chrome: https://www.google.com/intl/es_es/chrome/
- Git: https://git-scm.com/downloads
- Vagrant: https://www.vagrantup.com/downloads.html
- PhpStorm: https://www.jetbrains.com/phpstorm/nextversion/

Creamos la máquina virtual de homestead usando como terminal git bash, e importamos los proyectos a una carpeta Proyectosv en el Escritorio :

```
vagrant box add laravel/homestead
cd Desktop
mkdir Proyectosv
cd Proyectosv
git clone https://github.com/laravel/homestead.git
git clone https://github.com/AngelCanovas/social.git
cd ~/Homestead
git checkout release
init.bat
```

Cambiar lo siguiente del Homestead.yaml (situado en Proyectosv/homestead):

```
folders:
        - map: C:\Users\Username\Desktop\Proyectosv
        to: /home/vagrant/projects

sites:
        - map: social.local
        to: /home/vagrant/projects/social/public
        php: "7.2"

databases:
        - social
```

Añadir al archivo de hosts la dirección de la página: 192.168.10.10 social.local

En la carpeta social dentro de Proyectosv, copiar el contenido del archivo .env.dusk.local a un nuevo archivo .env

En el archivo .env cambiamos nombre de base de datos a social:

```
DB_DATABASE=social
```

Situarse en carpeta homestead en la terminal y levantar la máquina de vagrant:

```
vagrant up
vagrant ssh
cd projects/social
php72
```

Instalar en la máquina de vagrant el chrome:

```
sudo apt-get update
sudo apt-get install chromium-browser
```

Y ahora ejecutar lo siguiente:

```
composer install
php artisan key:generate
php artisan migrate
npm install
npm run dev
php artisan dusk:chrome-driver
```

Finalmente ya puedes acceder en el navegador a la página: http://social.local/
Registrar un usuario (o usar angel@example.com y password: secret) y ya puedes comenzar a utilizar la aplicación.
Para ejecutar tests de phpunit desde terminal, poner: vendor/phpunit/phpunit/phpunit
Y para los test de dusk: php artisan dusk

-> Guia para comenzar el proyecto desde 0 <-

Instalamos los mismos programas que en la guía anterior.

Creamos la máquina virtual de vagrant de la misma forma, pero no clonamos mi proyecto, sino solo el de homestead.

Realizamos los mismos cambios al Homestead.yaml

Añadir al archivo de hosts la dirección de la página: 192.168.10.10 social.local

Situarse en carpeta homestead en la terminal y levantar la máquina de vagrant:

vagrant up
vagrant ssh
php72

[Comienzo del proyecto]

Ejecutar la terminal, dentro de la carpeta Proyectosv:

composer create-project laravel/laravel social 5.6.*

Creamos un test para probar que los usuarios puedan crear estados en la página de inicio mediante terminal:

php artisan make:test CreateStatusTest

Cambiamos el contenido de la clase del fichero recién creado en tests/Features a:

```
class CreateStatusTest extends TestCase
{
        /** @test */
        public function a_user_can_create_statuses()
        {
        // 1. Given => Teniendo un usuario autenticado
        // Creamos datos de prueba usando la factoría UserFactory que usa la librería Faker
        $user = factory(User::class)->create();
        $this->actingAs($user);

        // 2. When => Cuando hace un post request status. Url de la ruta + información a enviar
        $this->post(route('statuses.store'),['body' => 'Mi primer status']);

        // 3. Then => Entonces veo un nuevo estado en la base de datos
        $this->assertDatabaseHas('statuses',[
```

```
            'body' => 'Mi primer status'
            ]);

            }
}
```

Añadiendo dentro del create [‘email’ => ‘angel@angel.com’] por ejemplo, modificamos los valores definidos en el factory, creándolo con un email concreto.

assertDatabaseHas() sirve para verificar datos en la base de datos. Primer parámetro nombre de la tabla a comprobar + array de datos con información a comprobar que existe en la tabla.

Realizamos el test correspondiente por consola vendor/bin/phpunit y comprobamos que efectivamente da error al no escribir nada de codigo de produccion.

Test de dusk se ejecutan con social\vendor\laravel\dusk\bin\chromedriver-win.exe

- Si no carga la página social.local en el navegador (error 502): https://laracasts.com/discuss/channels/laravel/502-bad-gateway-nginx-197-homestead?page=1
  Comentar primera línea de: sudo vim /etc/php/7.4/mods-available/xdebug.ini
  sudo service nginx restart
  sudo service php7.4-fpm restart
- Arreglar conexion laravel dusk vagrant windows: https://github.com/laravel/dusk/issues/55#issuecomment-275976841
  Ejecutar manualmente el chromedriver-win en: **social\vendor\laravel\dusk\bin**

[Fin video 5]

Al ejecutar el test da error de acceso denegado para el usuario homestead, porque no puede crear el usuario al tener configurada una base de datos.
Configuración de base de datos esta en archivo .env, la base de datos mysql por defecto la utilizaremos para pruebas en el navegador y necesitamos crear otra solo para hacer tests porque nos limpiará la base de datos al lanzar pruebas en esa database.

Definimos una nueva database en phpunit.xml, añadimos una nueva línea <env debajo de la anteriores para definir una conexión utilizando sqlite:

```
<env name="DB_CONNECTION" value="sqlite"/>
```

Y el nombre de la base de datos que será en memoria, para que la cree y la elimine cuando se cierre la conexión:

```
<env name="DB_DATABASE" value=":memory:"/>
```

Ejecutamos el test desde consola nuevamente, y ahora muestra un error diferente; que no encuentra la tabla users, porque tenemos la base de datos pero no ejecutamos las migraciones al momento de realizar el test.
Para ello añadimos a CreateStatusTest dentro de la clase:

```
use RefreshDatabase;
```

Ahora al ejecutar el test sale la ruta statuses.store no esta definida, por lo que vamos a crearla:
Buscamos el archivo routes/web y borrando lo anterior creamos la ruta tipo post con un controlador y método store, con el nombre que le dimos a la ruta al crear el test.

```
Route::post('statuses', 'StatusesController@store')->name('statuses.store');
```

Ahora el test nos dice que no existe la tabla statuses, por lo que creamos la migración de la tabla statuses en la terminal.

php artisan make:model Models/Status -mc

Esto nos crea el modelo, la migración y el controlador. Comprobamos en la ruta App\Http\Controllers y cambiamos el nombre a StatusesController clicando encima del nombre y Shif+F6.

Ahora el test no encuentra un registro en la tabla statuses con el valor del body, y que la tabla está vacía.

Añadir al test: `$this->withoutExceptionHandling();` para evitar que laravel maneje las excepciones y ahora aparece que el método store no existe.

Por lo que vamos al StatusesController y creamos el método store.

```
public function store()
{
Status::create(['body' => request('body')]);
}
```

Control+click encima del Status:: para ir al modelo, y añadimos la propiedad guarded donde añadimos los atributos a proteger contra asignación masiva, en nuestro caso ninguno. No hacer request()->all().

```
protected $guarded = [];
```

Agregamos a la migración de los estados en el schema de create_statuses_table:

```
$table->text('body');
```

Y ahora el test pasa, pero no verificamos quien creó el estado por lo que añadimos al test otro parámetro de asserDatabaseHas: `'user_id' => $user->id;`
y añadimos a la migracion: `$table->unsignedInteger('user_id');`
y en el controlador: `'user_id' => auth()->id()`

[Fin video 6]

Verificamos que recibimos una respuesta adecuada, guardamos respuesta en variable, añadiendo lo siguiente en CreateStatusTest encima de assertDatabaseHas:

```
$response = $this->post(route('statuses.store'),['body' => 'Mi primer status']);

$response->assertRedirect('/');
```

Asi verificamos que después de crear el estado redireccione a inicio, vemos que el test falla. Añadimos a StatusesController: return redirect('/'); y pasa.

Pero vamos a utilizar vuejs para enviar estados via ajax. Modificamos el test:

```
$response->assertJson([
'body' => 'Mi primer status'
]);
```

Guardamos el estado del controlador en variable y retornamos la respuesta en json:

```
$status= Status::create([
'body' => request('body'),
'user_id' => auth()->id()
]);

return response()->json(['body' => $status->body]);
```

Para que solo los usuarios autentificados creen estados, hay que añadir un middleware a la ruta, pero no tenemos un test que indique que el middleware es necesario para el correcto funcionamiento. Para ello, creamos un nuevo test en CreateStatusTest:

```
function guests_users_can_not_create_statuses()
{
$response = $this->post(route('statuses.store'),['body' => 'Mi primer status']);

$response->assertRedirect('login');
}
```

Y agregamos el middleware en web.php:

```
Route::post('statuses', 'StatusesController@store')->name('statuses.store')->middleware('auth');
```

Pero sigue fallando, deshabilitamos excepciones para ver el error:

```
$this->withoutExceptionHandling();
```

Para inspeccionar la respuesta que obtenemos: dd($response->content());
El test nos dice entonces que la redirección es ok, pero no esta definida la ruta login.

Agregamos todas las rutas de autenticación en web.php:

```
Route::auth();
```

Al comprobar que funciona correcto, quitamos las líneas que utilizamos para ver los errores.

EXTRA: el middleware puede ponerse también en el controlador en vez de en la ruta. Para ello crear un constructor y agregar el middleware:

```
public function __construct() {
$this->middleware('auth');
}
```

Pero lo hago de la forma anterior

[Fin video 7]

Si enviamos un cuerpo vacío muestra error en el campo body que no puede ser nulo, la solución sería:
Creamos un test para el estado requiere un cuerpo, enviamos cuerpo vacío y verificamos que existe campo body.

```
function a_status_requires_a_body()
{
$user = factory(User::class)->create();
$this->actingAs($user);

$response = $this->postJson(route('statuses.store'),['body' => '']);

}
```

Si usamos ajax añadimos: `$response->assertSessionHasErrors('body');` aparece error, si añadimos la validación en StatusController dentro de store():

`request()->validate(['body' => 'required|min:5']);` pasa el test.

Pero no nos sirve al no redireccionar tras el error, necesitamos un json.

Si usamos en el test: `dd($response->getContent());` vemos el tipo de respuesta que es html, si cambiamos la función post a postJson(), ahora si recibimos el json.

Cambiamos a postJson el post del test de autentificación.
Y verificamos la respuesta Json: que contenga la llave message y la llave error que contiene una llave body.

```
$response->assertJsonStructure([
'message', 'errors' => ['body']
]);
```

Verificamos que el status es 422: `$response->assertStatus(422);`

Duplicamos el test status_requires_body para crear otro que valide el mínimo de caracteres ingresados.

```
/** @test */
function a_status_body_requires_a_minimum_length()
{
$user = factory(User::class)->create();
$this->actingAs($user);

$response = $this->postJson(route('statuses.store'),['body' => 'asdf']);

$response->assertStatus(422);

$response->assertJsonStructure([
'message', 'errors' => ['body']
]);
}
```

Agregamos la otra regla en StatusesController modificando la línea:

```
request()->validate(['body' => 'required|min:5']);
```

Extra: Para agregar + reglas de validación crearíamos 1 o + tests similares al anterior por regla de validación.

[Fin video 8]

Hemos probado la url para crear post, ahora vamos a probar que un usuario pueda crear una publicación desde la página de inicio.

Para probar estas funcionalidades como un navegador real, usaremos laravel dusk:
https://laravel.com/docs/5.6/dusk
Ejecutamos en la terminal: composer require --dev laravel/dusk:"^3.0"
Para que genere archivos necesarios: php artisan dusk:install
Borramos el test de ejemplo generado en la carpeta test/browser.

Y generamos nuestro test:

php artisan dusk:make UsersCanCreateStatusesTest

Ahora modificamos el test creado de la siguiente forma para que vaya al home, rellene un campo con nombre body y de click en elemento create-status:

```
<?php

namespace Tests\Browser;
```

```
use http\Client\Curl\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanCreateStatusesTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * A Dusk test example.
   *
   * @test
   * @throws \Throwable
   */
  public function users_can_create_statuses()
  {
      $user = factory(\App\User::class)->create();

      $this->browse(function (Browser $browser) use ($user) {
          $browser->loginAs($user)
                  ->visit('/')
                  ->type('body', 'Mi primer status')
                  ->press('#create-status')
                  ->screenshot('create-status')
                  ->assertSee('Mi primer status')
          ;
      });
  }
}
```

El método screenshot sirve para que laravel dusk haga una captura de pantalla de la página. Usamos loginAs() para hacer la prueba como un usuario autenticado. Refrescamos la base de datos antes de ejecutar los test con use DatabaseMigrations;

Para ejecutar el test desde terminal: php artisan dusk

En mi caso particular con vagrant en windows es necesario ejecutar el driver de chrome manualmente para ejecutar pruebas de laravel dusk. Pasos a seguir:

Ir a social\vendor\laravel\dusk\bin y ejecutar el chromedriver-win.exe

Ir al archivo DuskTestCase.php (carpeta test) y modificar la ip de RemoteWebDriver por http://10.0.2.2:9515

Cada vez que ejecute test de Laravel Dusk hay que lanzar manualmente el chromedriver

El test dará error al no tener una ruta asociada, para solucionarlo:

Creamos una vista para el home añadiendo al archivo web.php (carpeta routes) lo siguiente para que al ir al home retorne la vista welcome:

```
Route::view('/', 'welcome')->name('home');
```

Ahora modificamos la vista welcome.blade.php (carpeta resources/views) borrando lo que tenga y añadiendo:

```
<form action="{{ route('statuses.store') }}" method="POST">
  @csrf
  <textarea name="body" cols="30" rows="10"></textarea>
  <button id="create-status">Publicar estado</button>
</form>
```

Pero al ejecutar el test nos dará error porque laravel dusk intenta acceder mediante el navegador a la dirección localhost, donde no hay nada. Para solucionarlo modificamos el archivo .env para añadir la dirección correcta:

APP_URL=http://social.local

Además vamos a usar una base de datos diferente para los test de laravel dusk, para ello copiamos el archivo .env y lo pegamos en la misma carpeta renombrandolo por .env.dusk.local , y modificamos:

DB_DATABASE=social_test
DB_USERNAME=root
DB_PASSWORD=

Creamos la base de datos desde la terminal:

mysql -u root
create database social_test;
exit

Comprobamos finalmente que el test pasa con: php artisan dusk
Y ejecutamos los otros test para comprobar que funcionan correctamente: phpunit

Si da error al pasar los test puede ser por un problema de la cache, para arreglarlo probar con los siguientes comandos:
php artisan config:clear (https://stackoverflow.com/questions/46325790/phpunit-expected-status-code-200-but-received-419-with-laravel)
php artisan config:cache (https://stackoverflow.com/questions/58233866/sqlstatehy000-1045-access-denied-for-user-rootlocalhost-using-password)

[Fin video 9]

Vamos a utilizar bootstrap 4 para el diseño de la interfaz.
Empezamos instalando las dependencias de npm definidas en el archivo

package.json. Escribimos en terminal (estando en la carpeta del proyecto):

npm install

Mientras se instala: creamos una plantilla dentro de resources/views, usando phpstorm creamos un nuevo archivo llamado layouts/app.blade.php

En el archivo creado vamos a escribir la estructura html que se repetirá en todas las páginas:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
        content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link rel="stylesheet" href="{{ mix('css/app.css') }}">
  <title>SocialApp</title>
</head>
<body>

<nav class="navbar navbar-expand-lg navbar-light navbar-socialapp">
  <div class="container">


    <a class="navbar-brand" href="#">SocialApp</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mx-auto">
    {{--            <li class="nav-item active">--}}
    {{--                <a class="nav-link" href="#">Inicio <span class="sr-only">(current)</span></a>--}}
    {{--            </li>--}}
    {{--            <li class="nav-item">--}}
    {{--                <a class="nav-link" href="#">Link</a>--}}
    {{--            </li>--}}
        </ul>

        <ul class="navbar-nav ml-auto">
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
              Dropdown
            </a>
            <div class="dropdown-menu" aria-labelledby="navbarDropdown">
```

```
                    <a class="dropdown-item" href="#">Action</a>
                    <a class="dropdown-item" href="#">Another action</a>
                    <div class="dropdown-divider"></div>
                    <a class="dropdown-item" href="#">Something else here</a>
                </div>
            </li>
        </ul>

    </div>
  </div>
</nav>

<main class="py-4">
  @yield('content')
</main>

<script src="{{ mix('js/app.js') }}"></script>
</body>
</html>
```

En esta vista utilizamos la plantilla de un navbar de la página de bootstrap y lo editamos. Añadimos distintas clases de bootstrap a diferentes elementos para darle un formato visual atractivo. Añadimos también las rutas a nuestros ficheros de css (css/app.css) y javascript (js/app.js) .La etiqueta @yield('content') es donde aparece el contenido dinámico en nuestra página.

Modificamos ahora la vista welcome.blade.php de la siguiente manera:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    <div class="row">
      <div class="col-8 mx-auto">
        <div class="card border-0 bg-light">
          <form action="{{ route('statuses.store') }}" method="POST">
            @csrf
            <div class="card-body">
                        <textarea class="form-control border-0 bg-light" name="body"
placeholder="¿Que estas pensando Angel?"></textarea>
            </div>

            <div class="card-footer">
              <button class="btn btn-primary" id="create-status">Publicar</button>
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
@endsection
```

En la vista welcome extendemos al layout que hemos creado anteriormente. Encerramos el formulario en una sección content. Y le damos formato con bootstrap para que quede bien (usando tarjetas bootstrap).

Vamos ahora al archivo webpack.mix.js en la raíz del proyecto y agregamos browserSync al final.

```
mix.browserSync({
  proxy: 'http://social.local',
  open: false
});
```

En la terminal ejecutamos: npm run dev
Una vez finalizada la instalación ejecutamos: npm run watch

Vamos a modificar el css presente en resources/assets/sass/app.scss :
cambiamos la clase navbar-laravel por navbar-socialapp

Finalmente editamos el archivo de variables de bootstrap (resources/assets/sass/_variables.scss) :
cambiamos el color de fondo: $body-bg: #e9ebee;
cambiamos el color de la variable primary añadiendo al final:
$primary: #5e6e9e;

Si ocurren problemas durante la instalación de npm o los comandos de terminal, ver las webs:
https://github.com/JeffreyWay/laravel-mix/issues/1072#issuecomment-319401164
https://laracasts.com/index.php/discuss/channels/laravel/cross-env-not-found-on-npm-run-dev?
https://stackoverflow.com/questions/52215541/module-build-failed-error-cannot-find-module-node-sass
https://stackoverflow.com/questions/33186123/npm-install-errors-on-vagrant-homestead-windows-eproto-protocol-error-symlink
También recordar ejecutar vagrant desde una terminal con privilegios de administrador y levantar la máquina con vagrant up --provision . Si fallan probar a ejecutar los comandos con sudo delante.

[Fin video 10]

Vamos a probar el formulario de login. Para ello creamos un nuevo test de Dusk en la terminal:
php artisan dusk:make LoginTest

Vamos al archivo creado y lo modificamos para que compruebe las características necesarias para que funcione el login. Cuando hacemos consultas a base de datos utilizamos:

```
use DatabaseMigrations.

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class LoginTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * A Dusk test example.
   *
   * @test
   * @throws \Throwable
   */
  public function registered_users_can_login()
  {
      factory(User::class)->create(['email' => 'angel@email.com']);

      $this->browse(function (Browser $browser) {
          $browser->visit('/login')
                  ->type('email', 'angel@email.com')
                  ->type('password', 'secret')
                  ->press('#login-btn')
                  ->assertPathIs('/')
                  ->assertAuthenticated()
          ;
      });
  }
}
```

Para probar el test si ejecutamos php artisan dusk ejecutará todos los tests.
Para ejecutar solo este test lo filtramos de la manera:

```
php artisan dusk --filter registered_users_can_login
```

Nos dará error al no encontrar la vista login. Vamos a crearla en resources/views , crear nuevo archivo auth/login.blade.php

Para revisar las rutas de la aplicación: `php artisan route:list`

Editamos la vista login recién creada. Creamos un formulario, añadimos el token csrf

al necesitarlo, creamos los inputs necesarios y el botón de login.

```
<form action="{{ route('login') }}" method="POST">
  @csrf
  <input type="email" name="email">
  <input type="password" name="password">
  <button id="login-btn">Login</button>

</form>
```

Falta cambiar a donde se redirecciona después de hacer login. Vamos al archivo LoginController.php que trae Laravel por defecto (app\Http\Controllers\Auth) y cambiamos la dirección a la raiz:

```
protected $redirectTo = '/';
```

[Fin video 11]

Vamos a dar diseño al login con bootstrap.

Modificamos el fichero login.blade.php (resources/views/auth) de la siguiente manera:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    <div class="row">
      <div class="col-md-6 mx-auto">
        <div class="card border-0 bg-light px-4 py-2">
          <form action="{{ route('login') }}" method="POST">
            @csrf
            <div class="card-body">

              <div class="form-group">
                <label>Email:</label>
                    <input class="form-control border-0" type="email" name="email"
placeholder="Tu email...">
              </div>

              <div class="form-group">
                <label>Contraseña:</label>
                  <input class="form-control border-0" type="password" name="password"
placeholder="Tu contraseña...">
              </div>

              <button class="btn btn-primary btn-block" id="login-btn">Login</button>
            </div>
```

```
                </form>
            </div>
        </div>
    </div>
  </div>
@endsection
```

Explicacion: Implementamos la plantilla layouts.app anteriormente creada, agregamos la sección content. Esto agrega la barra de navegación. Añadimos la forma de tarjeta al formulario mediante los 4 divs añadidos antes del form (usados anteriormente en la vista welcome).
Añadimos los inputs en un card-body y les damos formato con divs form-group y agregándoles un label a cada input con el nombre de su función.
Agregamos clases de bootstrap a los inputs y button para darles el formato visual necesario.
Cambiamos el ancho del div a col-md-6 para que en pantallas pequeñas ocupe toda la pantalla el login.

Ahora modificamos el archivo app.blade.php (resources/views/layouts):

```
(...)

<nav class="navbar navbar-expand-lg navbar-light navbar-socialapp">
  <div class="container">

    <a class="navbar-brand" href="{{ route('home') }}">SocialApp</a>

(...)

<ul class="navbar-nav ml-auto">
  @guest()
    <li class="nav-item"><a href="{{ route('login') }}" class="nav-link">Login</a></li>
  @else
    <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button"
data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          {{ Auth::user()->name }}
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="#">Action</a>
          <a class="dropdown-item" href="#">Another action</a>
          <div class="dropdown-divider"></div>
            <a onclick="document.getElementById('logout').submit()" class="dropdown-item"
href="#">Cerrar sesión</a>
        </div>
    </li>

    <form id="logout" action="{{ route('logout') }}" method="POST">@csrf</form>
  @endguest
</ul>
```

Explicación: Cambiamos la ruta de la barra de navegación para que cuando hacemos click en SocialApp nos lleve al home. Más abajo en el archivo, añadimos la directiva @guest para que aparezca en la barra de navegación un enlace login cuando no estamos logueados.
Y que cuando SI estamos logueados aparezca la opción de cerrar sesión en el desplegable, usando un form que lleva a la ruta logout.
Además modificamos la palabra Dropdown para que aparezca nuestro nombre de usuario en la barra de navegación al hacer login.

Al no tener una base de datos definida si intentamos loguear, nos dará una excepción. Para solucionarlo, vamos al archivo .env (raíz del proyecto) y modificamos lo siguiente:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=social
DB_USERNAME=root
DB_PASSWORD=
```

Ahora migramos la base de datos, desde terminal: **php artisan migrate**
Vamos a crear un usuario para probar el login, en la terminal:

```
php artisan tinker

factory('App\User')->create()
```

Y aparecerá en pantalla un usuario creado, usamos el correo que nos pone en el campo login de nuestra página (alexis04@example.org) y como contraseña sabemos que por defecto es: secret

Finalmente comprobamos que la aplicación funciona correctamente ejecutando los test de dusk en la terminal: php artisan dusk
(En mi caso recordar que hay que lanzar manualmente el chromedriver desde vendor\laravel\dusk\bin\)

Y los test de phpunit en la terminal: phpunit
(En mi caso no funciona y lo he comprobado desde el phpstorm lanzando los tests individualmente)

[Fin video 12]

Vamos a convertir el formulario de la página home de la aplicación web en un componente de vue.js.

En mi caso aunque vue.js está instalado correctamente, laravel NO ha creado un componente de ejemplo en una carpeta llamada components en resources/assets/js ,ni modificado el archivo app.js en la misma ruta anterior para incluir vuejs en la página web.
Por ello es necesario crear la carpeta en la ruta anterior, y modificar el app.js incluyendo lo siguiente:

```
window.Vue = require('vue');

Vue.component('status-form', require('./components/StatusForm'));

const app = new Vue({
  el: '#app'
});
```

A continuación dentro la carpeta components creada, creamos un nuevo archivo en phpstorm usando como plantilla Vue component y lo llamamos StatusForm.

Modificamos el archivo StatusForm recién creado:

```
<template>
  <div>
    <form @submit.prevent="submit">
      <div class="card-body">
                    <textarea v-model="body" class="form-control border-0 bg-light" name="body" placeholder="¿Que estas pensando Angel?"></textarea>
      </div>

      <div class="card-footer">
        <button class="btn btn-primary" id="create-status">Publicar</button>
      </div>
    </form>
    <div v-for="status in statuses" v-text="status.body"></div>

  </div>
</template>

<script>
  export default {
    data() {
      return {
        body: '',
        statuses: []
      }
    },
    methods: {
      submit() {
        axios.post('/statuses', {body: this.body})
```

```
                .then(res => {
                    this.statuses.push(res.data);
                    this.body = ''
                })
                .catch(err => {
                    console.log(err.response.data)
                })
            }
        }
    }
</script>

<style scoped>

</style>
```

Explicación: El formulario se ha cortado desde la vista welcome.blade.php y pegado en StatusForm. Como vamos a utilizar axios para enviar el formulario, quitamos el método post y el token csrf del formulario.

Ponemos @submit en el form para que escuche cuando se envíe el formulario y llame al método submit que definimos, prevent es para que no se envíe el formulario automáticamente.

El script usa sintaxis de vue.js, añadimos el método submit para enviar el formulario manualmente. Enviamos usando axios a la ruta statuses los campos body. Para ello empleamos un v-model en el formulario que añadimos como datos en la información del componente vuejs.

Encerramos el formulario en un div, y añadimos otro debajo del form para imprimir los estados por pantalla con un loop v-for de los status en statuses e imprimiendo el texto status.body. Esto lo haremos de forma temporal al no ser responsabilidad del formulario mostrar un listado de estados.

Y si recibimos una respuesta exitosa agregue un nuevo estado al array, que es el estado que recibimos como respuesta. Si falla, recibimos los errores y hacemos un console.log.
Finalmente para que cuando se envíe el estado el textarea este vacío añadimos `this.body = ''` después de agregar el estado.

Modificamos el test UserCanCreateStatusTest (tests/browser) y añadimos una nueva línea (waitForText) a lo que comprueba el test para que espere por el texto y de tiempo a Vue.js a que procese y envíe el formulario usando axios.

```
->screenshot('create-status')
->waitForText('Mi primer status')
->assertSee('Mi primer status')
```

En la terminal, para ver los cambios que vamos haciendo: **npm run watch**
En mi caso no me funciona la url del npm run watch, por lo que debo ejecutar npm run dev y actualizar la pagina manualmente para ver los cambios cada vez.
(Si el navegador muestra la página desde cache, refrescar página con Ctrl+F5)

Incluimos el formulario en la vista welcome (resources/views) añadiendo <status-form></status-form> donde estaba antes el formulario, quedaría:

```
@section('content')
  <div class="container">
    <div class="row">
      <div class="col-8 mx-auto">
        <div class="card border-0 bg-light">
          <status-form></status-form>
        </div>
      </div>
    </div>
  </div>
@endsection
```

Si probamos a cargar la página con el inspector, mostrará error de que no encuentra el token csrf y un error de vue de que no encuentra el elemento con id app.
Para solucionarlos, añadimos un nuevo meta al principio de la página debajo de los ya existentes.

```
<link rel="stylesheet" href="{{ mix('css/app.css') }}">
```

Y añadimos el id app al main, de la siguiente forma:

```
<main id="app" class="py-4">
  @yield('content')
</main>
```

Como necesitamos estar logueados para enviar un estado usando el formulario de la página, vamos a crear un seeder de usuarios para crear un usuario que recordemos.
En la consola: **php artisan make:seed UsersTableSeeder**

Vamos a la carpeta database/seeds, modificamos el archivo DatabaseSeeder.php descomentar la linea:

```
$this->call(UsersTableSeeder::class);
```

Y la clase UsersTableSeeder creada la modificamos para crear nuestro usuario:

```
use App\User;
use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
```

```
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        User::truncate();

        factory(User::class)->create(['email' => 'angel@email.com']);
    }
}
```

Refrescamos la base de datos desde terminal: php artisan migrate:fresh --seed
Ahora hacemos login en nuestra aplicación con el usuario creado (angel@email.com en mi caso como usuario) y como contraseña secret (contraseña por defecto).
Y probamos a enviar un estado como 'Hola a todos' y vemos que en la consola de desarrollador del navegador todo va correcto.

Por último ejecutamos los test de dusk (php artisan dusk) y los de phpunit (en mi caso desde phpstorm) para comprobar que no hemos roto nada de la aplicación

IMPORTANTE: Si al ejecutar los test de phpunit da error, puede ser problema de la caché. Ejecutar en terminal: php artisan config:clear
Y volver a lanzar phpunit.
Fuente:https://stackoverflow.com/questions/46325790/phpunit-expected-status-code-200-but-received-419-with-laravel/46326721#46326721

[Fin video 13]

Vamos a implementar una url que devuelve un listado de estados en el orden correcto.

Creamos un nuevo test desde terminal:
php artisan make:test ListStatusesTest

Abrimos el nuevo test (test/feature) y lo modificamos para probar que podemos obtener todos los estados:

```
namespace Tests\Feature;

use App\Models\Status;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;
```

```
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ListStatusesTest extends TestCase
{
  use DatabaseMigrations;

  /** @test */
  function can_get_all_statuses()
  {
    $this->withoutExceptionHandling(); //para obtener detalles del error

    $status1 = factory(Status::class)->create(['created_at' => now()->subDays(4)]);
    $status2 = factory(Status::class)->create(['created_at' => now()->subDays(3)]);
    $status3 = factory(Status::class)->create(['created_at' => now()->subDays(2)]);
    $status4 = factory(Status::class)->create(['created_at' => now()->subDays(1)]);

    $response = $this->getJson(route('statuses.index'));

    $response->assertSuccessful();

    $response->assertJson([
      'total' => 4
    ]);

    $response->assertJsonStructure([
      'data', 'total', 'first_page_url', 'last_page_url'
    ]);

    //dd($response->json('data.0.id'));  //obtener json de la respuesta

    $this->assertEquals(
      $status4->id,
      $response->json('data.0.id')

    );
  }
}
```

Explicación: creamos los estados manualmente en la base de datos con factory() dentro de la función `can_get_all_statuses` para darles distinta fecha de creación, siendo el último creado el status.
Agregamos una ruta con nombre y request tipo Json.
Verificamos respuesta del servidor exitosa. Verificamos el total de estados sea igual a los creados (4 creados). Verificamos que recibimos una respuesta Json con los resultados paginados.
Añadimos el trait DatabaseMigrations para migrar la base de datos automáticamente. Comprobamos que el último estado de la respuesta esté al principio.

Creamos el factory para el modelo Status desde la terminal:

```
php artisan make:factory StatusFactory -m Models\\Status
```

Abrimos el nuevo factory StatusFactory (database/factories) y lo modificamos:

```
use App\User;
use Faker\Generator as Faker;

$factory->define(App\Models\Status::class, function (Faker $faker) {
  return [
     'body' => $faker->paragraph,
     'user_id' => function() {
        return factory(User::class)->create();
     }
  ];
});
```

Explicacion: Cuando creemos un estado, se creara automáticamente un usuario en la base de datos y se guardará el id del usuario en user_id. Y agregamos el body para que no sea null.

Modificamos el archivo de rutas web.php (carpeta routes) añadiendo una nueva ruta get para que al hacer una peticion get no de error:

```
Route::view('/', 'welcome')->name('home');

Route::get('statuses', 'StatusesController@index')->name('statuses.index');
Route::post('statuses', 'StatusesController@store')->name('statuses.store')->middleware('auth');

Route::auth();
```

Como el método index del StatusesController no existe, vamos a añadirlo (app/http/Controllers) donde vamos a enviar todos los estados paginados:

```
public function index()
{
  return Status::latest()->paginate();
}
```

Por último ejecutamos los test de dusk (php artisan dusk) y los de phpunit (en mi caso desde phpstorm) para comprobar que no hemos roto nada de la aplicación

[Fin video 14]

Vamos a mostrar el listado de estados en la página principal.

Creamos un nuevo test de dusk para probar funcionalidades de javascript:

```
php artisan dusk:make UsersCanSeeAllStatusesTest
```

Lo abrimos en phpstorm y lo modificamos:

```
namespace Tests\Browser;

use App\Models\Status;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanSeeAllStatusesTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * @test
   * @throws \Throwable
   */
  public function users_can_see_all_statuses_on_the_homepage()
  {
     $statuses = factory(Status::class, 3)->create();

     $this->browse(function (Browser $browser) use ($statuses) {
        $browser->visit('/')
              ->waitForText($statuses->first()->body);

        foreach($statuses as $status) {
           $browser->assertSee($status->body);
        }
     });
  }
}
```

Explicación: Creamos 3 estados en la base de datos. Cuando visitemos el home queremos que espere por el body del primer estado. Migramos la base de datos. Y usamos un foreach para asegurar que vemos por pantalla todos los estados.

Y vamos comprobando los errores que surgen al ejecutar:

```
php artisan dusk --filter users_can_see_all_statuses_on_the_homepage
```

Vamos a la vista welcome (resources/views) y agregamos el listado de estados:

```
<div class="col-8 mx-auto">
  <div class="card border-0 bg-light">
     <status-form></status-form>
  </div>
  <statuses-list></statuses-list>
</div>
```

Creamos un nuevo componente vuejs desde phpstorm en la carpeta components de la ruta (resources/assets/js) y lo llamamos StatusesList:

```
<template>
  <div>
    <div v-for="status in statuses" v-text="status.body"></div>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        statuses: []
      }
    },
    mounted() {
      axios.get('/statuses')
        .then(res => {
          this.statuses = res.data.data
        })
        .catch(err => {
          console.log(err.response.data);
        });

      EventBus.$on('status-created', status => {
        this.statuses.unshift(status);
      })
    }
  }
</script>

<style scoped>

</style>
```

Explicación: Añadimos el método mounted y llamamos a la ruta statuses. Cuando recibimos la respuesta asignamos los estados que recibimos paginados. Creamos un div dentro del template que contenga los estados y los muestre. Accedemos al eventbus para escuchar eventos y agregamos el estado al array de estados al principio de este .

Añadimos el nuevo componente de vuejs creado al archivo app.js de la ruta (resources/assets/js) y creamos un bus de eventos:

```
window.EventBus = new Vue();

Vue.component('status-form', require('./components/StatusForm'));
Vue.component('statuses-list', require('./components/StatusesList'));
```

Refrescamos la base de datos para empezar desde 0:

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

Comprobamos que al hacer login en la página y publicar un estado aparecen en la página y permanecen.

Modificamos ahora el StatusForm.vue (resources/assets/js/components), eliminamos el loop del template para mostrar los estados y cambiamos para añadir los estados por un bus de eventos de vuejs:

```
<script>
  export default {
      data() {
          return {
              body: '',
          }
      },
      methods: {
          submit() {
              axios.post('/statuses', {body: this.body})
              .then(res => {
                  EventBus.$emit('status-created', res.data);
                  this.body = ''
              })
              .catch(err => {
                  console.log(err.response.data)
              })
          }
      }
  }
</script>
```

[Fin video 15]

SOLUCION ERROR: SQLSTATE[HY000] [1045] Access denied for user 'root'@'localhost' (using password: NO)
https://stackoverflow.com/questions/30594962/sqlstatehy000-1045-access-denied-for-user-rootlocalhost-using-password/37061837

Modificar archivo database.php, linea de conexion a base de datos por: 'default' => env('DB_CONNECTION', ''),
Modificar archivo .env línea password base de datos por: DB_PASSWORD=secret

Vamos a estilizar los estados con bootstrap.
En terminal ejecutamos el watcher para actualizar los archivos automáticamente:

npm run watch

Modificamos el archivo StatusesList para dar formato a los estados:

```
<template>
  <div>
    <div class="card border-0 mb-3 shadow-sm" v-for="status in statuses" >
      <div class="card-body d-flex flex-column">
        <div class="d-flex align-items-center mb-3">
          <img class="rounded mr-3 shadow-sm" width="40px" src="https://aprendible.com/
images/default-avatar.jpg" alt="">
          <div class="">
            <h5 class="mb-1">Angel C.</h5>
            <div class="small text-muted">Hace una hora</div>
          </div>
        </div>
        <p class="card-text text-secondary" v-text="status.body"></p>
      </div>
    </div>
  </div>
</template>
```

Explicación: Mostramos cada estado como una tarjeta. Creamos una representación de cómo queremos que se vean los estados, con datos provisionales. Para ello usamos la funcionalidad flexbox de bootstrap 4.

Y añadimos unas clases de bootstrap a la vista welcome.blade.php para darle un formato acorde al estilo de los estados (resources/views):

```
<div class="card border-0 bg-light mb-3 shadow-sm">
  <status-form></status-form>
</div>
```

Por último ejecutamos los test de dusk (php artisan dusk) y los de phpunit (en mi caso desde phpstorm) para comprobar que no hemos roto nada de la aplicación

[Fin video 16]

Antes de comenzar el video 17 me han salido errores al ejecutar npm run dev. Realizo lo que se explica en el enlace y todo solucionado:
https://github.com/JeffreyWay/laravel-mix/issues/1072#issuecomment-319401164

En el componente de vue StatusesList donde agregamos los estados manualmente, no es lo correcto pues cada estado tendrá el mismo nombre y hora. Para

solucionarlo vamos a hacer uso de recursos API.

Primero creamos un test unitario para probarlo, con la misma estructura de carpetas de la aplicación:

```
php artisan make:test Http/Resources/StatusResourceTest --unit
```

Vamos al test recién creado (tests/Unit/Resources) y lo modificamos:

```
namespace Tests\Unit\Http\Resources;

use App\Http\Resources\StatusResource;
use App\Models\Status;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class StatusResourceTest extends TestCase
{
  use RefreshDatabase;
  /** @test */
  public function a_status_resources_must_have_the_necessary_fields()
  {
     $status = factory(Status::class)->create();

     $statusResource = StatusResource::make($status)->resolve();

     $this->assertEquals(
        $status->body,
        $statusResource['body']
     );
     $this->assertEquals(
        $status->user->name,
        $statusResource['user_name']
     );
     $this->assertEquals(
        'https://aprendible.com/images/default-avatar.jpg',
        $statusResource['user_avatar']
     );
     $this->assertEquals(
        $status->created_at->diffForHumans(),
        $statusResource['ago']
     );

  }
}
```

Explicación: Creamos un estado con factory. Usamos el trait RefreshDatabase. Creamos el recurso api statusResource y le pasamos el estado para que lo transforme y devuelva el array que retornara al devolverlo de un controlador o ruta con el método resolve.

Comprobamos el contenido de cada campo con assertEquals en vez de assertArrayHasKey para que de error si por ejemplo si enviamos un body que no es el que necesitamos. En el avatar colocamos la imagen para que pase temporalmente.

Creamos el recurso anteriormente mencionado en la terminal:

```
php artisan make:resource StatusResource
```

Vamos al recurso Api para modificar la respuesta (app/http/Resources/StatusResource.php):

```
public function toArray($request)
{
  return [
     'body' => $this->body,
     'user_name' => $this->user->name,
     'user_avatar' => 'https://aprendible.com/images/default-avatar.jpg',
     'ago' => $this->created_at->diffForHumans()
  ];
}
```

Explicación: Accedemos al estado body. Accedemos a la relación user y al nombre de usuario. En el user_avatar devolvemos la imagen provisionalmente. Y en ago devuelve la fecha de creación formateada por Carbon.

Creamos un nuevo test al no estar creada la relación entre el estado y el usuario. Para ello creamos primero un test mediante la terminal:

```
php artisan make:test Models/StatusTest --unit
```

Y vamos al test creado para modificarlo (tests/Unit/Models/StatusTest.php):

```
namespace Tests\Unit\Models;

use App\User;
use Tests\TestCase;
use App\Models\Status;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class StatusTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function a_status_belongs_to_a_user()
  {
     $status = factory(Status::class)->create();

     $this->assertInstanceOf(User::class, $status->user);
```

```
    }
}
```

Explicación: Creamos un estado en la base de datos. Usamos `assertInstanceOf` para verificar si un objeto es una instancia de determinada clase. Verificamos que recibimos una instancia de la clase User cuando llamemos a `$status->user`. Y agregamos el trait de `RefreshDatabase`.

Vamos al modelo Status.php de (app/Models) y añadimos lo siguiente para que se reciba una instancia de App\User en el test anterior:

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

[Fin video 17]

Debido a problemas en el video anterior al ejecutar los test creados, saliendo error:
`array_key_exists(): Using array_key_exists() on objects is deprecated. Use isset() or property_exists() instead`
He cambiado la versión de php a 7.2, por ser un problema que ocurre con versiones modernas de php en proyectos antiguos. Para ello sengun la documentacion: https://laravel.com/docs/7.x/homestead#multiple-php-versions
Modifico el archivo Homestead.yaml () añadiendo a los sites php: "7.2" en social.
Y una vez iniciada la maquina con --provision ejecutar dentro de ella: php72 para modificar la versión de cli usada. Y no olvidar cambiar la version de php que usa phpstorm actualizando a la 7.2 de vagrant.

Vamos a utilizar el recurso API creado.
Creamos estados con tinker, obtenemos los estados paginados, y transformamos los estados paginados:

```
php artisan tinker
factory('App\Models\Status', 10)->create()
$stauses = App\Models\Status::paginate();
App\Http\Resources\StatusResource::collection($stauses)->response()->getData()
```

En la respuesta vemos una llave data, y dentro los estados transformados como queremos. Pero también hay una llave link y otra meta que no hemos definido.

Modificamos ListStatusesTest (test/feature) para que pruebe la respuesta del recurso. Agregando la llave meta al total, agregamos los links prev y next a la estructura a comprobar, y basamos la verificamos del orden en el body:

```
$response->assertJson([
  'meta' => ['total' => 4]
]);
$response->assertJsonStructure([
  'data', 'links' => ['prev', 'next']
]);

//dd($response->json('data.0.id'));

$this->assertEquals(
  $status4->body,
  $response->json('data.0.body')

);
```

Al ejecutar el test (desde phpstorm) falla al esperar un meta pero obtener respuesta directa de los datos paginados. Para solucionarlo realizamos la implementación cambiando el StatusesController (app/http/controllers):

```
public function index()
{
  return StatusResource::collection(
      Status::latest()->paginate()
  );
}

(... al final en el metodo store ...)

return StatusResource::make($status);
```

Explicación: Encerramos los datos paginados en el recurso Api usando el método collection que recibe los estados por parámetro. Y retornamos el statusresource usando el método make para que transforme un estado.

Mostramos la nueva informacion por pantalla modificando el test UsersCanSeeAllStatusesTest (tests/browser) para que verifique los cambios a realizar. Aparte de ver el cuerpo de cada estado, queremos ver el nombre de usuario del que creó el estado y la fecha de publicación:

```
foreach($statuses as $status) {
  $browser->assertSee($status->body)
      ->assertSee($status->user->name)
      ->assertSee($status->created_at->diffForHumans())
  ;
}
```

Usamos en terminal npm run watch para que compile los cambios que vamos a realizar automáticamente. Y modificamos StatusesLis.vue (resources/assets/js/components) para añadir nombre de usuario y fecha:

```
<template>
  <div>
    <div class="card border-0 mb-3 shadow-sm" v-for="status in statuses" >
      <div class="card-body d-flex flex-column">
        <div class="d-flex align-items-center mb-3">
          <img class="rounded mr-3 shadow-sm" width="40px" src="https://aprendible.com/
images/default-avatar.jpg" alt="">
          <div class="">
            <h5 class="mb-1" v-text="status.user_name"></h5>
            <div class="small text-muted" v-text="status.ago"></div>
          </div>
        </div>
        <p class="card-text text-secondary" v-text="status.body"></p>
      </div>
    </div>
  </div>
</template>
```

Ejecutamos test para comprobar que pasa:

```
php artisan dusk --filter users_can_see_all_statuses_on_the_homepage
```

Para que cuando el usuario cree un test y se muestre automáticamente en ese listado, modificamos UsersCanCreateStatusesTest (tests/browser) para que verifique el nombre del usuario $user:

```
$this->browse(function (Browser $browser) use ($user) {
  $browser->loginAs($user)
        ->visit('/')
        ->type('body', 'Mi primer status')
        ->press('#create-status')
        ->screenshot('create-status')
        ->waitForText('Mi primer status')
        ->assertSee('Mi primer status')
        ->assertSee($user->name)
  ;
});
```

Modificamos CreateStatusTest (test/feature) para que la respuesta contenga un body dentro de la respuesta data:

```
public function an_authenticated_user_can_create_statuses()
{
  $user = factory(User::class)->create();
  $this->actingAs($user);
```

```
    $response = $this->postJson(route('statuses.store'),['body' => 'Mi primer status']);

    $response->assertJson([
        'data' => ['body' => 'Mi primer status'],
    ]);

    $this->assertDatabaseHas('statuses',[
        'user_id' => $user->id,
        'body' => 'Mi primer status'
    ]);

}
```

Para que vea el nombre del usuario después de crear un estado modificamos StatusForm.vue (resources/assets/js/components) accediendo a la llave data que dentro tiene la llave body, por utilizar ahora el recurso api statusresource:

```
axios.post('/statuses', {body: this.body})
.then(res => {
    EventBus.$emit('status-created', res.data.data); // ['data' => ['body' => 'asdfg']]
    this.body = ''
})
```

Finalmente ejecutamos los test de dusk (php artisan dusk) y los de phpunit (en mi caso desde phpstorm) para comprobar que no hemos roto nada de la aplicación.

Y comprobamos en el navegador que estando logueados, si creamos un nuevo estado se agrega correctamente con el nombre y fecha formateada.

[Fin video 18]

Vamos a mostrar el nombre del usuario autenticado en el formulario de publicar estados usando vue.js.

En el layout app.blade.php (resources/views/layouts), de la misma forma que pasamos el token csrf pasamos el usuario actualmente autenticado:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
<meta name="user" content="{{ Auth::user() }}">
<link rel="stylesheet" href="{{ mix('css/app.css') }}">
```

Modificamos el StatusForm.vue (resources/assets/js/components), haciendo un v-bind al placeholder (añadir :) y usando template string (` `) imprimimos la variable currentuser en vez del nombre anterior. Y si el usuario es un invitado que no muestre el formulario de publicar estados y salga un botón que redireccione al login:

```
<template>
  <div>
    <form @submit.prevent="submit" v-if="isAuthenticated">
      <div class="card-body">
        <textarea v-model="body"
                  class="form-control border-0 bg-light"
                  name="body"
                  :placeholder="`¿Que estas pensando ${currentUser.name}?`">

        </textarea>
      </div>

      <div class="card-footer">
        <button class="btn btn-primary" id="create-status">Publicar</button>
      </div>
    </form>
    <div v-else class="card-body">
      <a href="/login">Debes hacer login</a>
    </div>
  </div>
</template>
```

Creamos el usuario como propiedad calculada usando un mixins para evitar repetición de codigo y poder usar el codigo en otras partes de la aplicación. Para ello, dentro de la carpeta js (resources/assets) creamos la carpeta mixins y dentro el archivo auth.js . Modificamos el archivo añadiendo:

```
let user = document.head.querySelector('meta[name="user"]');

module.exports = {
  computed: {
    currentUser() {
      return JSON.parse(user.content);
    },
    isAuthenticated(){
      return !! user.content;
    },
    guest(){
      return ! this.isAuthenticated;
    }

  },
};
```

Explicación: Usando la linea de codigo `let token = document.head.querySelector('meta[name="csrf-token"]');` del archivo bootstrap.js como ejemplo para obtener el usuario a través del meta.
Y exportamos las propiedades calculadas con un module.exports . Añadimos nuevas propiedades para comprobar si el usuario está autenticado (convirtiendo en boolean la respuesta con !!), y si el usuario en un invitado.

Podemos usar el mixins en StatusForm importando el mixins:

import auth from '../mixins/auth';

Y añadiendo: mixins: [auth], debajo de la llave data().

Pero para poder usarlo en otras partes de la aplicación sin tener que importarlo (pues se necesitará en muchos otros sitios), lo agregamos al archivo app.js (resources/assets/js) para que esté presente globalmente:

```
import auth from './mixins/auth';
Vue.mixin(auth);
```

[Fin video 19]

Vamos a permitir que los usuarios puedan dar me gusta a los estados.

Empezamos escribiendo el test: php artisan make:test CanLikeStatusesTest

```
namespace Tests\Feature;

use App\User;
use Tests\TestCase;
use App\Models\Status;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanLikeStatusesTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function an_authenticated_user_can_like_statuses()
  {
     $this->withoutExceptionHandling();

     $user = factory(User::class)->create();
     $status = factory(Status::class)->create();

     $this->actingAs($user)->postJson( route('statuses.likes.store', $status));

     $this->assertDatabaseHas('likes', [
        'user_id' => $user->id,
        'status_id' => $status->id

     ]);

  }
}
```

Explicación: Primero probamos que un usuario autenticado pueda dar like a los estados. Creamos el usuario que dara like con factory, y el estado a dar like.

Actuando como el usuario hacemos un post a la ruta statuses.likes.store , y verificamos en la base de datos que hay una tabla likes y que existe un registro del usuario que realiza la acción y un campo status_id. Deshabilitamos manejo de excepciones. .

Creamos la ruta statuses.likes.store en routes/web.php, añadiendo una ruta con url statuses, parámetro status y luego likes. Usando un nuevo controlador StatusLikesController que crearemos a continuación:

```
// Statuses Likes routes
Route::post('statuses/{status}/likes', 'StatusLikesController@store')->name('statuses.likes.store')->middleware('auth');
```

Creamos la migración para crear la tabla likes, en terminal:

```
php artisan make:migration create_likes_table
```

Creamos el controlador, en terminal:

```
php artisan make:controller StatusLikesController
```

Y agregamos el método store al controlador (app/http/controllers) que recibe como parámetro el estado y creamos un nuevo like:

```
namespace App\Http\Controllers;

use App\Models\Status;
use Illuminate\Http\Request;

class StatusLikesController extends Controller
{
  public function store(Status $status)
  {
     $status->like();
  }
}
```

Implementamos un nuevo test en StatusTest (tests/unit/models) que verifique el método like, añadiendo:

```
/** @test */
function a_status_has_many_likes()
{
  $status = factory(Status::class)->create();

  factory(Like::class)->create(['status_id' => $status->id]);

  $this->assertInstanceOf(Like::class, $status->likes->first());
}
```

Creamos el modelo like por terminal: php artisan make:model Models/Like
Importamos la clase like al principio del StatusTest:

```
use App\Models\Like;
```

Creamos el factory para el modelo Like en la terminal:

```
php artisan make:factory LikeFactory -m Models/Like
```

Vamos ahora a la migración de create_likes_table (database/migrations) y agregamos el status_id y el user_id:

```
public function up()
{
  Schema::create('likes', function (Blueprint $table) {
    $table->increments('id');
    $table->unsignedInteger('user_id');
    $table->unsignedInteger('status_id');
    $table->timestamps();
  });
}
```

Implementamos la relación de likes en el modelo Status (app/Models) añadiendo:

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

Vamos al modelo Like (app/models) para desabilitar la proteccion:

```
protected $guarded = [];
```

Vamos a probar que un invitado no pueda dar likes a los estados, añadimos en CanLikeStatusTest al principio:

```
  /** @test */
  function guests_users_can_not_like_statuses()
  {
    $status = factory(Status::class)->create();

    $response = $this->post(route('statuses.likes.store', $status));

    $response->assertRedirect('login');
  }
```

Hacemos una reestructuración del codigo implementando el método like() del StatusLikesController. Pero antes creamos un test para verificar el funcionamiento de este método, en StatusTest (tests/unit/models) añadimos un nuevo test:

```
/** @test */
function a_status_can_be_liked()
{
  $status = factory(Status::class)->create();

  $this->actingAs( factory(User::class)->create() );
```

```
    $status->like();

    $this->assertEquals(1, $status->likes->count());
}
```

Asi verificamos que un usuario autenticado llama al método like y que la cantidad de likes del estado sea 1.

Creamos el método like en Status.php (app/models) añadiendolo al final:

```
public function like()
{
    $this->likes()->firstOrCreate([
        'user_id' => auth()->id()
    ]);
}
```

Por último creamos un nuevo test en StatusTest para comprobar que un usuario no puede dar más de un like a un estado, añadimos:

```
/** @test */
function a_status_can_be_liked_once()
{
    $status = factory(Status::class)->create();

    $this->actingAs( factory(User::class)->create() );

    $status->like();

    $this->assertEquals(1, $status->likes->count());

    $status->like();

    $this->assertEquals(1, $status->fresh()->likes->count());
}
```

Finalmente para que los tests creados anteriormente pasen, es necesario añadir una información por defecto al LikeFactory (database/factories) creado

```
use App\User;
use App\Models\Status;
use Faker\Generator as Faker;

$factory->define(App\Models\Like::class, function (Faker $faker) {
    return [
        'user_id' => function(){
            return factory(User::class)->create();
        },
        'status_id' => function(){
```

```
        return factory(Status::class)->create();
    }
  ];
});
```

Y compilamos los assets con npm run dev para que pasen los test de Dusk.

[Fin video 20]

Vamos a implementar el botón para dar like a los estados. Empezamos implementando el test: php artisan dusk:make UsersCanLikeStatusesTest
Y lo modificamos:

```
use App\User;
use App\Models\Status;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanLikeStatusesTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * @test
   * @throws \Throwable
   */
  public function users_can_like_statuses()
  {
    $user = factory(User::class)->create();
    $status = factory(Status::class)->create();

    $this->browse(function (Browser $browser) use ($user, $status){
      $browser->loginAs($user)
        ->visit('/')
        ->waitForText($status->body)
        ->press('@like-btn') // arroba es selector de dusk
        ->waitForText('TE GUSTA') // texto del boton cambia a te gusta
        ->assertSee('TE GUSTA')
      ;
    });
  }
}
```

Explicación: Creamos un usuario en la base de datos, un estado para darle like, el trait de base de datos, pasamos las variables creadas al closure.
Nos autenticamos con el usuario creado, visitamos el homepage y esperamos el

texto status->body (para que dé tiempo a renderizar los estados con vuejs). Presionamos el botón like-btn, y espere recibir el texto ME GUSTA.

Usamos el **npm run watch** para actualizar los archivos que estamos modificando.

Vamos a crear el botón para que pase el test, abrimos StatusesList (resources/assets/js/components) y modificamos el template para mostrar un botón u otro dependiendo de si el estado me gusta:

```
</div>
  <p class="card-text text-secondary" v-text="status.body"></p>
  <button v-if="status.is_liked">TE GUSTA</button>
  <button v-else dusk="like-btn" @click="like(status)">ME GUSTA</button>
</div>
```

(... y añadimos el método like en el script, que recibe el estado como parámetro y al recibir la respuesta cambie la propiedad is_liked a verdadero …)

```
    EventBus.$on('status-created', status => {
        this.statuses.unshift(status);
      })
    },
    methods: {
      like(status){
        axios.post(`/statuses/${status.id}/likes`)
          .then(res => {
            status.is_liked = true;
          })
      }
    }
  }
</script>
```

Creamos la propiedad is_liked en el StatusResourceTest (tests/unit/http/resources):

```
$this->assertEquals(
        $status->created_at->diffForHumans(),
        $statusResource['ago']
    );
    $this->assertEquals(
        false,
        $statusResource['is_liked']
    );
  }
}
```

Definimos ahora la propiedad is_liked en StatusResource (app/http/resources) cuya lógica delegamos en un método isLiked del modelo:

```
            'user_avatar' => 'https://aprendible.com/images/default-avatar.jpg',
            'ago' => $this->created_at->diffForHumans(),
            'is_liked' => $this->isLiked()
        ];
    }
}
```

Creamos un test unitario específico para testear el método anterior isLiked, para ello vamos al Statustest (tests/unit/models) y creamos un nuevo test al final:

```
/** @test */
function a_status_knows_if_it_has_been_liked()
{
    $status = factory(Status::class)->create();

    $this->assertFalse($status->isLiked());

    $this->actingAs(factory(User::class)->create());

    $this->assertFalse($status->isLiked());

    $status->like();

    $this->assertTrue($status->isLiked());
}
```

Explicación: Verificamos que devuelva falso por defecto al llamar al método isLiked. Autenticamos al usuario creado, y al darle like al estado comprobamos que isLiked devuelve verdadero.

Implementamos el método isLiked en el modelo Status (app/models) al final, donde obtenemos todos los likes del estado y si dentro de ellos existe 1 con el user_id del usuario actualmente autenticado.:

```
public function isLiked()
{
    return $this->likes()->where('user_id', auth()->id())->exists();
}
```

Si ejecutamos el test de dusk: php artisan dusk --filter users_can_like_statuses
Vemos que falla. Revisamos los screenshots (tests/browser/screenshots) y vemos que aparece el boton correctamente. Revisamos el log de la consola del navegador (tests/browser/console) y vemos que falla al haber un undefined en la ruta.
Para solucionarlo, debemos retornar un id en el StatusResource (app/http/resources).

Añadimos el test para que compruebe esta característica en StatusResourceTest

(tests/unit/http/resources) añadiendo que status-id sea igual al del statusResource:

```
public function a_status_resources_must_have_the_necessary_fields()
{
  $status = factory(Status::class)->create();

  $statusResource = StatusResource::make($status)->resolve();

  $this->assertEquals(
     $status->id,
     $statusResource['id']
  );
  $this->assertEquals(
     $status->body,
     $statusResource['body']
  );
```

Hacemos ahora una prueba manual en el navegador, vemos que muestra errores por consola al faltar ejecutar la nueva migración creada.
Ejecutamos **php artisan migrate** para solucionarlo

[Fin video 21]

Vamos a permitir quitar los likes al usuario. Para ello vamos a crear un nuevo test en CanLikeStatusesTest siguiendo la forma del test anterior realizado, añadimos:

```
/** @test */
function an_authenticated_user_can_unlike_statuses()
{
  $this->withoutExceptionHandling();

  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();
  $this->actingAs($user)->postJson( route('statuses.likes.store', $status));

  $this->actingAs($user)->deleteJson( route('statuses.likes.destroy', $status));


  $this->assertDatabaseMissing('likes', [
     'user_id' => $user->id,
     'status_id' => $status->id

  ]);

}
```

Explicación: Necesitamos un estado que haya sido gustado por un usuario para

poder quitar el like. Hacemos un delete a la ruta statuses.likes.destroy y verifique que la base de datos no tenga los campos.

Añadimos la ruta statuses.likes.destroy al archivo de rutas web.php (routes):

```
Route::delete('statuses/{status}/likes', 'StatusLikesController@destroy')->name('statuses.likes.destroy')->middleware('auth');
```

Añadimos el método destroy al StatusLikesController (app/http/controllers):

```
public function destroy(Status $status)
{
  $status->unlike();
}
```

Como el método destroy no lo tenemos, lo comprobamos con un test. Para ello modificamos el test a_status_can_be_liked de StatusTest.php (tests/unit/models) y lo reemplazamos de la siguiente manera:

```
/** @test */
function a_status_can_be_liked_and_unlike()
{
  $status = factory(Status::class)->create();

  $this->actingAs( factory(User::class)->create() );

  $status->like();

  $this->assertEquals(1, $status->fresh()->likes->count());

  $status->unlike();

  $this->assertEquals(0, $status->fresh()->likes->count());

}
```

Agregamos el método unlike en el modelo Status (app/models) debajo del metodo like. El método busca el like con el user_id del usuario autenticado y lo quitamos:

```
public function unlike()
{
  $this->likes()->where([
     'user_id' => auth()->id()
  ])->delete();
}
```

Ahora verificamos que los cambios realizados funcionan en la interfaz de usuario, vamos al UsersCanLikeStatusesTest (tests/browser) y añadimos al test que busque un botón unlike-btn, lo pulse y vea el texto ME GUSTA:

```
public function users_can_like_and_unlike_statuses()
```

```
{
  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();

  $this->browse(function (Browser $browser) use ($user, $status){
    $browser->loginAs($user)
      ->visit('/')
      ->waitForText($status->body)
      ->press('@like-btn') // arroba es selector de dusk
      ->waitForText('TE GUSTA') // texto del botón cambia a te gusta
      ->assertSee('TE GUSTA')

      ->press('@unlike-btn')
      ->waitForText('ME GUSTA')
      ->assertSee('ME GUSTA')
    ;
  });
}
```

Agregamos el boton unlike-btn en el StatusesList (resources/assets/js/components) en el template y el método unlike en los métodos de vue donde pasamos el estado:

```
 </div>
  <p class="card-text text-secondary" v-text="status.body"></p>
        <button v-if="status.is_liked" dusk="unlike-btn" @click="unlike(status)">TE
GUSTA</button>
  <button v-else dusk="like-btn" @click="like(status)">ME GUSTA</button>
</div>

(... y despues en el script ...)

methods: {
  like(status){
    axios.post(`/statuses/${status.id}/likes`)
      .then(res => {
        status.is_liked = true;
      })
  },
  unlike(status){
    axios.delete(`/statuses/${status.id}/likes`)
      .then(res => {
        status.is_liked = false;
      })
  }
}
```

Revisamos que funcionan los cambios en la interfaz del navegador compilando los cambios realizados con **npm run dev** y ejecutando los test de dusk en terminal.

[Fin video 22]

Vamos a darle estilos a los botones con iconos de fontawesome.
Primero a la vista welcome.blade, modificamos como de grandes son los comentarios en pantallas pequeñas con bootstrap:

```
( … )
<div class="row">
  <div class="col-md-8 mx-auto">
    <div class="card border-0 bg-light mb-3 shadow-sm">
      <status-form></status-form>
( … )
```

Añadimos en el head de la vista app.blade (resources/views/layout) el cdn de font-awesome, y añadimos un icono al nombre de la aplicación:

```
<meta name="user" content="{{ Auth::user() }}">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.13.0/css/all.css"
integrity="sha384-Bfad6CLCknfcloXFOyFnlgtENryhrpZCe29RTifKEixXQZ38WheV+i/
6YWSzkz3V" crossorigin="anonymous">
<link rel="stylesheet" href="{{ mix('css/app.css') }}">
<title>SocialApp</title>

( … Y en el navbar  ... )

<nav class="navbar navbar-expand-lg navbar-light navbar-socialapp">
  <div class="container">

      <a class="navbar-brand" href="{{ route('home') }}"><i class="fa fa-address-book text-
primary mr-1"></i> SocialApp</a>
```

Modificamos ahora el template del componente StatusesList (resources/assets/js/components) para añadir iconos dentro de un card-footer:

```
<template>
  <div>
    <div class="card border-0 mb-3 shadow-sm" v-for="status in statuses" >
      <div class="card-body d-flex flex-column">
        <div class="d-flex align-items-center mb-3">
          <img class="rounded mr-3 shadow-sm" width="40px" src="https://aprendible.com/
images/default-avatar.jpg" alt="">
          <div class="">
            <h5 class="mb-1" v-text="status.user_name"></h5>
            <div class="small text-muted" v-text="status.ago"></div>
          </div>
        </div>
        <p class="card-text text-secondary" v-text="status.body"></p>
      </div>
      <div class="card-footer pt-2">
```

```
                    <button class="btn btn-link btn-sm" v-if="status.is_liked" dusk="unlike-btn" @click="unlike(status)"><strong><i class="fa fa-thumbs-up text-primary mr-1"></i> TE GUSTA</strong></button>
                <button class="btn btn-link btn-sm" v-else dusk="like-btn" @click="like(status)"><i class="far fa-thumbs-up text-primary mr-1"></i> ME GUSTA</button>
            </div>
        </div>
    </div>
</template>
```

Por último añadimos un icono al StatusForm (resources/assets/js/components):

```
    <div class="card-footer">
        <button class="btn btn-primary" id="create-status"><i class="fa fa-paper-plane mr-1"></i> Publicar</button>
    </div>
</form>
```

Y compilamos los cambios con npm run dev para aplicarlos.

[Fin video 23]

He tenido el problema de que no podía hacer login, daba error de tabla social.users no existe, para solucionarlo:
Ejecutar en terminal: php artisan migrate:fresh --seed

Vamos a hacer que cuando un invitado haga click en me gusta de un estado, redirecciona al login.

Creamos un nuevo test para probar la funcionalidad a crear en UsersCanLikeStatusesTest (test/browser), antes del test ya creado:

```
class UsersCanLikeStatusesTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * @test
     * @throws \Throwable
     */
    public function guest_users_cannot_like_statuses()
    {
        $status = factory(Status::class)->create();

        $this->browse(function (Browser $browser) use ($status){
            $browser->visit('/')
                ->waitForText($status->body)
                ->press('@like-btn')
                ->assertPathIs('/login')
```

```
        ;
    });
  }
```

Vamos al componente StatusesList (resources/assets/js/components) y añadimos al template la función redirectIfGuest para que redireccione al hacer click en el elemento:

```
<template>
  <div @click="redirectIfGuest">
    <div class="card border-0 mb-3 shadow-sm" v-for="status in statuses" >
      <div class="card-body d-flex flex-column">
```

Implementamos la lógica de la función en el mixin (resources/assets/js/mixins), para poder acceder a ella desde otras partes de la página:

```
  guest(){
      return ! this.isAuthenticated();
    }
  },
  methods: {
    redirectIfGuest(){
      if (this.guest) {
        return window.location.href = '/login';
      }
    }
  }
};
```

Ejecutamos los test de dusk: **php artisan dusk** Y vemos que el test users_can_see_all_statuses_on_the_homepage del archivo UsersCanSeeAllStatusesTest falla. Para solucionarlo, ajustamos el desfase de tiempo al crear los usuarios:

```
public function users_can_see_all_statuses_on_the_homepage()
{
  $statuses = factory(Status::class, 3)->create(['created_at' => now()->subMinute()]);

  $this->browse(function (Browser $browser) use ($statuses) {
```

[Fin video 24]

Vamos a mostrar el número de likes que tiene cada estado.
Comenzamos creando el test, añadiendo un nuevo test en StatusTest (tests/unit/models) al final:

```
/** @test */
```

```
function a_status_knows_how_many_likes_it_has()
{
  $status = factory(Status::class)->create();

  $this->assertEquals(0, $status->likesCount());

  factory(Like::class, 2)->create(['status_id' => $status->id]);

  $this->assertEquals(2, $status->likesCount());
}
```

Explicación: Creamos un estado y le añadimos 2 likes. Luego comprobamos que el total de likes del estado sea 0 antes de crearlo y de 2 al crearlo.

Creamos el método likesCount, en el modelo Status (app/models) añadiendo al final el método que retorna el conteo de los likes del estado :

```
public function likesCount()
{
  return $this->likes()->count();
}
```

Para mostrar estos estados en la pantalla, primero modificamos el test users_can_like_and_unlike_statuses de UsersCanLikeStatusesTest y agregamos la comprobacion del numero de likes dentro del elemento likes-count:

```
public function users_can_like_and_unlike_statuses()
{
  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();

  $this->browse(function (Browser $browser) use ($user, $status){
    $browser->loginAs($user)
      ->visit('/')
      ->waitForText($status->body)
      ->assertSeeIn('@likes-count', 0)
      ->press('@like-btn')
      ->waitForText('TE GUSTA')
      ->assertSee('TE GUSTA')
      ->assertSeeIn('@likes-count', 1)

      ->press('@unlike-btn')
      ->waitForText('ME GUSTA')
      ->assertSee('ME GUSTA')
      ->assertSeeIn('@likes-count', 0)
    ;
  });
}
```

Creamos ahora el elemento likes-count (con el estado con atributo likes-count) en el

formulario StatusesList (resources/assets/js/components) dentro del card-footer:

```
<div class="card-footer pt-2">
        <button class="btn btn-link btn-sm" v-if="status.is_liked" dusk="unlike-btn" @click="unlike(status)"><strong><i class="fa fa-thumbs-up text-primary mr-1"></i> TE GUSTA</strong></button>
      <button class="btn btn-link btn-sm" v-else dusk="like-btn" @click="like(status)"><i class="far fa-thumbs-up text-primary mr-1"></i> ME GUSTA</button>
  <span dusk="likes-count">{{ status.likes_count }}</span>
</div>
```

(… Y añadimos que cuando damos like o unlike, actualice el conteo de likes del estado dentro de los métodos …)

```
methods: {
  like(status){
    axios.post(`/statuses/${status.id}/likes`)
      .then(res => {
        status.is_liked = true;
        status.likes_count++;
      })
  },
  unlike(status){
    axios.delete(`/statuses/${status.id}/likes`)
      .then(res => {
        status.is_liked = false;
        status.likes_count--;
      })
  }
```

Vamos al StatusResourceTest (tests/unit/http/resources) y añadimos el atributo likes_count (que por defecto devuelve 0) al final del test:

```
$this->assertEquals(
  0,
  $statusResource['likes_count']
);
```

Y agregamos al StatusResource (app/http/resources) el atributo likes_count:

```
public function toArray($request)
{
  return [
    'id' => $this->id,
    'body' => $this->body,
    'user_name' => $this->user->name,
    'user_avatar' => 'https://aprendible.com/images/default-avatar.jpg',
    'ago' => $this->created_at->diffForHumans(),
    'is_liked' => $this->isLiked(),
    'likes_count' => $this->likesCount()
```

```
    ];
  }
```

Compilar cambios con npm run dev.

[Fin video 25]

Vamos a modificar el estilo del conteo de likes con bootstrap y flexbox.

Vamos al StatusesList (resources/assets/js/components) y modificamos las siguientes clases de bootstrap para darle un formato visual bonito, encerrando el elemento span likes-count en un div con un icono:

```
<div class="card-footer pt-2 d-flex justify-content-between align-items-center">
        <button class="btn btn-link btn-sm" v-if="status.is_liked" dusk="unlike-btn" @click="unlike(status)"><strong><i class="fa fa-thumbs-up text-primary mr-1"></i> TE GUSTA</strong></button>
      <button class="btn btn-link btn-sm" v-else dusk="like-btn" @click="like(status)"><i class="far fa-thumbs-up text-primary mr-1"></i> ME GUSTA</button>

  <div class="text-secondary mr-2">
    <i class="far fa-thumbs-up"></i>
    <span dusk="likes-count">{{ status.likes_count }}</span>
  </div>
</div>
```

[Fin video 26]

Vamos a empezar con los comentarios, comenzamos con un test:

php artisan make:test CreateCommentsTest

Abrimos el CreateCommentsTest en (tests/feature) y añadimos:

```
namespace Tests\Feature;

use App\User;
use Tests\TestCase;
use App\Models\Status;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CreateCommentsTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function authenticated_users_can_comment_statuses()
```

```
  {
      $status = factory(Status::class)->create();
      $user = factory(User::class)->create();
      $comment = ['body' => 'Mi primer comentario'];

      $this->actingAs($user)
          ->postJson(route('statuses.comments.store', $status), $comment);

      $this->assertDatabaseHas('comments', [
          'user_id' => $user->id,
          'status_id' => $status->id,
          'body' => $comment['body']
      ]);
  }
}
```

Explicación: Creamos un estado, un usuario que realice la acción y un comentario. Actuando como el usuario hacemos un postJson a la ruta no creada statuses.comments.store con el estado a comentar y el comentario. Y compruebe en la base de datos que existe la tabla comments y dentro la información del comentario creado (la id del usuario, la id del status y el cuerpo del comentario).

Agregamos la ruta statuses.comments.store al archivo de rutas web.php, con el controlador NO creado todavía StatusCommentsController y el método store:

```
// Statuses Comment routes
Route::post('statuses/{status}/comments', 'StatusCommentsController@store')-
>name('statuses.comments.store')->middleware('auth');
```

Creamos el StatusCommentsController por terminal:

```
php artisan make:controller StatusCommentsController
```

Y agregamos el método store al StatusCommentsController (app/http/controllers) donde creamos el comentario con el modelo Comment todavía no creado:

```
namespace App\Http\Controllers;

use App\Models\Comment;
use App\Models\Status;
use Illuminate\Http\Request;

class StatusCommentsController extends Controller
{
  public function store(Status $status)
  {
      Comment::create([
          'user_id' => auth()->id(),
          'status_id' => $status->id,
          'body' => request('body')
```

```
        ]);
    }
}
```

Creamos la tabla comments con una migración por terminal:

```
php artisan make:migration create_comments_table
```

Creamos el modelo en la carpeta models usando la terminal:

```
php artisan make:model Models/Comment
```

Vamos al modelo Comment (app/models) y deshabilitamos la protección:

```
class Comment extends Model
{
    protected $guarded = [];

}
```

Editamos la migración de la tabla comments (database/migrations) para agregar los campos que hacen falta:

```
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->increments('id');
        $table->unsignedInteger('user_id');
        $table->unsignedInteger('status_id');
        $table->text('body');
        $table->timestamps();
    });
}
```

Ahora vamos a verificar que los usuarios No autenticados no puedan escribir comentarios. Para ello vamos al CreateCommentsTest (tests/feature) y creamos un nuevo test al principio:

```
    /** @test */
    public function guest_users_cannot_create_comments()
    {
        $status = factory(Status::class)->create();
        $comment = ['body' => 'Mi primer comentario'];

        $response = $this->postJson(route('statuses.comments.store', $status), $comment);

        $response->assertStatus(401);
    }
```

Verificamos que la respuesta que recibimos al no estar autenticamos, tiene un código de estado 401 Unauthorized.

Vamos a cambiar los tests CreateStatusTest y CanLikeStatusesTest para que verifique el estado de la respuesta al estar utilizando ajax.
Para el test guests_users_can_not_create_statuses de CreateStatusTest:

```
function guests_users_can_not_create_statuses()
{
  $response = $this->postJson(route('statuses.store'), ['body' => 'Mi primer status']);

  $response->assertStatus(401);
}
```

Y para el test guests_users_can_not_like_statuses de CanLikeStatusesTest:

```
function guests_users_can_not_like_statuses()
{
  $status = factory(Status::class)->create();

  $response = $this->postJson(route('statuses.likes.store', $status));

  $response->assertStatus(401);
}
```

Finalmente ejecutamos los test de phpunit (desde phpstorm) y los de dusk en terminal (php artisan dusk) para comprobar que todo funciona correctamente.

[Fin video 27]

Para mostrar el comentario tras crearlo es necesario devolverlo en la respuesta. Para ello, en el CreateCommentTest (test/feature) guardamos la respuesta recibida al crear un comentario y verificamos que recibimos el comentario como respuesta.

```
public function authenticated_users_can_comment_statuses()
{
  $status = factory(Status::class)->create();
  $user = factory(User::class)->create();
  $comment = ['body' => 'Mi primer comentario'];

  $response = $this->actingAs($user)
    ->postJson(route('statuses.comments.store', $status), $comment);

  $response->assertJson([
    'data' => ['body' => $comment['body']]
  ]);

  $this->assertDatabaseHas('comments', [
    'user_id' => $user->id,
```

```
        'status_id' => $status->id,
        'body' => $comment['body']
    ]);
}
```

Vamos al StatusCommentController (app/http/controllers) y guardamos el comentario en una variable y devolvemos el recurso API (todavía no creado). Además importamos la clase CommentResource.

```
use App\Http\Resources\CommentResource;

(... Importamos la clase y añadimos lo siguiente ...)

public function store(Status $status)
{
    $comment = Comment::create([
        'user_id' => auth()->id(),
        'status_id' => $status->id,
        'body' => request('body')
    ]);

    return CommentResource::make($comment);
}
```

Creamos la clase CommentResource por terminal:

```
php artisan make:resource CommentResource
```

Vamos a la clase CommentResource recién creada y retornamos un array con el cuerpo del comentario:

```
public function toArray($request)
{
    return [
        'body' => $this->body
    ];
}
```

Creamos otro test para el recurso API, usando como modelo el test StatusResourceTest, lo copiamos y modificamos el nombre a CommentResourceTest pegando en la misma carpeta. Luego lo modificamos:

```
namespace Tests\Unit\Http\Resources;

use Tests\TestCase;
use App\Models\Status;
use App\Http\Resources\CommentResource;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CommentResourceTest extends TestCase
{
    use RefreshDatabase;
```

```
    /** @test */
    public function a_comment_resources_must_have_the_necessary_fields()
    {
        $comment = factory(Status::class)->create();

        $commentResource = CommentResource::make($comment)->resolve();

        $this->assertEquals(
            $comment->body,
            $commentResource['body']
        );
    }
}
```

[Fin video 28]

Antes de mostrar la caja de comentarios debajo de cada estado, vamos a refactorizar el código para hacerlo más legible.
Primero cambiamos el nombre del StatusesList (resources/assets/js/components) por StatusList (click derecho, refactorizar en phpstorm).

Modificamos el archivo app.js (resources/assets/js) para cambiar el nombre anterior:

```
Vue.component('status-list', require('./components/StatusList'));
```

Y en la vista welcome (resource/views) cambiamos el nombre también:

```
<status-list></status-list>
```

Creamos un nuevo componente de vue en la carpeta components (resources/assets/js/components) llamado StatusListItem (click derecho en la carpeta y crear nuevo componente de vue con phpstorm). Y extraemos el contenido de la redirección si invitado del StatusList para pasarlo al StatusListItem, quedaría:

```
<template>
    <div class="card border-0 mb-3 shadow-sm">
        <div class="card-body d-flex flex-column">
            <div class="d-flex align-items-center mb-3">
                                        <img class="rounded mr-3 shadow-sm" width="40px"
src="https://aprendible.com/images/default-avatar.jpg" alt="">
                <div class="">
                    <h5 class="mb-1" v-text="status.user_name"></h5>
                    <div class="small text-muted" v-text="status.ago"></div>
                </div>
            </div>
            <p class="card-text text-secondary" v-text="status.body"></p>
        </div>
        <div class="card-footer pt-2 d-flex justify-content-between align-items-center">
            <like-btn
```

```
            :status="status"
        ></like-btn>
        <div class="text-secondary mr-2">
            <i class="far fa-thumbs-up"></i>
            <span dusk="likes-count">{{ status.likes_count }}</span>
        </div>
    </div>
  </div>
</template>

<script>
  import LikeBtn from './LikeBtn'

  export default {
      props: {
          status: {
              type: Object,
              required: true
          }
      },
      components: { LikeBtn }
  }
</script>
```

Explicación: Al hacer referencia a un objeto status, lo recibimos como propiedad y le decimos que sea un objeto para que de error por consola en caso contrario. El status-list-item recibe una propiedad llamada status, para que muestre error al no pasar esta propiedad, definimos que su tipo sea un objeto y que sea requerido.
E importamos el componente LikeBtn que crearemos algo más adelante.

En el StatusList, usamos el v-for en el componente recién creado que lo importamos, y registramos el componente en la instancia de vue:

```
<template>
  <div @click="redirectIfGuest">
      <status-list-item
          v-for="status in statuses"
          :status="status"
          :key="status.id"
      ></status-list-item>
  </div>
</template>

<script>
  import StatusListItem from './StatusListItem'

  export default {
      components: { StatusListItem },
      data() {
        return {
            statuses: []
```

```
        }
    },
    mounted() {
        axios.get('/statuses')
            .then(res => {
                this.statuses = res.data.data
            })
            .catch(err => {
                console.log(err.response.data);
            });

        EventBus.$on('status-created', status => {
            this.statuses.unshift(status);
        })
    }
  }
</script>
```

Explicación: Debemos añadir una llave única al componente status-list-item creado para que vue pueda darle seguimiento y diferenciar los componentes individualmente al hacer bucles.
Ahora el StatusList solo contiene la lógica necesaria para mostrar el estado.

Por último, nos llevamos la lógica para dar me gusta del StatusList (los botones y los metodos like y unlike) a un nuevo componente que lo llamamos LikeBtn que crearemos en la misma carpeta components de la misma forma que antes:

```
<template>
  <button class="btn btn-link btn-sm"
        v-if="status.is_liked"
        dusk="unlike-btn"
        @click="unlike(status)"
  ><strong>
      <i class="fa fa-thumbs-up text-primary mr-1"></i>
      TE GUSTA
  </strong></button>

  <button class="btn btn-link btn-sm"
        v-else dusk="like-btn"
        @click="like(status)"
  ><i class="far fa-thumbs-up text-primary mr-1"></i>
      ME GUSTA</button>
</template>

<script>
  export default {
      props: {
          status: {
              type: Object,
              required: true
          }
```

```
    },
    methods: {
        like(status){
            axios.post(`/statuses/${status.id}/likes`)
                .then(res => {
                    status.is_liked = true;
                    status.likes_count++;
                })
        },
        unlike(status){
            axios.delete(`/statuses/${status.id}/likes`)
                .then(res => {
                    status.is_liked = false;
                    status.likes_count--;
                })
        }
    }
  }
</script>
```

Finalmente comprobamos que las reestructuraciones no han dañado nada, compilando los cambios npm run dev y pasando los test de dusk php artisan dusk .

[Fin video 29]

Vamos a agregar un textarea debajo de cada estado para poder añadir los comentarios. Comenzamos creando un test de dusk en terminal:

php artisan dusk:make UsersCanCommentStatusesTest

Abrimos el test recien (tests/browser) creado y modificamos:

```
namespace Tests\Browser;

use App\Models\Status;
use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UserCanCommentStatusTest extends DuskTestCase
{
  use DatabaseMigrations;

  /** @test */
  public function authenticated_users_can_comment_statuses()
  {
    $status = factory(Status::class)->create();
    $user = factory(User::class)->create();
```

```
        $this->browse(function (Browser $browser) use ($status, $user) {
            $comment = 'Mi primer comentario';

            $browser->loginAs($user)
                    ->visit('/')
                    ->waitForText($status->body)
                    ->type('comment', $comment)
                    ->press('@comment-btn')
                    ->waitForText($comment)
                    ->assertSee($comment)
            ;
        });
    }
}
```

Explicación: Creamos un estado a comentar, creamos un usuario y un comentario. Probamos que el usuario haga login, visitemos el home, esperamos por el texto status->body. Escriba dentro de un campo con id comment-btn, espere por el texto aparezca en pantalla y verifique el texto.

Vamos ahora al StatusListItem (resources/assets/js/components) y añadimos el textarea dentro de un formulario en el footer (dentro de template) :

```
            <div class="text-secondary mr-2">
              <i class="far fa-thumbs-up"></i>
              <span dusk="likes-count">{{ status.likes_count }}</span>
            </div>

            <form @submit.prevent="addComment">
              <textarea name="comment" v-model="newComment"></textarea>
              <button dusk="comment-btn">Enviar</button>
            </form>

            <div v-for="comment in comments">{{ comment.body }}</div>

        </div>
    </div>
</template>
```

Explicación: Añadimos el textarea con un v-model para guardar lo que escriba el usuario. El botón lleva el id comment-btn definido en el test. Encerramos los componentes en un form para al enviarlo llamar al método addComment (añadir prevent para que no recargue la página al enviar). Debajo del formulario,

Debajo del template, en el script del StatusListItem modificamos:

```
<script>
  import LikeBtn from './LikeBtn'
```

```
    export default {
        props: {
            status: {
                type: Object,
                required: true
            }
        },
        components: { LikeBtn },
        data() {
            return {
                newComment: '',
                comments: []
            }
        },
        methods: {
            addComment() {
                axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
                    .then(res => {
                        this.newComment = '';
                        this.comments.push(res.data.data);
                    })
            }
        }
    }
</script>
```

Explicación: Definimos la información escrita antes en el formulario (del template) añadiendoles al data(). Y añadiendo el método addComment haciendo el post a la url de los comentarios, y al tener la respuesta que agregue el comentario a la variable de los comentarios y cambie el contenido del textarea a vacío.

Compilamos los cambios realizados: npm run dev
Verificamos que el test está funcionando ejecutando el test en phpstorm.
Y lanzamos las migraciones, php artisan migrate , antes de comprobar en el navegador que efectivamente podemos realizar un comentario a un estado.

[Fin video 30]

Vamos a hacer que los comentarios persistan y se muestren debajo de cada estado.

Empezamos añadiendo un nuevo método al test UsersCanCommentStatusTest (tests/browser) para probar nuevas características, en vez de crear un nuevo test.

```
/** @test */
public function users_can_see_all_comments()
{
```

```
    $status = factory(Status::class)->create();
    $comments = factory(Comment::class, 2)->create(['status_id' => $status->id]);

    $this->browse(function (Browser $browser) use ($status, $comments) {

        $browser->visit('/')
                ->waitForText($status->body)
                ->assertSee($comments->shift()->body)
                ->assertSee($comments->shift()->body)

        ;
    });
}
```

Explicación: Creamos un estado y comentarios. Testeamos que visitamos el home, esperamos por el texto, y comprobamos que vemos el body del primer comentario. Usamos shift() porque devuelve el primer comentario de la colección y luego lo elimina, así obtenemos el primer y segundo comentario. No olvidar añadir `use App\Models\Comment;` al principio del test.

Creamos el factory para el modelo Comment por terminal:

```
php artisan make:factory CommentFactory -m Models\\Comment
```

Vamos al CommentFactory (database/factories) recién creado y añadimos el user_id que sea igual a una función que retorna un usuario creado. Y creamos un body :

```
use App\User;
use Faker\Generator as Faker;

$factory->define(App\Models\Comment::class, function (Faker $faker) {
    return [
        'body' => $faker->paragraph,
        'user_id' => function() {
            return factory(User::class)->create();
        }
    ];
});
```

Agregamos la verificación de otros campos en el StatusResourceTest (tests/unit/http/resources), agregando un comentario con la id del estado creado, al principio del test e importando la clase Comment al test:

```
public function a_status_resources_must_have_the_necessary_fields()
{
    $status = factory(Status::class)->create();
    factory(Comment::class)->create(['status_id' => $status->id]);

    $statusResource = StatusResource::make($status)->resolve();
```

(... Y añadimos al final la verificación de los comentarios y el tipo de instancia para comprobar que devolvemos un objeto CommentResource …)

```
// dd($statusResource['comments']->first()->resource); //para comprobar la respuesta obtenida
$this->assertEquals(
  CommentResource::class,
  $statusResource['comments']->collects
);
$this->assertInstanceOf(
  Comment::class,
  $statusResource['comments']->first()->resource
);
```

Definimos el campo de los comentarios en el StatusResource (app/http/resources) añadiendo en el return al final. De esta forma si pasamos otra cosa que no sea una colección de comentarios del recurso Comment, fallará.:

```
'comments' => CommentResource::collection($this->comments)
```

Vamos al StatusTest (tests/unit/models) para definir la relación entre el estado y los comentarios.
De forma similar al metodo a_status_has_many_likes que comprueba la relación con los likes, añadimos entre este test y el anterior a_status_belongs_to_a_user lo siguiente:

```
/** @test */
function a_status_has_many_comments()
{
  $status = factory(Status::class)->create();

  factory(Comment::class)->create(['status_id' => $status->id]);

  $this->assertInstanceOf(Comment::class, $status->comments->first());
}
```

Vamos al modelo Status (app/models) y entre el método user() y el de likes() añadimos el nuevo método para los comentarios:

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

Finalmente para poder ver el comentario, vamos al StatusListItem y en vez de inicializar los comentarios a un array vacío, los obtenemos del statusComment:

```
data() {
  return {
    newComment: '',
    comments: this.status.comments
  }
```

```
    },
```

Compilamos los cambios: npm run dev
Y comprobamos que pasan todos los tests: php artisan dusk

[Fin video 31]

Vamos a verificar que cada comentario muestre el nombre de la persona que lo escribió. Comenzamos modificando el test de dusk UsersCanCommentStatusTest para que por cada comentario, verificamos que vemos el body y el user name:

```
/** @test */
public function users_can_see_all_comments()
{
  $status = factory(Status::class)->create();
  $comments = factory(Comment::class, 2)->create(['status_id' => $status->id]);

  $this->browse(function (Browser $browser) use ($status, $comments) {

    $browser->visit('/')->waitForText($status->body);

    foreach ($comments as $comment) {
      $browser->assertSee($comment->body)
          ->assertSee($comment->user->name)
      ;
    }
  });
}
```

Creamos un test unitario que verifique la propiedad user del comentario, en terminal:
php artisan make:test Models/CommentTest --unit

Abrimos el test CommentTest (tests/unit/models) recien creado, y verificamos que un comentario pertenece a un usuario. Por ello, creamos un comentario y verificamos que se recibe una instancia de la clase App\User al llamar a comment user:

```
namespace Tests\Unit\Models;

use App\User;
use Tests\TestCase;
use App\Models\Comment;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CommentTest extends TestCase
{
```

```
    use RefreshDatabase;

    /** @test */
    public function a_comment_belongs_to_a_user()
    {
        $comment = factory(Comment::class)->create();

        $this->assertInstanceOf(User::class, $comment->user);
    }
}
```

Al crear el comentario, no estamos pasando el status id.
Abrimos el ComentFactory (database/factory) para añadirlo:

```
use App\Models\Status;
use App\User;
use Faker\Generator as Faker;

$factory->define(App\Models\Comment::class, function (Faker $faker) {
    return [
        'body' => $faker->paragraph,
        'user_id' => function() {
            return factory(User::class)->create();
        },
        'status_id' => function() {
            return factory(Status::class)->create();
        }
    ];
});
```

Vamos al modelo Comment (app/models) y agregamos el modelo User:

```
use App\User;
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    protected $guarded = [];

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

Vamos al componente StatusListItem (resources/assets/js/components) y añadimos el comment.user_name para que lo imprima en pantalla:

```
<form @submit.prevent="addComment">
            <textarea name="comment" v-model="newComment"></textarea>
            <button dusk="comment-btn">Enviar</button>
```

```
        </form>

        <div v-for="comment in comments">
            {{ comment.user_name }}
            {{ comment.body }}
        </div>

    </div>
  </div>
</template>
```

Ahora vamos al CommentResourceTest (tests/unit/http/resources) para verificar que recibimos el user name al llamar al CommentResource, y recibimos el avatar:

```
 public function a_comment_resources_must_have_the_necessary_fields()
  {
    $comment = factory(Status::class)->create();

    $commentResource = CommentResource::make($comment)->resolve();

    $this->assertEquals(
      $comment->body,
      $commentResource['body']
    );

    $this->assertEquals(
      $comment->user->name,
      $commentResource['user_name']
    );

    $this->assertEquals(
      'https://aprendible.com/images/default-avatar.jpg',
      $commentResource['user_avatar']
    );
  }
}
```

Seguidamente vamos al CommentResource (app/http/resources) para añadir el user_name y el avatar:

```
public function toArray($request)
{
  return [
    'body' => $this->body,
    'user_name' => $this->user->name,
    'user_avatar' => 'https://aprendible.com/images/default-avatar.jpg'
  ];
}
```

Compilamos los cambios: npm run dev
Y comprobamos que pasan todos los tests: php artisan dusk

[Fin video 32]

Vamos a mejorar el estilo visual de los comentarios y caja de comentarios utilizando Bootstrap. Abrimos StatusListItem (resources/assets/js/components) y modificamos:

```
<template>
  <div class="card border-0 mb-3 shadow-sm">
    <div class="card-body d-flex flex-column">
      <div class="d-flex align-items-center mb-3">
                                          <img class="rounded mr-3 shadow-sm" width="40px"
src="https://aprendible.com/images/default-avatar.jpg" alt="">
        <div class="">
          <h5 class="mb-1" v-text="status.user_name"></h5>
          <div class="small text-muted" v-text="status.ago"></div>
        </div>
      </div>
      <p class="card-text text-secondary" v-text="status.body"></p>
    </div>
    <div class="card-footer pt-2 d-flex justify-content-between align-items-center">
      <like-btn
        :status="status"
      ></like-btn>
      <div class="text-secondary mr-2">
        <i class="far fa-thumbs-up"></i>
        <span dusk="likes-count">{{ status.likes_count }}</span>
      </div>
    </div>

    <div class="card-footer">

      <div v-for="comment in comments" class="mb-3">
                                          <img class="rounded shadow-sm float-left mr-2"
width="34px" :src="comment.user_avatar" :alt="comment.user_name">
        <div class="card border-0 shadow-sm">
          <div class="card-body p-2 text-secondary">
            <a href=""><strong>{{ comment.user_name }}</strong></a>
            {{ comment.body }}
          </div>
        </div>
      </div>

      <form @submit.prevent="addComment" v-if="isAuthenticated">
        <div class="d-flex align-items-center ">
                                    <img class="rounded shadow-sm float-left mr-2" width="34px"
src="https://aprendible.com/images/default-avatar.jpg" :alt="currentUser.name">
          <div class="input-group">
                                <textarea class="form-control border-0 shadow-sm" name="comment" v-
model="newComment" placeholder="Escribe un comentario.." rows="1"></textarea>
```

```
                    <div class="input-group-append">
                        <button class="btn btn-primary" dusk="comment-btn">Enviar</button>
                    </div>
                </div>
            </div>
        </form>

    </div>
  </div>
</template>
```

Explicación: El card footer anterior encierra a los botones de like. Creamos un card-footer para los comentarios. Ubicamos el formulario después de los comentarios y modificamos la parte de comentarios y formulario.

En los comentarios, agregamos clases de bootstrap para darles formato, colocamos una imagen fuera de la tarjeta con la imagen del avatar.

En el formulario, hacemos que solo se muestre si el usuario está autenticado, añadimos una imagen del usuario autenticado (actualmente la imagen por defecto) y agregamos clases de bootstrap usando flexbox.

Compilamos los cambios: npm run dev .
Y verificamos que todo funciona OK en el navegador.

[Fin video 33]

Vamos a crear la validación de los comentarios. Para ello primero vamos al CreateCommentsTest (tests/feature) y creamos un nuevo test al final:

```
/** @test */
function a_comment_requires_a_body()
{
  $status = factory(Status::class)->create();
  $user = factory(User::class)->create();
  $this->actingAs($user);

  $response = $this->postJson(route('statuses.comments.store', $status),['body' => '']);

  $response->assertStatus(422);

  $response->assertJsonStructure([
      'message', 'errors' => ['body']
  ]);
}
```

Explicación: Siguiendo la estructura del test a_status_requires_a_body del CreateStatusTest lo modificamos añadiendo un estado y modificando la ruta.
De esta forma cuando hacemos login y tratamos de crear un comentario con el cuerpo vacio, verificamos que recibimos el estado 422 y la estructura de error.

Vamos al StatusCommentsController (app/http/controllers) y añadimos la validacion:

```
public function store(Status $status)
{
  request()->validate([
     'body' => 'required'
  ]);

  $comment = Comment::create([
    'user_id' => auth()->id(),
     'status_id' => $status->id,
     'body' => request('body')
  ]);

  return CommentResource::make($comment);
}
```

A continuación, añadimos un console.log de la respuesta al StatusListItem (resources/assets/js/components), de la misma manera que en el StatusForm:

```
methods: {
  addComment() {
     axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
        .then(res => {
           this.newComment = '';
           this.comments.push(res.data.data);
        })
        .catch(err => {
           console.log(err.response.data)
        })
  }
}
```

Por último, añadimos la propiedad required al textarea del formulario para crear comentarios del StatusListitem. Y otro required al textarea del formulario de crear estados del StatusForm.

Finalmente, compilamos los cambios: npm run dev .
Y verificamos que en el navegador al no poner nada en el formulario de comentarios y enviarlo, por consola aparezca un error 422.

[Fin video 34]

Vamos a implementar los likes a los comentarios. Empezamos creando un test:

```
php artisan make:test CanLikeCommentsTest
```

Este test sera muy similar al test CanLikeStatusesTest, por lo que lo usaremos como plantilla para crear los nuevos tests.
Empezamos añadiendo y modificando los 2 primeros tests para comprobar que invitados no puedan hacer like y que un usuario autenticado si pueda dar like.

```
namespace Tests\Feature;

use App\Models\Comment;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanLikeCommentsTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function guests_users_can_not_like_comments()
  {
     $comment = factory(Comment::class)->create();

     $response = $this->postJson(route('comments.likes.store', $comment));

     $response->assertStatus(401);
  }


  /** @test */
  function an_authenticated_user_can_like_comments()
  {
     $this->withoutExceptionHandling();

     $user = factory(User::class)->create();
     $comment = factory(Comment::class)->create();

     $this->actingAs($user)->postJson( route('comments.likes.store', $comment));

     $this->assertDatabaseHas('likes', [
        'user_id' => $user->id,
        'status_id' => $comment->id

     ]);

  }
}
```

Explicación: En la verificación del segundo test, verificamos un user_id y un status_id , pero en este caso no estamos utilizando un status_id . Queremos vincular el like a un comentario. Actualmente deberíamos ir a la migración de likes y añadir un nuevo campo para cada relación, pero no es óptimo. Por ello utilizaremos polimorfismo en siguientes lecciones.

Vamos al archivo de rutas web.php (routes) para añadir la nueva ruta de likes a comentarios recién definida en el test. Añadimos debajo de las rutas anteriores:

```
// Comments Like routes
Route::post('comments/{comment}/likes', 'CommentLikesController@store')->name('comments.likes.store')->middleware('auth');
```

Creamos el controlador de los likes de comentarios en terminal:

```
php artisan make:controller CommentLikesController
```

Vamos al controlador creado y añadimos el método store de momento vacío:

```
public function store()
{

}
```

[Fin video 35]

Continuamos como en la lección anterior, creando los likes de forma polimórfica.

Vamos al CanLikeStatusesTest (tests/feature) para realizar reestructuraciones.
Unimos el test de un usuario autenticado puede dar like, con el de quitar like de la siguiente forma y eliminamos los 2 tests anteriores por el nuevo test unión de ambos:

```
/** @test */
function an_authenticated_user_can_like_and_unlike_statuses()
{
  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();

  $this->assertCount(0, $status->likes);

  $this->actingAs($user)->postJson( route('statuses.likes.store', $status));

  $this->assertCount(1, $status->fresh()->likes);

  $this->assertDatabaseHas('likes', ['user_id' => $user->id]);

  // Validacion de quitar like del test an_authenticated_user_can_unlike_statuses borrado
```

```
    $this->actingAs($user)->deleteJson( route('statuses.likes.destroy', $status));

    $this->assertCount(0, $status->fresh()->likes);

    $this->assertDatabaseMissing('likes', ['user_id' => $user->id]);
}
```

Explicación: El assertCount compara al valor dado con la llamada al método count() del parámetro que le pasamos (compara si hay 0 o 1 like). Verificamos que recibimos 0 o 1 al contar los likes, que la base de datos en la tabla users exista un registro con el user_id. Y que al hacer el post para eliminar el like, el número de likes sea 0, y que en la base de datos falte el campo user_id en la tabla likes.

Al quitar las referencias al status_id, podemos hacer una transición al polimorfismo sin que los tests fallen.

Hacemos ahora algo similar a lo anterior en el CanLikeCommentsTest, uniendo el codigo de los tests de like y unlike en un nuevo test:

```
/** @test */
function an_authenticated_user_can_like_and_unlike_comments()
{
    // Para que phpunit ignore el test al ejecutarse: $this->markTestIncomplete();
    $this->withoutExceptionHandling();

    $user = factory(User::class)->create();
    $comment = factory(Comment::class)->create();

    $this->assertCount(0, $comment->likes);

    $this->actingAs($user)->postJson( route('comments.likes.store', $comment));

    $this->assertCount(1, $comment->fresh()->likes);

    $this->assertDatabaseHas('likes', ['user_id' => $user->id]);

    // Comprobamos que podemos quitar likes
    $this->actingAs($user)->deleteJson( route('comments.likes.destroy', $comment));

    $this->assertCount(0, $comment->fresh()->likes);

    $this->assertDatabaseMissing('likes', ['user_id' => $user->id]);
}
```

Implementamos que la relación de likes de los comentarios sea contable para que funcione el assertCount en los tests.
Abrimos el CommentTest (tests/unit/models) y usando como plantilla el test a_status_has_many_likes del StatusTest, creamos un nuevo test :

```
/** @test */
function a_comment_morph_many_likes()
{
  $comment = factory(Comment::class)->create();

  factory(Like::class)->create([
     'likeable_id' => $comment->id,          // Id del comentario: 1
     'likeable_type' => get_class($comment), // Se guardara: App\\Models\\Comment

  ]);

  $this->assertInstanceOf(Like::class, $comment->likes->first());
}
```

Explicación: Usamos la llave likeable_id que será la id del comentario, y likeable_type que será igual al nombre de clase del comentario.

Añadimos los campos likeable a la migración de los likes create_likes_table para poder usar polimorfismo:

```
public function up()
{
  Schema::create('likes', function (Blueprint $table) {
     $table->increments('id');
     $table->unsignedInteger('user_id');
     $table->unsignedInteger('status_id')->nullable();
     $table->nullableMorphs('likeable'); // crea campos para polimorfismo
     $table->timestamps();
  });
}
```

Explicacion: Al llamar al método Morphs, recibe el nombre likeable y lo usa para crear el likeable_id y el likeable_type. Hacemos nullable el status_id y el morph para que otros test no fallen al no estar definiendolos en el StatusTest.

Implementamos la relación likes de los comentarios en el modelo Comment (app/models) añadiendo el método likes al final:

```
public function likes()
{
  return $this->morphMany(Like::class, 'likeable');
}
```

Vamos al CommentLikesController (app/http/controllers) para realizar la implementacion del metodo store y destroy para crear y quitar likes:

```
namespace App\Http\Controllers;

use App\Models\Comment;
```

```
class CommentLikesController extends Controller
{
  public function store(Comment $comment)
  {
     $comment->likes()->create([
        'user_id' => auth()->id()
     ]);
  }

  public function destroy(Comment $comment)
  {
     $comment->likes()->where([
        'user_id' => auth()->id()
     ])->delete();
  }
}
```

Explicacion: Realizamos una reestructuración de código llamando al objeto comment, accediendo a los likes (polimórficos) y creandolos o destruyendolos usando solo el user_id, al verificar Laravel automáticamente los campos likeable.

Creamos la ruta para quitar los likes en el archivo de rutas web.php:

```
// Comments Like routes
Route::post('comments/{comment}/likes', 'CommentLikesController@store')->name('comments.likes.store')->middleware('auth');
Route::delete('comments/{comment}/likes', 'CommentLikesController@destroy')->name('comments.likes.destroy')->middleware('auth');
```

[Fin video 36]

Vamos a implementar la relacion polimórfica con los estados.
Para ello vamos a quitar los campos nullable de la migración de la tabla de crear likes, y que todos los test pasen correctamente.

Antes de esto, vamos al StatusTest y modificamos el test a_status_has_many_likes siguiendo el ejemplo del test a_comment_morph_many_likes del CommentTest:

```
/** @test */
function a_status_morph_many_likes()
{
  $status = factory(Status::class)->create();

  factory(Like::class)->create([
     'likeable_id' => $status->id,          // 1
     'likeable_type' => get_class($status), // App\\Models\\Status

  ]);
```

```
    $this->assertInstanceOf(Like::class, $status->likes->first());
}
```

Reemplazamos el método likes() del modelo Status (app/models) por el mismo modelo likes() del modelo Comment, quedaría:

```
public function likes()
{
    return $this->morphMany(Like::class, 'likeable');
}
```

Vamos al StatusTest (tests/unit/models) y modificamos el test a_status_knows_how_many_likes_it_has para que no utilice el status_id, quedaría:

```
function a_status_knows_how_many_likes_it_has()
{
    $status = factory(Status::class)->create();

    $this->assertEquals(0, $status->likesCount());

    factory(Like::class, 2)->create([
        'likeable_id' => $status->id,          // 1
        'likeable_type' => get_class($status), // App\\Models\\Status

    ]);

    $this->assertEquals(2, $status->likesCount());
}
```

Como el CommentTest va a tener muchas funcionalidades parecidas al StatusTest, vamos a copiar los tests que prueban los métodos del Status.
Para ello, vamos al StatusTest y copiamos los siguientes tests:
a_status_can_be_liked_and_unlike()
a_status_can_be_liked_once()
a_status_knows_if_it_has_been_liked()
a_status_knows_how_many_likes_it_has()
Y los pegamos en el CommentTest, y los modificamos:

```
/** @test */
function a_comment_can_be_liked_and_unlike()
{
    $comment = factory(Comment::class)->create();

    $this->actingAs( factory(User::class)->create() );

    $comment->like();

    $this->assertEquals(1, $comment->fresh()->likes->count());
```

```
  $comment->unlike();

  $this->assertEquals(0, $comment->fresh()->likes->count());

}

/** @test */
function a_comment_can_be_liked_once()
{
  $comment = factory(Comment::class)->create();

  $this->actingAs( factory(User::class)->create() );

  $comment->like();

  $this->assertEquals(1, $comment->likes->count());

  $comment->like();

  $this->assertEquals(1, $comment->fresh()->likes->count());

}

/** @test */
function a_comment_knows_if_it_has_been_liked()
{
  $comment = factory(Comment::class)->create();

  $this->assertFalse($comment->isLiked());

  $this->actingAs(factory(User::class)->create());

  $this->assertFalse($comment->isLiked());

  $comment->like();

  $this->assertTrue($comment->isLiked());
}

/** @test */
function a_comment_knows_how_many_likes_it_has()
{
  $comment = factory(Comment::class)->create();

  $this->assertEquals(0, $comment->likesCount());

  factory(Like::class, 2)->create([
    'likeable_id' => $comment->id,          // 1
    'likeable_type' => get_class($comment), // App\\Models\\Status

  ]);

  $this->assertEquals(2, $comment->likesCount());
```

```
}
```

Añadimos los métodos para dar like, quitar like, preguntar si tiene like, y para contar los likes al modelo Comment. Estos métodos son iguales que en el modelo Status:

```
public function like()
{
   $this->likes()->firstOrCreate([
       'user_id' => auth()->id()
   ]);
}

public function unlike()
{
   $this->likes()->where([
       'user_id' => auth()->id()
   ])->delete();
}

public function isLiked()
{
   return $this->likes()->where('user_id', auth()->id())->exists();
}

public function likesCount()
{
   return $this->likes()->count();
}
```

Ahora ya podemos quitar sin romper nada el campo status_id de la migración de la tabla de crear likes (create_likes_table), y no permitimos que los morphs sean nulos cambiandolos a: $table->morphs('likeable');
También quitamos del LikeFactory (database/factories) el status_id entero (las 3 líneas de código).

Por último comprobamos que los tests de phpunit pasan.
Y los test de dusk también: php artisan dusk

[Fin video 37]

Un Trait es un mecanismo de Php para encapsular y reutilizar métodos.

Vamos a utilizar un trait para utilizar métodos que están repetidos en los modelos Comment y Status. Para ello creamos un modelo que modificaremos para que sea

un trait en una nueva carpeta traits, escribimos por terminal:

php artisan make:model Traits/HasLikes

Vamos al trait creado (app/traits) y añadimos los métodos repetidos:

```
namespace App\Traits;

use App\Models\Like;

trait HasLikes
{
  public function likes()
  {
    return $this->morphMany(Like::class, 'likeable');
  }

  public function like()
  {
    $this->likes()->firstOrCreate([
      'user_id' => auth()->id()
    ]);
  }

  public function unlike()
  {
    $this->likes()->where([
      'user_id' => auth()->id()
    ])->delete();
  }

  public function isLiked()
  {
    return $this->likes()->where('user_id', auth()->id())->exists();
  }

  public function likesCount()
  {
    return $this->likes()->count();
  }
}
```

En el modelo Comment eliminamos métodos repetidos y añadimos el trait:

```
namespace App\Models;

use App\Traits\HasLikes;
use App\User;
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
```

```
    use HasLikes;

    protected $guarded = [];

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

Y en el modelo Status hacemos lo mismo, quitando métodos y añadiendo el trait:

```
namespace App\Models;

use App\Traits\HasLikes;
use App\User;
use Illuminate\Database\Eloquent\Model;

class Status extends Model
{
    use HasLikes;

    protected $guarded = [];

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

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

[Fin video 38]

Actualmente hay mucha duplicación con los tests de los likes, para ello vamos a unificar los test de los likes en un nuevo test unitario que verifique el trait HasLikes.

En la terminal: php artisan make:test Traits/HasLikesTest --unit

Vamos al HasLikesTest (tests/unit/traits) recién creado y vamos pegando los tests que verifican propiedades de los likes (los que se encuentran en CommentTest y StatusTest), los modificamos y luego eliminamos los tests originales copiados de CommentTest y StatusTest.
El test del trait (HasLikesTest) quedaría:

```php
namespace Tests\Unit\Traits;

use App\Models\Like;
use App\Traits\HasLikes;
use App\User;
use Illuminate\Database\Eloquent\Model;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class HasLikesTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function a_model_morph_many_likes()
  {
     $model = new ModelWithLikes(['id' => 1]);

     factory(Like::class)->create([
        'likeable_id' => $model->id,          // Id del comentario: 1
        'likeable_type' => get_class($model), // Se guardara: App\\Models\\Comment

     ]);

     $this->assertInstanceOf(Like::class, $model->likes->first());
  }

  /** @test */
  function a_model_can_be_liked_and_unlike()
  {
     $model = new ModelWithLikes(['id' => 1]);

     $this->actingAs( factory(User::class)->create() );

     $model->like();

     $this->assertEquals(1, $model->likes()->count());

     $model->unlike();

     $this->assertEquals(0, $model->likes()->count());

  }

  /** @test */
  function a_model_can_be_liked_once()
  {
     $model = new ModelWithLikes(['id' => 1]);

     $this->actingAs( factory(User::class)->create() );

     $model->like();
```

```
        $this->assertEquals(1, $model->likes()->count());

        $model->like();

        $this->assertEquals(1, $model->likes()->count());

    }

    /** @test */
    function a_model_knows_if_it_has_been_liked()
    {
        $model = new ModelWithLikes(['id' => 1]);

        $this->assertFalse($model->isLiked());

        $this->actingAs(factory(User::class)->create());

        $this->assertFalse($model->isLiked());

        $model->like();

        $this->assertTrue($model->isLiked());
    }

    /** @test */
    function a_model_knows_how_many_likes_it_has()
    {
        $model = new ModelWithLikes(['id' => 1]);

        $this->assertEquals(0, $model->likesCount());

        factory(Like::class, 2)->create([
            'likeable_id' => $model->id,          // 1
            'likeable_type' => get_class($model), // App\\Models\\Status

        ]);

        $this->assertEquals(2, $model->likesCount());
    }
}

class ModelWithLikes extends Model
{
    use HasLikes;

    protected $fillable = ['id'];
}
```

Explicación: Creamos una nueva clase ModelWithLikes al final, porque para testear un trait debemos usar un nuevo modelo que solo usaremos para el test.
Al copiar los tests, vamos modificandolos para que usen una instancia del modelo

ModelWithLikes definido para probar el trait, modificando el nombre de variables a la del modelo. Y cambiando el nombre del test de comment o status, a modelo.
Los cambios pueden verse más claramente al observar las partes que he dejado marcadas en amarillo arriba.

Finalmente borramos los tests originales de CommentTest y StatusTest.
El CommentTest quedaría de la siguiente forma:

```
namespace Tests\Unit\Models;

use App\Traits\HasLikes;
use App\User;
use Tests\TestCase;
use App\Models\Comment;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CommentTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function a_comment_belongs_to_a_user()
    {
        $comment = factory(Comment::class)->create();

        $this->assertInstanceOf(User::class, $comment->user);
    }
}
```

Y el StatusTest quedaría:

```
namespace Tests\Unit\Models;

use App\Models\Comment;
use App\Traits\HasLikes;
use App\User;
use Tests\TestCase;
use App\Models\Status;
use Illuminate\Foundation\Testing\RefreshDatabase;

class StatusTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    function a_status_belongs_to_a_user()
    {
        $status = factory(Status::class)->create();

        $this->assertInstanceOf(User::class, $status->user);
    }
```

```
    /** @test */
    function a_status_has_many_comments()
    {
        $status = factory(Status::class)->create();

        factory(Comment::class)->create(['status_id' => $status->id]);

        $this->assertInstanceOf(Comment::class, $status->comments->first());
    }
}
```

Ahora tenemos que hacer que si quitamos el trait HasLikes del modelo Comment (app/models) o del modelo Status, los tests que verifican ambos modelos (CommentTest y StatusTest) dan error al no tener el trait.
Es decir, que sea necesario el uso del trait HasLikes para que los tests CommentTest y StatusTest pasen Ok.

Para ello, vamos a crear un test que verifique lo anterior.
En el CommentTest añadimos el nuevo test al final:

```
/** @test */
public function a_comment_model_must_use_the_trait_has_likes()
{
    $this->assertClassUsesTrait(HasLikes::class, Comment::class);
}
```

Explicación: Usamos un nuevo método que verifica que la case usa un trait, pasando como parámetros el trait y la clase a verificar. Verificamos que el array contenga una llave \App\Traits\HasLikes en el array que pasamos para verificar (los traits que utiliza la clase a verificar).

Este nuevo método lo tenemos que crear nosotros, por ello vamos al TestCase (tests) y añadimos el nuevo método `assertClassUsesTrait` con un mensaje de error personalizado para mostrar mas claramente el error:

```
protected function assertClassUsesTrait($trait, $class)
{
    $this->assertArrayHasKey(
        $trait,
        class_uses($class),
        "{$class} must use {$trait} trait"
    );
}
```

Finalmente, agregamos al StatusTest el mismo test que hemos agregado al CommentTest. Al final añadimos:

```
/** @test */
public function a_comment_model_must_use_the_trait_has_likes()
{
  $this->assertClassUsesTrait(HasLikes::class, Status::class);
}
```

Así cada vez que necesitemos añadir likes a una, le añadimos el trait HasLikes.
Por último comprobamos que los tests de phpunit pasan.
Y los test de dusk también: php artisan dusk

[Fin video 39]

Para solucionar el problema SQLSTATE[HY000] [1049] Unknown database social_test de que no encuentra la base de datos social_test al ejecutar los tests de dusk, dentro de vagrant ssh ir a la consola de mysql (mysql -u root) y comprobar las bases de datos que hay (show databases;)
Como no estará creada la base de datos social_test , la creamos (create database social_test;).
Y ya deberían pasar los tests de dusk.
También es posible que el error sea por la cache, para limpiarla: php artisan cache:clear

Vamos a agregar los tests que prueben la funcionalidad de los likes de comentarios.

Comenzamos creando un test de dusk:
php artisan dusk:make UsersCanLikeCommentsTest

Y siguiendo como ejemplo el test users_can_like_and_unlike_statuses del UsersCanLikeStatusesTest, creamos un nuevo test en el archivo creado.
El UsersCanLikeCommentsTest quedaria asi:

```
namespace Tests\Browser;

use App\Models\Comment;
use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanLikeCommentsTest extends DuskTestCase
{
  use DatabaseMigrations;
  /**
   * @test
   * @throws \Throwable
```

```
     */
    public function users_can_like_and_unlike_comment()
    {
        $user = factory(User::class)->create();
        $comment = factory(Comment::class)->create();

        $this->browse(function (Browser $browser) use ($user, $comment){
            $browser->loginAs($user)
                ->visit('/')
                ->waitForText($comment->body)
                ->assertSeeIn('@comment-likes-count', 0)
                ->press('@comment-like-btn')
                ->waitForText('TE GUSTA')
                ->assertSee('TE GUSTA')
                ->assertSeeIn('@comment-likes-count', 1)

                ->press('@comment-unlike-btn')
                ->waitForText('ME GUSTA')
                ->assertSee('ME GUSTA')
                ->assertSeeIn('@comment-likes-count', 0)
            ;
        });
    }
}
```

Explicación: Estando autenticado como el usuario, visita el home, espera el cuerpo del comentario, verificamos que dentro del elemento comment-likes-count no haya likes. Luego que al presionar dar like muestre el texto ME GUSTA e incremente el número de likes. Y que al dar otra vez click al botón quite el like.

Vamos al StatusListItem.vue y agregamos los botones en el card-footer de los comentarios y los métodos para dar like y quitarlo en la parte de script de vue:

```
<div class="card-footer">
    <div v-for="comment in comments" class="mb-3">
                                    <img class="rounded shadow-sm float-left mr-2"
width="34px" :src="comment.user_avatar" :alt="comment.user_name">
        <div class="card border-0 shadow-sm">
            <div class="card-body p-2 text-secondary">
                <a href=""><strong>{{ comment.user_name }}</strong></a>
                {{ comment.body }}
            </div>
        </div>

        <span dusk="comment-likes-count">{{ comment.likes_count }}</span>
                                <button v-if="comment.is_liked" dusk="comment-unlike-btn"
@click="unlikeComment(comment)">TE GUSTA</button>
        <button v-else dusk="comment-like-btn" @click="likeComment(comment)">ME GUSTA</
button>

    </div>
```

```
(... Y luego en el script añadimos los métodos para dar/quitar like ...)

methods: {
  addComment() {
    axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
      .then(res => {
        this.newComment = '';
        this.comments.push(res.data.data);
      })
      .catch(err => {
        console.log(err.response.data)
      })
  },
  likeComment(comment) {
    axios.post(`/comments/${comment.id}/likes`)
      .then(res => {
        comment.likes_count ++;
        comment.is_liked = true;
      })
      .catch(err => {
        console.log(err.response.data)
      })
  },
  unlikeComment(comment) {
    axios.delete(`/comments/${comment.id}/likes`)
      .then(res => {
        comment.likes_count --;
        comment.is_liked = false;
      })
      .catch(err => {
        console.log(err.response.data)
      })
  }
```

Explicación: Mostramos el elemento de dusk comment-likes-count con el conteo de likes del comentario. Creamos 2 botones, uno se mostrara solo para dar el like y el otro para quitarlo. En la parte de los métodos creamos el método likeComment, que al hacer un post a la ruta de los likes de comentarios, incremente el número de likes del comentario y ponga la propiedad is_liked en verdadero.
Esta propiedad is_liked se usa para mostrar el botón correcto (dar o quitar like).
Y luego creamos el método unlikeComment de forma similar al anterior, pero borrando el like y decrementando el número de likes y la propiedad is_liked en falso.

Como es necesario agregar los campos id, likes_count y is_liked al objeto comment, vamos primero al CommentResourceTest para agregar estos campos al test de validación del CommentResource. De esta forma añadimos:

```
public function a_comment_resources_must_have_the_necessary_fields()
{
```

```
    $comment = factory(Status::class)->create();

    $commentResource = CommentResource::make($comment)->resolve();

    $this->assertEquals(
        $comment->id,
        $commentResource['id']
    );

    $this->assertEquals(
        $comment->body,
        $commentResource['body']
    );

    $this->assertEquals(
        $comment->user->name,
        $commentResource['user_name']
    );

    $this->assertEquals(
        'https://aprendible.com/images/default-avatar.jpg',
        $commentResource['user_avatar']
    );

    $this->assertEquals(
        0,
        $commentResource['likes_count']
    );

    $this->assertEquals(
        false,
        $commentResource['is_liked']
    );

}
```

Y en el CommentResource añadimos los campos también. Los métodos likesCount y isLiked los utilizamos al añadir el trait HasLikes:

```
public function toArray($request)
{
    return [
        'id' => $this->id,
        'body' => $this->body,
        'user_name' => $this->user->name,
        'user_avatar' => 'https://aprendible.com/images/default-avatar.jpg',
        'likes_count' => $this->likesCount(),
        'is_liked' => $this->isLiked()
    ];
}
```

Ejecutamos el CommentResourceTest para comprobar que pasa OK.
Y ejecutamos el test de dusk creado al principio para comprobar que pasa:

```
php artisan dusk --filter users_can_like_and_unlike_comment
```

[Fin video 40]

Vamos a hacer que el componente like-btn del StatusListItem sea reutilizable, para poder utilizarlo también para dar like a los comentarios. Para ello necesitamos quitar toda referencia a los estados en el like-btn.

Realizamos cambios en el archivo LikeBtn (resources/assets/js/components) para quedarnos con que un solo botón haga la funcion de los 2 anteriores:

```
<template>
  <button
    @click="toggle()"
    :class="getBtnClasses"
  ><i :class="getIconClasses"></i>
    {{ getText }}
  </button>
</template>

<script>
  export default {
    props: {
      model: {
        type: Object,
        required: true
      },
      url: {
        type: String,
        required: true
      }
    },
    methods: {
      toggle() {
        let method = this.model.is_liked ? 'delete' : 'post';
        axios[method](this.url)
          .then(res => {
            this.model.is_liked = !this.model.is_liked;

            if (method === 'post') {
              this.model.likes_count++;
            } else {
              this.model.likes_count--;
            }
          })
```

```
        }
    },
    computed: {
        getText(){
            return this.model.is_liked ? 'TE GUSTA' : 'ME GUSTA';
        },
        getBtnClasses(){
            return [
                this.model.is_liked ? 'font-weight-bold' : '',
                'btn', 'btn-link', 'btn-sm',
            ]
        },
        getIconClasses(){
            return [
                this.model.is_liked ? 'fa' : 'far',
                'fa-thumbs-up', 'text-primary', 'mr-1',
            ]
        }
    }
  }
</script>
```

Explicaciones: Comenzamos quitando el parámetro status de los métodos like y unlike, al tenerlo como propiedad de vue que cambiamos a model para darle un nombre mas generico.
Cambiamos la url para recibirla como parámetro porque la url cambia dependiendo del modelo al que va a dar like.

Creamos un solo método para dar y quitar likes, toggle, siguiendo el ejemplo de los 2 métodos anteriormente creados para que el mismo botón haga ambas acciones.

Creamos una propiedad calculada getText para obtener el texto del botón.
Y otras 2 propiedades calculadas getBtnClasses y getIconClasses para obtener las clases de bootstrap que diferencian a cada botón.

Quitamos el selector de dusk y lo agregamos al momento de usar el componente en el StatusListItem. Y por ultimo quitamos los metodos like y unlike al ya no utilizarlos.

Alt+J para seleccionar múltiples ocurrencias de lo seleccionado y reemplazarlo.

Vamos al StatusListItem y modificamos:

```
<div class="card-footer pt-2 d-flex justify-content-between align-items-center">

  <like-btn
    dusk="like-btn"
    :url="`/statuses/${status.id}/likes`"
    :model="status"
  ></like-btn>
```

```
  <div class="text-secondary mr-2">
    <i class="far fa-thumbs-up"></i>
    <span dusk="likes-count">{{ status.likes_count }}</span>
  </div>
</div>
<div class="card-footer">
  <div v-for="comment in comments" class="mb-3">
                                    <img class="rounded shadow-sm float-left mr-2"
width="34px" :src="comment.user_avatar" :alt="comment.user_name">
    <div class="card border-0 shadow-sm">
      <div class="card-body p-2 text-secondary">
        <a href=""><strong>{{ comment.user_name }}</strong></a>
        {{ comment.body }}
      </div>
    </div>
    <span dusk="comment-likes-count">{{ comment.likes_count }}</span>

    <like-btn
      dusk="comment-like-btn"
      :url="`/comments/${comment.id}/likes`"
      :model="comment"
    ></like-btn>

  </div>
```

(... Y en el script quitamos los métodos likeComment y unlikeComment …)

```
<script>
  import LikeBtn from './LikeBtn'

  export default {
    props: {
      status: {
        type: Object,
        required: true
      }
    },
    components: { LikeBtn },
    data() {
      return {
        newComment: '',
        comments: this.status.comments
      }
    },
    methods: {
      addComment() {
        axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
          .then(res => {
            this.newComment = '';
            this.comments.push(res.data.data);
          })
          .catch(err => {
```

```
                    console.log(err.response.data)
                })
            }
        }
    }
</script>
```

Explicación: Dentro del like-btn cambiamos la propiedad status a model, le pasamos la url, y añadimos el selector de dusk a usar para el botón.
Copiamos este like btn y lo añadimos en la parte de los comentarios quitando los botones anteriores. Por ultimo, quitamos los métodos likeComment y unlikeComment que ya no necesitamos

Vamos al test de dusk UsersCanLikeStatusesTest y quitamos las referencias al selector unlike-btn:

```
->press('@like-btn')
->waitForText('ME GUSTA')
->assertSee('ME GUSTA')
->assertSeeIn('@likes-count', 0)
```

Y en el test de dusk UsersCanLikeCommentsTest hacemos lo mismo que en test anterior quitando las referencias a unlike-btn:

```
->press('@comment-like-btn')
->waitForText('ME GUSTA')
->assertSee('ME GUSTA')
//->assertSeeIn('@comment-likes-count', 0) // Esta comprobación falla por algun motivo (video 41)
```

Comprobamos que las modificaciones realizadas no han roto los tests de dar like a los estados: php artisan dusk --filter UsersCanLikeStatusesTest
Y los tests de los comentarios: php artisan dusk --filter UsersCanLikeStatusesTest

[Fin video 41]

Comprobamos los cambios que llevamos realizados en navegador, si no se muestran los estados y comentarios puede ser por no haber actualizado las migraciones de los likes y comentarios.
Para ello ejecutamos en la terminal:

```
php artisan migrate:rollback --step=2
php artisan migrate
```

En mi caso no ha sido necesario realizarlo, la página se mostraba ya correctamente.

Ahora vamos a darle estilo con bootstrap al boton de like de los comentarios.
Para ello vamos al StatusListItem y modificamos el card-footer de comentarios:

```
<div class="card-footer">
  <div v-for="comment in comments" class="mb-3">
    <div class="d-flex">
                                    <img class="rounded shadow-sm mr-2" height="34px"
width="34px" :src="comment.user_avatar" :alt="comment.user_name">
      <div class="flex-grow-1">
        <div class="card border-0 shadow-sm">
          <div class="card-body p-2 text-secondary">
            <a href=""><strong>{{ comment.user_name }}</strong></a>
            {{ comment.body }}
          </div>
        </div>

                      <small class="badge badge-pill py-1 px-2 mt-1 badge-primary float-right"
dusk="comment-likes-count">
          <i class="fa fa-thumbs-up"></i>
          {{ comment.likes_count }}
        </small>

        <like-btn
          dusk="comment-like-btn"
          :url="`/comments/${comment.id}/likes`"
          :model="comment"
          class="comments-like-btn"
        ></like-btn>
      </div>
    </div>
  </div>
```

Explicación: Encerramos la imagen y el cuerpo del comentario en un div para agruparlos, y otro div para encerrar el comentario, el conteo de likes y el botón like.
Damos el estilo correcto a la imagen para que se muestre correctamente. Hacemos el conteo de likes más pequeño con etiqueta small, añadimos clases para darle formato y agregamos un icono.
Al componente like-btn le añadimos una clase comments-like-btn que vamos a definir nosotros en el archivo del componente de likes LikeBtn.vue

Añadimos la clase comments-like-btn de css al componente LikeBtn.vue:

```
<style lang="scss" scoped>
  .comments-like-btn{
    font-size: 0.6em;
    padding-left: 0;
    i { display: none}
  }
</style>
```

Compilamos los cambios: npm run dev
Y comprobamos que la página se muestra correctamente en el navegador.

[Fin video 42]

Vamos a comenzar a crear el perfil de usuario.
Creamos el test en la terminal: php artisan make:test CanSeeProfilesTest

Y agregamos el primer test a CanSeeProfilesTest (tests/feature) donde creamos un usuario con el que visitar su perfil al ir a la url: @nombre de usuario:

```
namespace Tests\Feature;

use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanSeeProfilesTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function can_see_profiles_test()
  {
     $this->withoutExceptionHandling();
     factory(User::class)->create(['name' => 'Angel']);

     $this->get('@Angel')->assertSee('Angel');

  }
}
```

Creamos la ruta en el archivo de rutas web.php, añadiendo:

```
// Users routes
Route::get('@{user}', 'UsersController@show')->name('users.show');
```

Creamos el controlador definido en la ruta, en terminal:

```
php artisan make:controller UsersController
```

Y creamos el método show en el UsersController (app/http/controllers) que devuelve la vista users.show del usuario:

```
namespace App\Http\Controllers;

use App\User;
```

```
use Illuminate\Http\Request;

class UsersController extends Controller
{
  public function show(User $user)
  {
     return view('users.show', compact('user'));
  }
}
```

Vamos al modelo User.php (app) y añadimos el método getRouteKeyName para que use el nombre del usuario en la ruta, en vez de usar el id para buscarlo:

```
public function getRouteKeyName()
{
  return 'name';
}
```

Ahora creamos la vista show.blade.php dentro de la carpeta users que también crearemos en (resources/views). Click derecho en resources/views y crear nuevo archivo users/show.blade.php para ello.
Y dentro del archivo imprimimos el nombre del usuario: {{ $user->name }}

Vamos a agregar la funcionalidad de que al hacer click en el nombre de un usuario (tanto en el estado como en comentarios) nos lleve a su perfil de usuario. Y que el avatar esté implementado en el modelo User en vez de tener que usar el enlace.

Para ello, vamos al StatusListItem y modificamos:

```
<template>
  <div class="card border-0 mb-3 shadow-sm">
     <div class="card-body d-flex flex-column">
        <div class="d-flex align-items-center mb-3">
              <img class="rounded mr-3 shadow-sm" width="40px" :src="status.user_avatar"
alt="">
           <div class="">
                                                <h5   class="mb-1"><a   :href="status.user_link"
v-text="status.user_name"></a></h5>
              <div class="small text-muted" v-text="status.ago"></div>
           </div>
        </div>
        <p class="card-text text-secondary" v-text="status.body"></p>
     </div>
     <div class="card-footer pt-2 d-flex justify-content-between align-items-center">

        <like-btn
           dusk="like-btn"
           :url="`/statuses/${status.id}/likes`"
           :model="status"
```

```
        ></like-btn>

        <div class="text-secondary mr-2">
          <i class="far fa-thumbs-up"></i>
          <span dusk="likes-count">{{ status.likes_count }}</span>
        </div>
      </div>

      <div class="card-footer">
        <div v-for="comment in comments" class="mb-3">
          <div class="d-flex">
                                        <img class="rounded shadow-sm mr-2" height="34px"
width="34px" :src="comment.user_avatar" :alt="comment.user_name">
            <div class="flex-grow-1">
              <div class="card border-0 shadow-sm">
                <div class="card-body p-2 text-secondary">
                                    <a :href="comment.user_link"><strong>{{ comment.user_name
}}</strong></a>
                  {{ comment.body }}
                </div>
              </div>

                  <small class="badge badge-pill py-1 px-2 mt-1 badge-primary float-right"
dusk="comment-likes-count">
                <i class="fa fa-thumbs-up"></i>
                {{ comment.likes_count }}
              </small>

              <like-btn
                dusk="comment-like-btn"
                :url="`/comments/${comment.id}/likes`"
                :model="comment"
                class="comments-like-btn"
              ></like-btn>
            </div>
          </div>
        </div>

        <form @submit.prevent="addComment" v-if="isAuthenticated">
          <div class="d-flex align-items-center ">
                                          <img class="rounded shadow-sm float-left mr-2"
width="34px" :src="currentUser.avatar" :alt="currentUser.name">
            <div class="input-group">
                  <textarea class="form-control border-0 shadow-sm" name="comment" v-
model="newComment" placeholder="Escribe un comentario.." rows="1" required></textarea>

              <div class="input-group-append">
                <button class="btn btn-primary" dusk="comment-btn">Enviar</button>
              </div>
            </div>
          </div>
        </form>
```

```
      </div>
    </div>
</template>
```

Vamos al StatusResourceTest y añadimos la validación para el user_link, y modificamos la validación del avatar.

```
$this->assertEquals(
  $status->user->link(),
  $statusResource['user_link']
);
$this->assertEquals(
  $status->user->avatar(),
  $statusResource['user_avatar']
);
```

En el CommentResourceTest tambien añadimos la validación para el user_link y modificamos la del avatar:

```
$this->assertEquals(
  $comment->user->link(),
  $commentResource['user_link']
);

$this->assertEquals(
  $comment->user->avatar(),
  $commentResource['user_avatar']
);
```

En el modelo User.php (app) creamos el método link y el avatar.
En el link retornamos la ruta users.show y pasamos el ususario actual.
En el avatar, devolvemos la imagen por defecto de avatar:

```
public function link()
{
  return route('users.show', $this);
}

public function avatar()
{
  return 'https://aprendible.com/images/default-avatar.jpg';
}
```

Finalmente vamos al StatusResource y añadimos el campo user_link y user_avatar:

```
return [
  'id' => $this->id,
  'body' => $this->body,
  'user_name' => $this->user->name,
  'user_avatar' => $this->user->avatar(),
  'user_link' => $this->user->link(),
```

```
    'ago' => $this->created_at->diffForHumans(),
    'is_liked' => $this->isLiked(),
    'likes_count' => $this->likesCount(),
    'comments' => CommentResource::collection($this->comments)

  ];
```

Y en el CommentResource agregamos igualmente el user_link y el user_avatar:

```
return [
  'id' => $this->id,
  'body' => $this->body,
  'user_name' => $this->user->name,
  'user_link' => $this->user->link(),
  'user_avatar' => $this->user->avatar(),
  'likes_count' => $this->likesCount(),
  'is_liked' => $this->isLiked()
];
```

Por último compilamos los cambios: npm run dev
Y comprobamos en el navegador que podemos acceder al perfil del usuario haciendo click en su nombre.

[Fin video 43]

Vamos a crear los tests que verifiquen el funcionamiento de los métodos getRouteKeyName, link y avatar del modelo User añadidos en el video anterior.

Para ello, creamos un nuevo test: php artisan make:test UserTest --unit

Abrimos el test UserTest creado (tests/unit) y creamos los métodos de validación:

```
namespace Tests\Unit;

use App\User;
use Tests\TestCase;

class UserTest extends TestCase
{
  /** @test */
  function route_key_name_is_set_to_name()
  {
     $user = factory(User::class)->make();

       $this->assertEquals('name', $user->getRouteKeyName(), 'The route key name must be
name');
  }
```

```
    /** @test */
    function user_has_a_link_to_their_profile()
    {
        $user = factory(User::class)->make();

        $this->assertEquals(route('users.show', $user), $user->link());
    }

    /** @test */
    function user_has_an_avatar()
    {
        $user = factory(User::class)->make();

        $this->assertEquals('https://aprendible.com/images/default-avatar.jpg', $user->avatar());
    }
}
```

Explicación: Para el test de la ruta, creamos una instancia del modelo User y verificamos que recibimos el name al llamar a getRouteKeyName.
Para el test del link, creamos el usuario y verificamos que recibe la ruta users.show con el usuario como parametro al llamar a user->link().
Y para el test del avatar, verificamos que recibimos la cadena de caracteres al llamar a user->avatar().

Vamos a crear un nuevo recurso API para el modelo User y poder encapsular los métodos user_name, user_link y user_avatar.

Primero vamos al StatusResourceTest, verificamos que recibimos una instancia de UserResource y que devuelva el modelo User al llamar a user->resource (importamos UserResource y User al principio)
Y eliminamos las comprobaciones del user_name, user_link y user_avatar:

```
namespace Tests\Unit\Http\Resources;

use App\Http\Resources\CommentResource;
use App\Http\Resources\StatusResource;
use App\Http\Resources\UserResource;
use App\Models\Comment;
use App\Models\Status;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class StatusResourceTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
```

```
    public function a_status_resources_must_have_the_necessary_fields()
    {
        $status = factory(Status::class)->create();
        factory(Comment::class)->create(['status_id' => $status->id]);

        $statusResource = StatusResource::make($status)->resolve();

        $this->assertEquals(
            $status->id,
            $statusResource['id']
        );
        $this->assertEquals(
            $status->body,
            $statusResource['body']
        );
        $this->assertEquals(
            $status->created_at->diffForHumans(),
            $statusResource['ago']
        );
        $this->assertEquals(
            false,
            $statusResource['is_liked']
        );
        $this->assertEquals(
            0,
            $statusResource['likes_count']
        );
        $this->assertEquals(
            CommentResource::class,
            $statusResource['comments']->collects
        );
        $this->assertInstanceOf(
            Comment::class,
            $statusResource['comments']->first()->resource
        );
        $this->assertInstanceOf(
            UserResource::class,
            $statusResource['user']
        );
        $this->assertInstanceOf(
            User::class,
            $statusResource['user']->resource
        );
    }
}
```

En el StatusResource, definimos el index user con el modelo User asociado, y quitamos los campos de user_name, user_link y user_avatar.

```
return [
    'id' => $this->id,
    'body' => $this->body,
```

```
  'user' => UserResource::make($this->user),
  'ago' => $this->created_at->diffForHumans(),
  'is_liked' => $this->isLiked(),
  'likes_count' => $this->likesCount(),
  'comments' => CommentResource::collection($this->comments)

];
```

Creamos el UserResource que hemos definido anteriormente, en terminal:

```
php artisan make:resource UserResource
```

Ahora en el CommentResourceTest añadimos los métodos definidos en StatusResourceTest como plantilla y los modificamos, también importamos las clases UserResource y App\User .
Y eliminamos las comprobaciones del user_name, user_link y user_avatar:

```
namespace Tests\Unit\Http\Resources;

use App\Http\Resources\UserResource;
use App\User;
use Tests\TestCase;
use App\Models\Status;
use App\Http\Resources\CommentResource;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CommentResourceTest extends TestCase
{
  use RefreshDatabase;
  /** @test */
  public function a_comment_resources_must_have_the_necessary_fields()
  {
    $comment = factory(Status::class)->create();

    $commentResource = CommentResource::make($comment)->resolve();

    $this->assertEquals(
      $comment->id,
      $commentResource['id']
    );

    $this->assertEquals(
      $comment->body,
      $commentResource['body']
    );

    $this->assertEquals(
      0,
      $commentResource['likes_count']
    );

    $this->assertEquals(
```

```
            false,
            $commentResource['is_liked']
        );


        $this->assertInstanceOf(
            UserResource::class,
            $commentResource['user']
        );
        $this->assertInstanceOf(
            User::class,
            $commentResource['user']->resource
        );
    }
}
```

En el CommentResource, añadimos el índice user igual que en el StatusResource, y quitamos los campos de user_name, user_link y user_avatar :

```
return [
    'id' => $this->id,
    'body' => $this->body,
    'user' => UserResource::make($this->user),
    'likes_count' => $this->likesCount(),
    'is_liked' => $this->isLiked()
];
```

A continuación, vamos a crear un test para el UserResource creado anteriormente.

Para ello, vamos a la carpeta (tests/unit/http/resources) y usando como plantilla el CommentResourceTest creamos un nuevo test UserResourceTest:

```
namespace Tests\Unit\Http\Resources;

use App\User;
use Tests\TestCase;
use App\Http\Resources\UserResource;
use Illuminate\Foundation\Testing\RefreshDatabase;

class UserResourceTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function a_user_resources_must_have_the_necessary_fields()
    {
        $user = factory(User::class)->create();

        $userResource = UserResource::make($user)->resolve();

        $this->assertEquals(
```

```
            $user->name,
            $userResource['name']
        );

        $this->assertEquals(
            $user->link(),
            $userResource['link']
        );

        $this->assertEquals(
            $user->avatar(),
            $userResource['avatar']
        );
    }
}
```

Explicación: Comprobamos que devuelva el name cuando pidamos el user->name, el link y el avatar cuando pidamos el user->link() y user->avatar() respectivamente.

Continuando, en el UserResource (app/http/resources) devolvemos un array con los campos que pedimos en el test (name, link y avatar):

```
public function toArray($request)
{
    return [
        'name' => $this->name,
        'link' => $this->link(),
        'avatar' => $this->avatar(),
    ];
}
```

Finalmente, vamos al componente de vue StatusListItem y cambiamos todas las referencias a las propiedades user_name, user_link y user_avatar por las correctas que son usando el punto en vez de guion: user.name, user.link y user.avatar

Por último compilamos los cambios: npm run dev
Ejecutamos todos los tests de phpunit: vendor/phpunit/phpunit/phpunit
Los tests de dusk: php artisan dusk , y vemos que todos pasan Ok.

[Fin video 44]

Vamos a realizar modificaciones al registro de usuarios que viene por defecto en Laravel.
Comenzamos creando un test: php artisan make:test RegistrationTest

Abrimos el test creado RegistrationTest (tests/feature) y añadimos la información del

usuario a enviar por el registro, enviamos la información a la ruta register, verificamos la redirección a la raíz. Verificamos en la base de datos la información enviada y que la contraseña está encriptada:

```
namespace Tests\Feature;

use App\User;
use Illuminate\Support\Facades\Hash;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class RegistrationTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  function users_can_register()
  {
     $userData = [
        'name' => 'Angel',
        'first_name' => 'C',
        'last_name' => 'M',
        'email' => 'angel@email.com',
        'password' => 'secret',
        'password_confirmation' => 'secret',
     ];

     $response = $this->post(route('register'), $userData);

     $response->assertRedirect('/');

     $this->assertDatabaseHas('users', [
        'name' => 'Angel',
        'first_name' => 'C',
        'last_name' => 'M',
        'email' => 'angel@email.com',
     ]);

     $this->assertTrue(
        Hash::check('secret', User::first()->password),
        'The password need to be hashed'
     );
  }
}
```

Vamos al RegisterController (app/http/controllers/auth) para realizar los cambios necesarios para que pase el test, cambiando la redirección a la raiz, añadiendo los campos first_name y last_name al metodo de validacion y al de creación.
Y cambiando el metodo de encriptacion de la contraseña:

```
protected $redirectTo = '/';

protected function validator(array $data)
{
  return Validator::make($data, [
    'name' => 'required|string|max:255',
    'first_name' => 'required|string|max:255',
    'last_name' => 'required|string|max:255',
    'email' => 'required|string|email|max:255|unique:users',
    'password' => 'required|string|min:6|confirmed',
  ]);
}

protected function create(array $data)
{
  return User::create([
    'name' => $data['name'],
    'first_name' => $data['first_name'],
    'last_name' => $data['last_name'],
    'email' => $data['email'],
    'password' => bcrypt($data['password']),
  ]);
}
```

Ahora es necesario cambiar los campos permitidos en la asignación masiva para el modelo User, que en nuestro caso vamos a deshabilitar la protección.
En el modelo User (app) modificamos:

```
protected $fillable = [
  'name', 'email', 'password',
];

(... Cambiamos $fillable por $guarded …)

protected $guarded = [];
```

Agregamos los campos first_name y last_name a la migración de la tabla users (create_users_table en databases/migrations):

```
public function up()
{
  Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name');
    $table->string('first_name');
    $table->string('last_name');
    $table->string('email')->unique();
    $table->string('password');
    $table->rememberToken();
    $table->timestamps();
  });
```

```
}
```

Y finalmente para que todos los test pasen, es necesario añadir también los campos first_name y last_name a la factoria de creacion de usuarios.
Por ello, vamos al UserFactory (databases/factories) y añadimos los campos:

```
$factory->define(App\User::class, function (Faker $faker) {
  return [
    'name' => $faker->userName,
    'first_name' => $faker->firstName,
    'last_name' => $faker->lastName,
    'email' => $faker->unique()->safeEmail,
                                                    'password'          =>
'$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
    'remember_token' => str_random(10),
  ];
});
```

Así, si ejecutamos todos los tests de phpunit: vendor/phpunit/phpunit/phpunit
Y los tests de dusk: php artisan dusk , vemos que todos pasan Ok.

[Fin video 45]

Vamos a agregar tests que verifiquen cada una de las reglas de validación que necesitamos para registrar a un usuario.

Comenzamos creando los tests para las validaciones en el RegistrationTest (tests/feature). Añadiendo tests para el nombre, first_name, last_name, email y password. El archivo quedaría de la siguiente forma:

```
namespace Tests\Feature;

use App\User;
use Illuminate\Support\Facades\Hash;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class RegistrationTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function users_can_register()
  {
    $response = $this->post(route('register'), $this->userValidData());
```

```
        $response->assertRedirect('/');

        $this->assertDatabaseHas('users', [
            'name' => 'Angel',
            'first_name' => 'C',
            'last_name' => 'M',
            'email' => 'angel@email.com',
        ]);

        $this->assertTrue(
            Hash::check('secret', User::first()->password),
            'The password need to be hashed'
        );
    }

    // Tests para el nombre
    /** @test */
    public function the_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => null])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => 1234])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_may_not_be_greater_than_60_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => str_random(61)])
        )->assertSessionHasErrors('name');

    }

    //tests para el first name
    /** @test */
    public function the_first_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => null])
```

```
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => 1234])
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_may_not_be_greater_than_60_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => str_random(61)])
        )->assertSessionHasErrors('first_name');

    }

    //tests para el last name
    /** @test */
    public function the_last_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => null])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => 1234])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_may_not_be_greater_than_60_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => str_random(61)])
        )->assertSessionHasErrors('last_name');
```

```
    }

    //tests para el email
    /** @test */
    public function the_email_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['email' => null])
        )->assertSessionHasErrors('email');

    }

    /** @test */
    public function the_email_must_be_a_valid_address()
    {
        $this->post(
            route('register'),
            $this->userValidData(['email' => 'invalid@email'])
        )->assertSessionHasErrors('email');

    }

    /** @test */
    public function the_email_must_be_unique()
    {
        factory(User::class)->create(['email' => 'angel@email.com']);

        $this->post(
            route('register'),
            $this->userValidData(['email' => 'angel@email.com'])
        )->assertSessionHasErrors('email');

    }

    // tests para la contraseña
    /** @test */
    public function the_password_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['password' => null])
        )->assertSessionHasErrors('password');

    }

    /** @test */
    public function the_password_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['password' => 1234])
        )->assertSessionHasErrors('password');
```

```
    }

    /** @test */
    public function the_password_must_be_at_least_6_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['password' => 'asdfg'])
        )->assertSessionHasErrors('password');

    }

    /** @test */
    public function the_password_must_be_confirmed()
    {
        $this->post(
            route('register'),
            $this->userValidData([
                'password' => 'secret',
                'password_confirmation' => null
            ])
        )->assertSessionHasErrors('password');

    }

// extraemos la informacion valida para crear un usuario a una funcion, desde el test users_can_register
    /**
     * @return string[]
     */
    public function userValidData($overrides = []): array
    {
        return array_merge([
            'name' => 'Angel',
            'first_name' => 'C',
            'last_name' => 'M',
            'email' => 'angel@email.com',
            'password' => 'secret',
            'password_confirmation' => 'secret',
        ], $overrides);
    }
}
```

Explicación: Extraemos la información para crear un usuario a una función con Ctrl+Alt+M en phpstorm, que llamamos userValidData y la modificamos para que mezcle los datos por defecto con los que le pasemos por el test.

Luego, creamos los tests para verificar que las propiedades son requerida, que el valor de esta es un string, que tiene como máximo 60 caracteres. Si los tests fallan es porque no se ha verificado estas propiedades al no dar error en la base de datos.

En el caso del email que es de tipo email y único, y para la contraseña que tiene longitud mínima de 6 caracteres y necesita confirmación.

Estos test los creamos para el nombre, el primer apellido, el segundo apellido, el email y la contraseña.
Los tests seguirán una estructura similar, pasando los datos del usuario a crear a la ruta register con un dato incorrecto dependiendo del test, y luego comprobando que en la respuesta hay un error.

Finalmente en el RegisterController (app/http/controllers/auth) añadimos la validación necesaria para que pasen los tests creados:

```
protected function validator(array $data)
{
  return Validator::make($data, [
     'name' => ['required', 'string', 'max:60'],
     'first_name' => ['required', 'string', 'max:60'],
     'last_name' => ['required', 'string', 'max:60'],
     'email' => ['required', 'email', 'unique:users'],
     'password' => ['required', 'string', 'min:6', 'confirmed'],
  ]);
}
```

Y comprobamos que todos los tests creados pasan al ejecutarlos en phpstorm.

[Fin video 46]

Continuamos con la validación del registro de usuarios en la clase RegistrationTest.

Añadimos que el nombre sea unico, tenga un mínimo de 3 caracteres y que el nombre pueda tener solo letras y números. Y que el first_name y last_name tengan un mínimo de 3 caracteres y solo permiten letras.
Y modificamos los datos de la función userValidData para cambiar los datos a pasar por defecto para comprobar los tests. El archivo quedaría:

```
class RegistrationTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function users_can_register()
  {
     $response = $this->post(route('register'), $this->userValidData());

     $response->assertRedirect('/');
```

```
        $this->assertDatabaseHas('users', [
            'name' => 'Angel2',
            'first_name' => 'canovas',
            'last_name' => 'canovas',
            'email' => 'angel@email.com',
        ]);

        $this->assertTrue(
            Hash::check('secret', User::first()->password),
            'The password need to be hashed'
        );
    }

    // Tests para el nombre
    /** @test */
    public function the_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => null])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => 1234])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_may_not_be_greater_than_60_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => str_random(61)])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_must_be_at_least_3_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => 'as'])
        )->assertSessionHasErrors('name');
```

```
    }

    /** @test */
    public function the_name_must_be_unique()
    {
        factory(User::class)->create(['name' => 'Angel']);

        $this->post(
            route('register'),
            $this->userValidData(['name' => 'Angel'])
        )->assertSessionHasErrors('name');

    }

    /** @test */
    public function the_name_may_only_contain_letters_and_numbers()
    {
        $this->post(
            route('register'),
            $this->userValidData(['name' => 'Angel C'])
        )->assertSessionHasErrors('name');

        $this->post(
            route('register'),
            $this->userValidData(['name' => 'Angel<>'])
        )->assertSessionHasErrors('name');
    }

    //tests para el first name
    /** @test */
    public function the_first_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => null])
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => 1234])
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_may_not_be_greater_than_60_characters()
    {
        $this->post(
```

```
            route('register'),
            $this->userValidData(['first_name' => str_random(61)])
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_must_be_at_least_3_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => 'as'])
        )->assertSessionHasErrors('first_name');

    }

    /** @test */
    public function the_first_name_may_only_contain_letters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['first_name' => 'Angel2'])
        )->assertSessionHasErrors('first_name');

        $this->post(
            route('register'),
            $this->userValidData(['first_name' => 'Angel<>'])
        )->assertSessionHasErrors('first_name');
    }

    //tests para el last name
    /** @test */
    public function the_last_name_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => null])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_must_be_a_string()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => 1234])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_may_not_be_greater_than_60_characters()
```

```
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => str_random(61)])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_must_be_at_least_3_characters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => 'as'])
        )->assertSessionHasErrors('last_name');

    }

    /** @test */
    public function the_last_name_may_only_contain_letters()
    {
        $this->post(
            route('register'),
            $this->userValidData(['last_name' => 'Angel2'])
        )->assertSessionHasErrors('last_name');

        $this->post(
            route('register'),
            $this->userValidData(['last_name' => 'Angel<>'])
        )->assertSessionHasErrors('last_name');
    }

    //tests para el email
    /** @test */
    public function the_email_is_required()
    {
        $this->post(
            route('register'),
            $this->userValidData(['email' => null])
        )->assertSessionHasErrors('email');

    }

    /** @test */
    public function the_email_must_be_a_valid_address()
    {
        $this->post(
            route('register'),
            $this->userValidData(['email' => 'invalid@email'])
        )->assertSessionHasErrors('email');

    }
```

```
/** @test */
public function the_email_must_be_unique()
{
    factory(User::class)->create(['email' => 'angel@email.com']);

    $this->post(
        route('register'),
        $this->userValidData(['email' => 'angel@email.com'])
    )->assertSessionHasErrors('email');

}

// tests para la contraseña
/** @test */
public function the_password_is_required()
{
    $this->post(
        route('register'),
        $this->userValidData(['password' => null])
    )->assertSessionHasErrors('password');

}

/** @test */
public function the_password_must_be_a_string()
{
    $this->post(
        route('register'),
        $this->userValidData(['password' => 1234])
    )->assertSessionHasErrors('password');

}

/** @test */
public function the_password_must_be_at_least_6_characters()
{
    $this->post(
        route('register'),
        $this->userValidData(['password' => 'asdfg'])
    )->assertSessionHasErrors('password');

}

/** @test */
public function the_password_must_be_confirmed()
{
    $this->post(
        route('register'),
        $this->userValidData([
            'password' => 'secret',
            'password_confirmation' => null
        ])
    )->assertSessionHasErrors('password');
```

```
    }

    // extraemos la información válida para crear un usuario a una función, desde el test users_can_register
    /**
     * @return string[]
     */
    public function userValidData($overrides = []): array
    {
        return array_merge([
            'name' => 'Angel2',
            'first_name' => 'canovas',
            'last_name' => 'canovas',
            'email' => 'angel@email.com',
            'password' => 'secret',
            'password_confirmation' => 'secret',
        ], $overrides);
    }
}
```

Y en el RegisterController añadimos la validación correspondiente:

```
return Validator::make($data, [
    'name' => ['required', 'string', 'max:60', 'min:3', 'unique:users', 'alpha_num'],
    'first_name' => ['required', 'string', 'max:60', 'min:3', 'alpha'],
    'last_name' => ['required', 'string', 'max:60', 'min:3', 'alpha'],
    'email' => ['required', 'email', 'unique:users'],
    'password' => ['required', 'string', 'min:6', 'confirmed'],
]);
```

[Fin video 47]

Vamos a testear con laravel Dusk que podemos registrarnos con el navegador.

Empezamos creando un test: php artisan dusk:make UsersCanRegisterTest

Vamos al test creado, y creamos los tests necesarios:

```
namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanRegisterTest extends DuskTestCase
{
    use DatabaseMigrations;
```

```
    /**
     * @test
     * @throws \Throwable
     */
    public function user_can_register()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/register')
                ->type('name', 'AngelCanovas')
                ->type('first_name', 'Angel')
                ->type('last_name', 'Canovas')
                ->type('email', 'angel@email.com')
                ->type('password', 'secret')
                ->type('password_confirmation', 'secret')
                ->press('@register-btn')
                ->assertPathIs('/')
                ->assertAuthenticated()
            ;
        });
    }

    /**
     * @test
     * @throws \Throwable
     */
    public function user_cannot_register_with_invalid_information()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/register')
                ->type('name', '')
                ->press('@register-btn')
                ->assertPathIs('/register')
                ->assertPresent('@validation-errors')
            ;
        });
    }
}
```

Explicación: Comprobamos que el usuario puede registrarse yendo a la ruta register, introduciendo la información necesaria en los campos del formulario, y al enviar el formulario la ruta es el home y estamos autenticados.
También comprobamos que el usuario no puede registrarse con información inválida, para ello enviamos el nombre vacío, y que la ruta sigue siendo register y existe el componente validation-errors donde se muestran los errores.

Ahora en el RegistrationTest, donde comprobamos que el usuario puede registrase realizamos una petición a la ruta register para recibir una respuesta exitosa.

```
    public function users_can_register()
    {
```

```
    $this->get(route('register'))->assertSuccessful();

    $response = $this->post(route('register'), $this->userValidData());

    $response->assertRedirect('/');

    $this->assertDatabaseHas('users', [
        'name' => 'Angel2',
        'first_name' => 'canovas',
        'last_name' => 'canovas',
        'email' => 'angel@email.com',
    ]);

    $this->assertTrue(
        Hash::check('secret', User::first()->password),
        'The password need to be hashed'
    );
}
```

A continuación creamos la vista para el registro en resources/views/auth, creamos el archivo register.blade.php . Ahora usando la vista login como plantilla creamos:

```
@extends('layouts.app')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-6 mx-auto">

                @include('partials.validation-errors')

                <div class="card border-0 bg-light px-4 py-2">
                    <form action="{{ route('register') }}" method="POST">
                        @csrf
                        <div class="card-body">
                            <div class="form-group">
                                <label>Username:</label>
                                    <input class="form-control border-0" type="text" name="name"
placeholder="Tu nombre de usuario...">
                            </div>
                            <div class="form-group">
                                <label>Nombre:</label>
                                    <input class="form-control border-0" type="text" name="first_name"
placeholder="Tu nombre...">
                            </div>
                            <div class="form-group">
                                <label>Apellido:</label>
                                    <input class="form-control border-0" type="text" name="last_name"
placeholder="Tu apellido...">
                            </div>
                            <div class="form-group">
                                <label>Email:</label>
                                    <input class="form-control border-0" type="email" name="email"
```

```
placeholder="Tu email...">
                    </div>
                    <div class="form-group">
                        <label>Contraseña:</label>
                          <input class="form-control border-0" type="password" name="password"
placeholder="Tu contraseña...">
                    </div>
                    <div class="form-group">
                        <label>Repite la contraseña:</label>
                                                <input  class="form-control  border-0"  type="password"
name="password_confirmation" placeholder="Repite tu contraseña...">
                    </div>

                     <button class="btn btn-primary btn-block" dusk="register-btn">Registrarse</
button>
                </div>
            </form>
        </div>
    </div>
  </div>
@endsection
```

Explicación: Creamos los campos necesarios para el formulario que son el username, el first_name, el last_name y la confirmación de contraseña. Cambiamos el identificador del botón, y agregamos el elemento donde se mostraran los errores (si hay) al registrarse, que todavía no hemos creado.

Seguidamente, creamos el partial, que es un elemento que contiene parte de código que se repite para las vistas, donde mostraremos los errores ocurridos en el registro o login del usuario en la página.
Para ello, en resources/views creamos la carpeta partials y dentro de ella el archivo validation-errors.blade.php . Y añadimos:

```
@if($errors->any())
  <div class="alert alert-danger" dusk="validation-errors">
      @foreach($errors->all() as $error)
          {{ $error }} <br>
      @endforeach
  </div>
@endif
```

Explicación: En este partial, comprobamos si hay errores de sesión. Si hay, creamos un div con selector de dusk validation-errors y clases de bootstrap de alerta, donde mostrar secuencialmente todos los errores. Si no hay errores no hacemos nada.

Para mantener la misma estructura de nombres en los test, vamos a refactorizar el nombre del LoginTest (tests/browser) por UsersCanLoginTest.

Para hacerlo, vamos al archivo, hacemos click en el nombre de la clase, shift+F6 y cambiamos el nombre a UsersCanLoginTest y confirmamos con enter.

Una vez refactorizado, añadimos el siguiente test para comprobar que no podemos hacer login con información invalida:

```
/**
* @test
* @throws \Throwable
*/
public function user_cannot_login_with_invalid_information()
{
  $this->browse(function (Browser $browser) {
    $browser->visit('/login')
      ->type('email', '')
      ->press('@login-btn')
      ->assertPathIs('/login')
      ->assertPresent('@validation-errors')
    ;
  });
}
```

Finalmente, vamos a la vista de login login.blade.php y añadimos el partial para mostrar los errores de login y cambiamos el id por un selector de dusk:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    <div class="row">
      <div class="col-md-6 mx-auto">

        @include('partials.validation-errors')

        <div class="card border-0 bg-light px-4 py-2">
          <form action="{{ route('login') }}" method="POST">
            @csrf
            <div class="card-body">

              <div class="form-group">
                <label>Email:</label>
                  <input class="form-control border-0" type="email" name="email"
placeholder="Tu email...">
              </div>

              <div class="form-group">
                <label>Contraseña:</label>
                  <input class="form-control border-0" type="password" name="password"
placeholder="Tu contraseña...">
              </div>
```

```
                    <button class="btn btn-primary btn-block" dusk="login-btn">Login</button>
                </div>
            </form>
        </div>
    </div>
</div>
</div>
@endsection
```

Por último, comprobamos que pasan todos los test de dusk.

[Fin video 48]

Vamos a probar el registro manualmente desde el navegador, pero antes debemos agregar un link a la barra de navegación que lleve al registro. Y también añadimos un elemento en el desplegable que lleve al perfil de usuario.

Para realizar estos cambios, modificamos el app.blade (resources/views/layouts):

```
@guest()
  <li class="nav-item"><a href="{{ route('register') }}" class="nav-link">Register</a></li>
  <li class="nav-item"><a href="{{ route('login') }}" class="nav-link">Login</a></li>
@else
  <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-
toggle="dropdown" aria-haspopup="true" aria-expanded="false">
        {{ Auth::user()->name }}
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
      <a class="dropdown-item" href="{{ route('users.show', Auth::user()) }}">Perfil</a>
      <div class="dropdown-divider"></div>
          <a  onclick="document.getElementById('logout').submit()"  class="dropdown-item"
href="#">Cerrar sesión</a>
    </div>
  </li>

  <form id="logout" action="{{ route('logout') }}" method="POST">@csrf</form>
@endguest
```

Antes vamos al navegador y hacer un registro, debemos refrescar la base de datos para que se apliquen los nuevos cambios que hemos realizado en videos anteriores:

php artisan migrate:fresh --seed

Y creamos algunos estados nuevos al borrarse por actualizar:

php artisan tinker

factory('App\Models\Status', 20)->create();

Una vez hecho el registro de un nuevo usuario, vemos que no se muestra el avatar del usuario autenticado en los comentarios.
Para solucionarlo, en el UserTest verificamos que al llamar a avatar como propiedad también devuelva la imagen. Añadimos:

```
/** @test */
function user_has_an_avatar()
{
  $user = factory(User::class)->make();

  $this->assertEquals('https://aprendible.com/images/default-avatar.jpg', $user->avatar());
  $this->assertEquals('https://aprendible.com/images/default-avatar.jpg', $user->avatar);

}
```

Y en el User.php (app) añadimos la propiedad appends con el avatar y definimos un getter para el avatar:

```
protected $hidden = [
      'password', 'remember_token',
  ];

  protected $appends = ['avatar'];

  public function getRouteKeyName()
  {
     return 'name';
  }

  public function link()
  {
     return route('users.show', $this);
  }

  public function avatar()
  {
     return 'https://aprendible.com/images/default-avatar.jpg';
  }

  public function getAvatarAttribute()
  {
     return $this->avatar();
  }
}
```

Finalmente, agregamos el diseño a la página del perfil del usuario. Para ello, vamos al show.blade.php de (resources/views/users):

```
@extends('layouts.app')
```

```
@section('content')
  <div class="container">
    <div class="row">
      <div class="col-md-3">
        <div class="card border-0 bg-light shadow-sm">
          <img src="{{ $user->avatar }}" alt="{{ $user->name }}" class="card-img-top">
          <div class="card-body">
            {{ $user->name }}
          </div>
        </div>
      </div>
      <div class="col-md-9">
        <div class="card border-0 bg-light shadow-sm">
          <div class="card-body">
            Mi contenido
          </div>
        </div>
      </div>
    </div>
  </div>

@endsection
```

Explicacion: Esta vista users show extiende del layouts.app con la seccion content. Dentro de la sección mostramos el avatar del usuario y dejamos creada la parte para la información personal del usuario. Damos estilo con bootstrap.

Al final comprobamos que los cambios realizados se muestran correctamente.

[Fin video 49]

Creamos un test para probar que obtenemos los estados de un usuario específico en ListStatusesTest, de forma similar a pasos anteriores:

```
/** @test */
function can_get_statuses_for_a_specific_user()
{
  $user = factory(User::class)->create();
    $status1 = factory(Status::class)->create(['user_id' => $user->id, 'created_at' => now()->subDay()]);
  $status2 = factory(Status::class)->create(['user_id' => $user->id]);

  $otherStatuses = factory(Status::class, 2)->create();

  $response = $this->actingAs($user)->getJson(route('users.statuses.index', $user));

  $response->assertJson([
```

```
        'meta' => ['total' => 2]
    ]);

    $response->assertJsonStructure([
        'data', 'links' => ['prev', 'next']
    ]);

    $this->assertEquals(
        $status2->body,
        $response->json('data.0.body')
    );
}
```

Ahora creamos la ruta en el archivo de rutas web.php:

```
// Users statuses routes
Route::get('users/{user}/statuses', 'UsersStatusController@index')->name('users.statuses.index');
```

Creamos un nuevo controlador UsersStatusController, donde devolvemos una colección de estados paginados:

```
namespace App\Http\Controllers;

use App\User;
use App\Http\Resources\StatusResource;

class UsersStatusController extends Controller
{
    public function index(User $user)
    {
        return StatusResource::collection(
            $user->statuses()->latest()->paginate()
        );
    }
}
```

Necesitamos crear ahora el statuses del controlador anterior. Por ello en el UserTest creamos un nuevo test para verificar que un usuario tiene muchos estados:

```
/** @test */
function user_has_many_statuses()
{
    $user = factory(User::class)->create();

    factory(Status::class)->create(['user_id' => $user->id]);

    $this->assertInstanceOf(Status::class, $user->statuses->first());
}
```

Seguidamente, en el modelo User.php agregamos la relación hasMany:

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

Si ejecutamos el test creado en el ListStatusesTest ahora pasa.

Continuando, creamos el test de dusk UsersCanSeeProfilesTest en la carpeta tests/Browser para comprobar que los usuarios pueden ver los perfiles:

```
namespace Tests\Browser;

use App\Models\Status;
use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanSeeProfilesTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * @test
   * @throws \Throwable
   */
  public function users_can_see_profiles()
  {
      $user = factory(User::class)->create();
      $statuses = factory(Status::class, 2)->create(['user_id' => $user->id]);
      $otherStatus = factory(Status::class)->create();

      $this->browse(function (Browser $browser) use ($user, $statuses, $otherStatus) {
          $browser->visit("/@{$user->name}")
              ->assertSee($user->name)
              ->waitForText($statuses->first()->body)
              ->assertSee($statuses->first()->body)
              ->assertSee($statuses->last()->body)
              ->assertDontSee($otherStatus->body)
          ;
      });
  }
}
```

Seguimos añadiendo el texto del estado en la vista de usuarios show.blade.php:

```
@extends('layouts.app')

@section('content')
  <div class="container">
```

```
        <div class="row">
            <div class="col-md-3">
                <div class="card border-0 bg-light shadow-sm">
                    <img src="{{ $user->avatar }}" alt="{{ $user->name }}" class="card-img-top">
                    <div class="card-body">
                        <h5 class="card-title">{{ $user->name }}</h5>
                    </div>
                </div>
            </div>
            <div class="col-md-9">
                    <status-list
                        url="{{route('users.statuses.index', $user)}}"
                    ></status-list>
                </div>
            </div>
        </div>
    </div>

@endsection
```

Ahora en el StatusList.vue añadimos la url como propiedad, la propiedad calculada getUrl (que devuelve la url si la recibimos, y si no la url statuses que devuelve todos los estados), y el get a la url:

```
<script>
    import StatusListItem from './StatusListItem'

    export default {
        components: { StatusListItem },
        props: {
            url: String
        },
        data() {
            return {
                statuses: []
            }
        },
        mounted() {
            axios.get(this.getUrl)
                .then(res => {
                    this.statuses = res.data.data
                })
                .catch(err => {
                    console.log(err.response.data);
                });

            EventBus.$on('status-created', status => {
                this.statuses.unshift(status);
            })
        },
```

```
        computed: {
            getUrl(){
                return this.url ? this.url : '/statuses';
            }
        }
    }
</script>
```

Una vez terminado lo anterior, vamos a comenzar con las solicitudes de amistad de los usuarios. Creamos un test: php artisan make:test CanRequestFriendshipTest

```
namespace Tests\Feature;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanRequestFriendshipTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function can_send_friendship_request()
    {
        $this->withoutExceptionHandling();

        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        $this->actingAs($sender)->postJson(route('friendships.store', $recipient));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'accepted' => false
        ]);
    }

    /** @test */
    public function can_accept_friendship_request()
    {
        $this->withoutExceptionHandling();

        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'accepted' => false
        ]);
```

```
        $this->actingAs($recipient)->postJson(route('request-friendships.store', $sender));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'accepted' => true
        ]);
    }
}
```

Explicación: Creamos un test para comprobar que podemos enviar solicitudes de amistad. Donde creamos el usuario que envía la solicitud y otro que la recibe. Hacemos login con el usuario sender y enviamos la solicitud al usuario recipient. Verificamos que en la base de datos la tabla friendships existe con el id del emisor y receptor y si ha sido aceptada la solicitud de amistad.

Luego creamos otro test para verificar que el usuario puede aceptar solicitudes de amistad, de forma similar al anterior .

En el archivo de rutas web.php añadimos las rutas para las solicitudes de amistad y las peticiones de amistad:

```
// Friendships routes
Route::post('friendships/{recipient}', 'FriendshipsController@store')->name('friendships.store');

// Request Friendships routes
Route::post('request-friendships/{sender}', 'RequestFriendshipsController@store')->name('request-friendships.store');
```

Creamos el controlador para las amistades:

```
php artisan make:controller FriendshipsController
```

Y dentro del FriendshipsController creado añadimos el método store, donde el sender_id es el id de la persona que realiza el postJson y el recipient_id es el parámetro que recibimos por la url:

```
namespace App\Http\Controllers;

use App\User;
use App\Models\Friendship;
use Illuminate\Http\Request;

class FriendshipsController extends Controller
{
    public function store(User $recipient)
    {
```

```
        Friendship::create([
            'sender_id' => auth()->id(),
            'recipient_id' => $recipient->id
        ]);
    }
}
```

Creamos la migración para la tabla de las amistades:

php artisan make:migration create_friendships_table

Creamos el modelo para las amistades:

php artisan make:model Models/Friendship

Deshabilitamos la protección de asignación masiva del modelo Friendship (app/models):

```
class Friendship extends Model
{
    protected $guarded = [];
}
```

Vamos a la migración de la tabla friendships (create_friendships_table), y añadimos los campos sender_id, recipient_id y accepted (este campo comprueba que la petición de amistad ha sido o no aceptada, por defecto no aceptada):

```
public function up()
{
    Schema::create('friendships', function (Blueprint $table) {
        $table->increments('id');
        $table->unsignedInteger('sender_id')->index();
        $table->unsignedInteger('recipient_id')->index();
        $table->boolean('accepted')->default(false);
        $table->timestamps();
    });
}
```

Finalmente, creamos el controlador para las solicitudes de amistad:

php artisan make:controller RequestFriendshipsController

Añadimos el metodo store al controlador creado RequestFriendshipsController, donde buscamos en la base de datos una solicitud no aceptada con la información pasada por parámetro, y aceptamos la solicitud:

```
namespace App\Http\Controllers;

use App\Models\Friendship;
use App\User;
use Illuminate\Http\Request;
```

```
class RequestFriendshipsController extends Controller
{
  public function store(User $sender)
  {
     Friendship::where([
        'sender_id' => $sender->id,
        'recipient_id' => auth()->id(),
        'accepted' => false
     ])->update(['accepted' => true]);
  }
}
```

Por último comprobamos que todos los tests pasan.

[Fin video 50]

Vamos a permitir que el usuario pueda aceptar la solicitud de amistad o denegarla.

Para ello creamos los tests correspondientes en CanRequestFriendshipTest:

```
namespace Tests\Feature;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanRequestFriendshipTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function can_create_friendship_request()
  {
     $sender = factory(User::class)->create();
     $recipient = factory(User::class)->create();

     $this->actingAs($sender)->postJson(route('friendships.store', $recipient));

     $this->assertDatabaseHas('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'pending'
     ]);
  }

  /** @test */
```

```
    public function can_delete_friendship_request()
    {
        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id
        ]);

        $this->actingAs($sender)->deleteJson(route('friendships.destroy', $recipient));

        $this->assertDatabaseMissing('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id
        ]);
    }

    /** @test */
    public function can_accept_friendship_request()
    {
        $this->withoutExceptionHandling();

        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'pending'
        ]);

        $this->actingAs($recipient)->postJson(route('accept-friendships.store', $sender));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'accepted'
        ]);
    }

    /** @test */
    public function can_deny_friendship_request()
    {
        $this->withoutExceptionHandling();

        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'pending'
```

```
        ]);

        $this->actingAs($recipient)->deleteJson(route('accept-friendships.destroy', $sender));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'denied'
        ]);
    }
}
```

Explicación: Creamos los tests para probar el aceptar o denegar la solicitud de amistad, siguiendo el esquema de los tests anteriores. También cambiamos el modo en que comprobamos el estado de la solicitud de amistad, por una propiedad status que determinara el estado de la solicitud (pendiente, aceptada o denegada).

Creamos las rutas para aceptar y denegar la amistad en el archivo de rutas web.php usando los controladores existentes.

```
// Friendships routes
Route::post('friendships/{recipient}', 'FriendshipsController@store')->name('friendships.store');
Route::delete('friendships/{recipient}', 'FriendshipsController@destroy')->name('friendships.destroy');

// Accept Friendships routes
Route::post('accept-friendships/{sender}', 'AcceptFriendshipsController@store')->name('accept-friendships.store');
Route::delete('accept-friendships/{sender}', 'AcceptFriendshipsController@destroy')->name('accept-friendships.destroy');
```

Cambiamos el nombre del controlador RequestFriendshipsController por AcceptFriendshipsController (nombre más descriptivo) refactorizando en phpstorm.

Creamos el método destroy en el FriendshipsController, y quitamos la comprobación del estado de la solicitud de amistad:

```
public function store(User $recipient)
{
    Friendship::create([
        'sender_id' => auth()->id(),
        'recipient_id' => $recipient->id
    ]);
}

public function destroy(User $recipient)
{
    Friendship::where([
        'sender_id' => auth()->id(),
        'recipient_id' => $recipient->id
    ])->delete();
```

```
}
```

Y también creamos el método destroy en el AcceptFriendshipsController, y cambiamos la comprobación del estado de la solicitud de amistad a denegado:

```
public function store(User $sender)
{
  Friendship::where([
     'sender_id' => $sender->id,
     'recipient_id' => auth()->id()
  ])->update(['status' => 'accepted']);
}

public function destroy(User $sender)
{
  Friendship::where([
     'sender_id' => $sender->id,
     'recipient_id' => auth()->id()
  ])->update(['status' => 'denied']);
}
```

Finalmente cambiamos en la migración de la tabla de amistades (create_friendships_table) cambiamos la forma de comprobar el estado de la petición de amistad por una propiedad status con 3 valores:

```
public function up()
{
  Schema::create('friendships', function (Blueprint $table) {
     $table->increments('id');
     $table->unsignedInteger('sender_id')->index();
     $table->unsignedInteger('recipient_id')->index();
     $table->enum('status', ['pending', 'accepted', 'denied'])->default('pending');
     $table->timestamps();
  });
}
```

Por último comprobamos que todos los tests de la clase CanRequestFriendshipTest pasan ejecutandolos en phpstorm.

Así ya tenemos definidas las urls para crear, eliminar, aceptar y denegar solicitudes.

[Fin video 51]

Comenzamos creando unos tests para comprobar que las 4 rutas para las amistades y solicitudes de amistad del web.php tengan el middleware auth.

Vamos al CanRequestFriendshipTest y creamos los 4 tests para comprobar que los invitados no pueden crear, borrar, aceptar o denegar solicitudes de amistad:

```
namespace Tests\Feature;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanRequestFriendshipTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function guests_users_cannot_create_friendship_request()
  {
    $recipient = factory(User::class)->create();

    $response = $this->postJson(route('friendships.store', $recipient));

    $response->assertStatus(401);
  }

  /** @test */
  public function can_create_friendship_request()
  {
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    $this->actingAs($sender)->postJson(route('friendships.store', $recipient));

    $this->assertDatabaseHas('friendships', [
      'sender_id' => $sender->id,
      'recipient_id' => $recipient->id,
      'status' => 'pending'
    ]);
  }

  /** @test */
  public function can_delete_friendship_request()
  {
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
      'sender_id' => $sender->id,
      'recipient_id' => $recipient->id
    ]);

    $this->actingAs($sender)->deleteJson(route('friendships.destroy', $recipient));
```

```
        $this->assertDatabaseMissing('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id
        ]);
    }

    /** @test */
    public function guests_users_cannot_delete_friendship_request()
    {
        $recipient = factory(User::class)->create();

        $response = $this->deleteJson(route('friendships.destroy', $recipient));

        $response->assertStatus(401);
    }

    /** @test */
    public function can_accept_friendship_request()
    {
        $this->withoutExceptionHandling();

        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'pending'
        ]);

        $this->actingAs($recipient)->postJson(route('accept-friendships.store', $sender));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'accepted'
        ]);
    }

    /** @test */
    public function guests_users_cannot_accept_friendship_request()
    {
        $user = factory(User::class)->create();

        $response = $this->postJson(route('accept-friendships.store', $user));

        $response->assertStatus(401);
    }

    /** @test */
    public function can_deny_friendship_request()
    {
        $this->withoutExceptionHandling();
```

```
        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'pending'
        ]);

        $this->actingAs($recipient)->deleteJson(route('accept-friendships.destroy', $sender));

        $this->assertDatabaseHas('friendships', [
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id,
            'status' => 'denied'
        ]);
    }

    /** @test */
    public function guests_users_cannot_deny_friendship_request()
    {
        $user = factory(User::class)->create();

        $response = $this->deleteJson(route('accept-friendships.destroy', $user));

        $response->assertStatus(401);
    }
}
```

Agregamos los middleware al archivo de rutas web.php:

```
// Friendships routes
Route::post('friendships/{recipient}', 'FriendshipsController@store')->name('friendships.store')->middleware('auth');
Route::delete('friendships/{recipient}', 'FriendshipsController@destroy')->name('friendships.destroy')->middleware('auth');

// Accept Friendships routes
Route::post('accept-friendships/{sender}', 'AcceptFriendshipsController@store')->name('accept-friendships.store')->middleware('auth');
Route::delete('accept-friendships/{sender}', 'AcceptFriendshipsController@destroy')->name('accept-friendships.destroy')->middleware('auth');
```

Y comprobamos que los test creados pasan.

Vamos a probar ahora que exista un botón en el perfil de usuario para poder solicitar su amistad. Empezamos creando un test de dusk:

```
php artisan dusk:make UsersCanRequestFriendshipTest
```

Dentro del test creado UsersCanRequestFriendshipTest agregamos la

comprobación de un botón request-friendship para enviar la solicitud:

```
namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanRequestFriendshipTest extends DuskTestCase
{
  use DatabaseMigrations;

  /**
   * @test
   * @return void
   */
  public function users_can_request_friendship()
  {
      $sender = factory(User::class)->create();
      $recipient = factory(User::class)->create();

      $this->browse(function (Browser $browser) use ($sender, $recipient) {
          $browser->loginAs($sender)
                  ->visit(route('users.show', $recipient))
                  ->press('@request-friendship')
                  ->waitForText('Solicitud enviada')
                  ->assertSee('Solicitud enviada')
          ;
      });
  }
}
```

Ahora vamos a la vista users show.blade.php y añadimos el botón mediante un componente de vue que crearemos a continuación:

```
@section('content')
  <div class="container">
      <div class="row">
          <div class="col-md-3">
              <div class="card border-0 bg-light shadow-sm">
                  <img src="{{ $user->avatar }}" alt="{{ $user->name }}" class="card-img-top">
                  <div class="card-body">
                      <h5 class="card-title">{{ $user->name }}</h5>
                                  <friendship-btn class="btn btn-primary btn-block" :recipient="{{ $user
}}"></friendship-btn>
                  </div>
              </div>
          </div>
          <div class="col-md-9">
                  <status-list
                      url="{{route('users.statuses.index', $user)}}"
```

```
                ></status-list>
            </div>
        </div>
    </div>
</div>

@endsection
```

Creamos ahora el componente de Vue FriendshipBtn en la carpeta resources/assets/js/components (mediante phpstorm).

Añadimos el componente de Vue creado al archivo app.js (resources/assets/js):

```
Vue.component('status-form', require('./components/StatusForm'));
Vue.component('status-list', require('./components/StatusList'));
Vue.component('friendship-btn', require('./components/FriendshipBtn'));
```

Y ahora en el componente de Vue creado FriendshipBtn añadimos:

```
<template>
  <button @click="sendFriendshipRequest" dusk="request-friendship">
    {{ textBtn }}
  </button>
</template>

<script>
  export default {
    props: {
        recipient: {
            type: Object,
            required: true
        }
    },
    data() {
        return {
            textBtn: 'Solicitar amistad'
        }
    },
    methods: {
        sendFriendshipRequest(){
            axios.post(`friendships/${this.recipient.name}`)
                .then(res => {
                    this.textBtn = 'Solicitud enviada';
                })
                .catch(err => {
                    console.log(err.response.data);
                })
        }
    }
  }
</script>
```

Explicación: Creamos el botón que al hacer click envíe la solicitud de amistad, y que su texto sea propiedad que podamos modificar textBtn.
En la parte del script, recibimos el usuario a enviar la solicitud como una propiedad. Creamos el método para enviar la solicitud usando la propiedad anterior, cambiando el texto del botón usando la propiedad textBtn.

Comprobamos que finalmente el test pasa:

```
php artisan dusk --filter users_can_request_friendship
```

Por último vamos a modificar el seeder de usuarios para que al actualizar las migraciones cree unos estados por defecto.
Para ello abrimos el UserTableSeeder (database/seeds) y agregamos un factory para crear los estados, y añadimos un nombre al factory de crear un usuario:

```
use App\Models\Status;
use App\User;
use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
{
  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
     User::truncate();
     Status::truncate();

     factory(User::class)->create([
        'name' => 'angel',
        'email' => 'angel@email.com'
     ]);
     factory(Status::class, 10)->create();

  }
}
```

Finalmente actualizamos las migraciones con los seeds:

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

Y podemos comprobar que en el navegador se muestra correctamente.

[Fin video 52]

Vamos a solucionar el problema de duplicación de peticiones de amistad, y permitir que el usuario pueda dar y quitar la solicitud de amistad..

Modificamos los tests de crear y borrar una solicitud de amistad del CanRequestFriendshipTest, para que solo haya una solicitud de amistad en base de datos. Verificamos también la respuesta para que sea tipo json con estado pendiente al crearla, o de tipo deleted al borrarla:

```
/** @test */
public function can_create_friendship_request()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $response = $this->actingAs($sender)->postJson(route('friendships.store', $recipient));

  $response->assertJson([
     'friendship_status' => 'pending'
  ]);

  $this->assertDatabaseHas('friendships', [
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id,
     'status' => 'pending'
  ]);

  $this->actingAs($sender)->postJson(route('friendships.store', $recipient));
  $this->assertCount(1, Friendship::all());
}

/** @test */
public function can_delete_friendship_request()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id
  ]);

  $response = $this->actingAs($sender)->deleteJson(route('friendships.destroy', $recipient));

  $response->assertJson([
     'friendship_status' => 'deleted'
  ]);

  $this->assertDatabaseMissing('friendships', [
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id
  ]);
```

```
}
```

Vamos al test de dusk UsersCanRequestFriendshipTest y modificamos para que al pulsar el botón otra vez borre la amistad, y modificando el texto a Cancelar solicitud:

```
public function senders_can_create_and_delete_friendship_requests()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $this->browse(function (Browser $browser) use ($sender, $recipient) {
    $browser->loginAs($sender)
        ->visit(route('users.show', $recipient))
        ->press('@request-friendship')
        ->waitForText('Cancelar solicitud')
        ->assertSee('Cancelar solicitud')
        ->visit(route('users.show', $recipient))
        ->assertSee('Cancelar solicitud')
        ->press('@request-friendship')
        ->waitForText('Solicitar amistad')
        ->assertSee('Solicitar amistad')
    ;
  });
}
```

Vamos al FriendshipBtn.vue y modificamos:

```
<template>
  <button @click="toggleFriendshipStatus">
    {{ getText }}
  </button>
</template>

<script>
  export default {
    props: {
      recipient: {
        type: Object,
        required: true
      },
      friendshipStatus: {
        type: String,
        required: true
      }
    },
    data(){
      return {
        localFriendshipStatus: this.friendshipStatus
      }
    },
    methods: {
```

```
        toggleFriendshipStatus(){
            let method = this.getMethod();

            axios[method](`friendships/${this.recipient.name}`)
                .then(res => {
                    this.localFriendshipStatus = res.data.friendship_status;
                })
                .catch(err => {
                    console.log(err.response.data);
                })
        },
        getMethod(){
            if (this.localFriendshipStatus === 'pending'){
                return 'delete';
            }
            return 'post';
        }
    },
    computed: {
        getText(){
            if (this.localFriendshipStatus === 'pending')
            {
                return 'Cancelar solicitud';
            }
            return 'Solicitar amistad';
        }
    }
  }
</script>
```

Explicación: Primero cambiamos el nombre a la variable textBtn por getText como propiedad calculada, que dependiendo del estado de la solicitud (aceptada o cancelada) cambie el texto a mostrar en el botón.

Continuamos añadiendo el estado de la solicitud como la propiedad `friendshipStatus` que usaremos como la variable `localFriendshipStatus` para evitar mutar propiedades directamente en vuejs (ocurre un warning en el navegador).

Empleamos el mismo método para enviar y cancelar solicitudes `toggleFriendshipStatus`, cambiará el tipo de petición (post o delete) dependiendo del estado de la solicitud de amistad mediante el método getMethod.

En show.blade.php le pasamos la propiedad friendship-status como parámetro, y el elemento de dusk que quitamos al botón del template de FriendshipBtn:

```
<friendship-btn
  dusk="request-friendship"
  class="btn btn-primary btn-block"
```

```
  friendship-status="{{ $friendshipStatus }}"
  :recipient="{{ $user }}"
></friendship-btn>
```

Seguidamente en el UsersController construimos la variable friendshipStatus:

```
public function show(User $user)
{
  $friendshipStatus = optional(Friendship::where([
     'recipient_id' => $user->id,
     'sender_id' => auth()->id()
  ])->first())->status; // devuelve null si no autenticado, uso optional para que falle si null

  return view('users.show', compact('user', 'friendshipStatus'));
}
```

Finalmente el el FriendshipsController, cambiamos el método create a firstorCreate para evitar duplicaciones. Devolvemos una respuesta Json con estado pendiente al crear la solicitud, y estado deleted al quitar la solicitud:

```
public function store(User $recipient)
{
  Friendship::firstOrCreate([
     'sender_id' => auth()->id(),
     'recipient_id' => $recipient->id
  ]);

  return response()->json([
     'friendship_status' => 'pending'
  ]);
}

public function destroy(User $recipient)
{
  Friendship::where([
     'sender_id' => auth()->id(),
     'recipient_id' => $recipient->id
  ])->delete();

  return response()->json([
     'friendship_status' => 'deleted'
  ]);
}
```

Ejecutamos los tests de phpunit y dusk y comprobamos que todo está Ok.

[Fin video 53]

Continuamos con la aceptación de solicitudes de amistad.

Primero vamos al UsersCanRequestFriendshipTest y usando el test ya escrito, creamos uno nuevo para comprobar que los que reciben la solicitud pueden aceptar y denegar la solicitud de amistad:

```
/**
* @test
* @return void
*/
public function recipients_can_accept_and_deny_friendship_requests()
{
   $sender = factory(User::class)->create();
   $recipient = factory(User::class)->create();

   Friendship::create([
      'sender_id' => $sender->id,
      'recipient_id' => $recipient->id
   ]);

   $this->browse(function (Browser $browser) use ($sender, $recipient) {
      $browser->loginAs($recipient)
         ->visit(route('accept-friendships.index'))
         ->assertSee($sender->name)
         ->press('@accept-friendship')
         ->waitForText('son amigos')
         ->assertSee('son amigos')
         ->visit(route('accept-friendships.index'))
         ->assertSee('son amigos')
      ;
   });
}
```

Explicación: Creamos una nueva solicitud, hacemos login como la persona que recibe la solicitud, visitamos una nueva ruta donde aparecerán las solicitudes que tengamos. Comprobamos el nombre de la persona que envió la solicitud, presionamos el boton de aceptar amistad y vemos que pone que son amigos tanto en la página como al visitar la nueva ruta.

Añadimos la nueva ruta definida anteriormente al archivo de rutas web.php:

```
// Accept Friendships routes
Route::get('friends/requests', 'AcceptFriendshipsController@index')->name('accept-friendships.index');
Route::post('accept-friendships/{sender}', 'AcceptFriendshipsController@store')->name('accept-friendships.store')->middleware('auth');
Route::delete('accept-friendships/{sender}', 'AcceptFriendshipsController@destroy')->name('accept-friendships.destroy')->middleware('auth');
```

Ahora añadimos el método index usado en la ruta en el AcceptFriendshipsController, donde devolvemos la vista index con los datos de todas las solicitudes del usuario, (con metodo with para precargar la relación sender):

```
public function index()
{
  $friendshipRequests = Friendship::with('sender')->where([
    'recipient_id' => auth()->id()
  ])->get();

  return view('friendships.index', compact('friendshipRequests'));
}
```

Creamos la vista index.blade.php dentro de la carpeta resources/views en una carpeta llamada friendships que también creamos:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    @foreach($friendshipRequests as $friendshipRequest)
      <accept-friendship-btn
        dusk="accept-friendship"
        :sender="{{ $friendshipRequest->sender }}"
        friendship-status="{{ $friendshipRequest->status }}"
      ></accept-friendship-btn>
    @endforeach
  </div>
@endsection
```

Explicación: Usamos un componente de Vue llamado accept-friendship-btn que definiremos más adelante, al que agregamos el selector de dusk, el sender y el estado de la solicitud. Agregamos también los scripts para ejecutar vue usando el template layouts con la sección content.

Creamos un nuevo test unitario para el modelo Friendship:

```
php artisan make:test Models/FriendshipTest --unit
```

Abrimos el test creado, y creamos un par de tests para comprobar que una solicitud de amistad pertenece a alguien que envía, y

```
namespace Tests\Unit\Models;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
```

```
class FriendshipTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function a_friendship_request_belongs_to_a_sender()
  {
      $sender = factory(User::class)->create();

      $friendship = factory(Friendship::class)->create(['sender_id' => $sender->id]);

      $this->assertInstanceOf(User::class, $friendship->sender);

  }

  /** @test */
  public function a_friendship_request_belongs_to_a_recipient()
  {
      $recipient = factory(User::class)->create();

      $friendship = factory(Friendship::class)->create(['recipient_id' => $recipient->id]);

      $this->assertInstanceOf(User::class, $friendship->recipient);

  }
}
```

Creamos el factory para el modelo Friendship:

```
php artisan make:factory FriendshipFactory -m Models\\Friendship
```

Vamos al FriendshipFactory creado y añadimos un valor por defecto al recipient_id y al sender_id:

```
use App\User;
use Faker\Generator as Faker;

$factory->define(App\Models\Friendship::class, function (Faker $faker) {
  return [
      'recipient_id' => function(){
          return factory(User::class)->create();
      },
      'sender_id' => function(){
          return factory(User::class)->create();
      }
  ];
});
```

Continuamos en el modelo Friendship (app/models) definiendo la relación de sender y recipient pertenecen a la clase user:

```
use App\User;
use Illuminate\Database\Eloquent\Model;

class Friendship extends Model
{
   protected $guarded = [];

   public function sender()
   {
      return $this->belongsTo(User::class);
   }

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

Creamos un nuevo componente de Vue llamado AcceptFriendshipBtn en la carpeta (resources/assets/js/components):

```
<template>
   <div v-if="localFriendshipStatus === 'pending'">
      <span v-text="sender.name"></span> te ha enviado una solicitud de amistad
      <button @click="acceptFriendshipRequest">Aceptar solicitud</button>
   </div>
   <div v-else>
      Tu y <span v-text="sender.name"></span> son amigos
   </div>
</template>

<script>
   export default {
      props:{
         sender: {
            type: Object,
            required: true
         },
         friendshipStatus:{
            type: String,
            required: true
         }
      },
      data(){
         return{
            localFriendshipStatus: this.friendshipStatus
         }
      },
      methods:{
         acceptFriendshipRequest(){
            axios.post(`/accept-friendships/${this.sender.name}`)
               .then(res => [
```

```
                    this.localFriendshipStatus = 'accepted'
                ])
                .catch(err => [
                    console.log(err.response.data)
                ])
            }
        }
    }
</script>
```

Explicacion: Creamos en el template un boton con el método acceptFriendshipRequest que hace una petición a la url accept-friendships con el nombre del sender que recibimos como propiedad. Creamos también una copia local del friendshipStatus.
Cuando recibamos una respuesta positiva cambiamos el estado a accepted.

Y mostramos el botón dependiendo de si el estado de la solicitud es pendiente.

Finalmente añadimos el componente de Vue creado al archivo app.js (resources/assets/js):

```
Vue.component('status-form', require('./components/StatusForm'));
Vue.component('status-list', require('./components/StatusList'));
Vue.component('friendship-btn', require('./components/FriendshipBtn'));
Vue.component('accept-friendship-btn', require('./components/AcceptFriendshipBtn'));
```

Compilamos los cambios realizados: npm run dev
Y ejecutamos los tests de phpunit y dusk para ver que pasan.

[Fin video 54]

Para solucionar que al cambiar el estado de la solicitud de amistad a denied, sigue mostrando el mensaje ‘Tu y xx son amigos’ en la url social.local/friends/requests.

Para ello vamos a agregar un nuevo botón que permita denegar la solicitud.

Empezamos en UsersCanRequestFriendshipTest modificando el nombre del ultimo test creado por recipients_can_accept_friendship_requests pues vamos a crear un nuevo test para denegar las solicitudes.
Añadimos usando el test creado anteriormente como modelo:

```
/**
* @test
* @return void
*/
public function recipients_can_deny_friendship_requests()
```

```
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id
  ]);

  $this->browse(function (Browser $browser) use ($sender, $recipient) {
     $browser->loginAs($recipient)
        ->visit(route('accept-friendships.index'))
        ->assertSee($sender->name)
        ->press('@deny-friendship')
        ->waitForText('Solicitud denegada')
        ->assertSee('Solicitud denegada')
        ->visit(route('accept-friendships.index'))
        ->assertSee('Solicitud denegada')
     ;
  });
}
```

Ahora vamos al componente de vue AcceptFriendshipBtn, al que añadimos el botón de denegar solicitud junto con el método para denegarla. Cambiamos también que se muestre un texto diferente dependiendo del estado de la solicitud, y accedemos directamente a la respuesta con res.data.friendship_status:

```
<template>
  <div v-if="localFriendshipStatus === 'pending'">
     <span v-text="sender.name"></span> te ha enviado una solicitud de amistad
     <button @click="acceptFriendshipRequest">Aceptar solicitud</button>
                  <button   dusk="deny-friendship"   @click="denyFriendshipRequest">Denegar
solicitud</button>
  </div>

  <div v-else-if="localFriendshipStatus === 'accepted'">
     Tu y <span v-text="sender.name"></span> son amigos
  </div>

  <div v-else-if="localFriendshipStatus === 'denied'">
     Solicitud denegada de <span v-text="sender.name"></span>
  </div>
</template>

<script>
  export default {
     props:{
        sender: {
           type: Object,
           required: true
        },
```

```
            friendshipStatus:{
                type: String,
                required: true
            }
        },
        data(){
            return{
                localFriendshipStatus: this.friendshipStatus
            }
        },
        methods:{
            acceptFriendshipRequest(){
                axios.post(`/accept-friendships/${this.sender.name}`)
                    .then(res => [
                        this.localFriendshipStatus = res.data.friendship_status
                    ])
                    .catch(err => [
                        console.log(err.response.data)
                    ])
            },
            denyFriendshipRequest(){
                axios.delete(`/accept-friendships/${this.sender.name}`)
                    .then(res => [
                        this.localFriendshipStatus = res.data.friendship_status
                    ])
                    .catch(err => [
                        console.log(err.response.data)
                    ])
            }
        }
    }
</script>
```

Continuamos al ir al test CanRequestFriendshipTest, y añadimos la comprobación de esperar una respuesta tipo json del estado de la solicitud de amistad:

```
public function can_accept_friendship_request()
{
    $this->withoutExceptionHandling();

    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'pending'
    ]);

        $response = $this->actingAs($recipient)->postJson(route('accept-friendships.store',
$sender));
```

```
    $response->assertJson([
        'friendship_status' => 'accepted'
    ]);

    $this->assertDatabaseHas('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'accepted'
    ]);
}

        ( … más abajo …)

public function can_deny_friendship_request()
{
    $this->withoutExceptionHandling();

    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'pending'
    ]);

    $response = $this->actingAs($recipient)->deleteJson(route('accept-friendships.destroy',
$sender));

    $response->assertJson([
        'friendship_status' => 'denied'
    ]);

    $this->assertDatabaseHas('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'denied'
    ]);
}
```

Y añadimos esta respuesta json en el AcceptFriendshipsController en los métodos store y destroy:

```
public function store(User $sender)
{
    Friendship::where([
        'sender_id' => $sender->id,
        'recipient_id' => auth()->id()
    ])->update(['status' => 'accepted']);

    return response()->json([
        'friendship_status' => 'accepted'
    ]);
```

```
}

public function destroy(User $sender)
{
  Friendship::where([
     'sender_id' => $sender->id,
     'recipient_id' => auth()->id()
  ])->update(['status' => 'denied']);

  return response()->json([
     'friendship_status' => 'denied'
  ]);
}
```

Compilamos los cambios realizados: npm run dev
Y ejecutamos los tests de phpunit y dusk para ver que pasan.

[Fin video 55]

Vamos a agregar un botón para poder eliminar la solicitud en cualquier momento.

Comenzamos creando un nuevo test en UsersCanRequestFriendshipTest para probar la funcionalidad de eliminar la solicitud de amistad:

```
/**
* @test
* @return void
*/
public function recipients_can_delete_friendship_requests()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id
  ]);

  $this->browse(function (Browser $browser) use ($sender, $recipient) {
     $browser->loginAs($recipient)
        ->visit(route('accept-friendships.index'))
        ->assertSee($sender->name)
        ->press('@delete-friendship')
        ->waitForText('Solicitud eliminada')
        ->assertSee('Solicitud eliminada')
        ->visit(route('accept-friendships.index'))
        ->assertDontSee('Solicitud eliminada')
        ->assertDontSee($sender->name)
```

```
    ;
  });
}
```

Explicacion: Siguiendo el mismo patrón que en los tests anteriores, teniendo un usuario que envía, otro que recibe y una solicitud pendiente.
Hacemos login como el usuario que recibe, visitamos la ruta, verificamos el nombre de la persona que envia y presionamos el botón delete-friendship.
Luego espera que muestre el texto Solicitud eliminada, y comprobamos que al visitar la ruta otra vez no aparezca el texto anterior ni el nombre del que envía.

Agregamos a continuación en el componente AcceptFriendshipBtn.vue el boton delete-friendship, de forma que se muestre siempre sin importar el estado de la solicitud. Añadiendo también el método deleteFriendship para ello:

```
<template>
<div>
  <div v-if="localFriendshipStatus === 'pending'">
    <span v-text="sender.name"></span> te ha enviado una solicitud de amistad
    <button @click="acceptFriendshipRequest">Aceptar solicitud</button>
            <button dusk="deny-friendship" @click="denyFriendshipRequest">Denegar
solicitud</button>
  </div>

  <div v-else-if="localFriendshipStatus === 'accepted'">
    Tu y <span v-text="sender.name"></span> son amigos
  </div>

  <div v-else-if="localFriendshipStatus === 'denied'">
    Solicitud denegada de <span v-text="sender.name"></span>
  </div>

  <button v-if="localFriendshipStatus === 'deleted'">Solicitud eliminada</button>
  <button v-else dusk="delete-friendship" @click="deleteFriendship">Eliminar</button>
</div>
</template>

<script>
  export default {
    props:{
      sender: {
        type: Object,
        required: true
      },
      friendshipStatus:{
        type: String,
        required: true
      }
    },
    data(){
```

```
            return{
                localFriendshipStatus: this.friendshipStatus
            }
        },
        methods:{
            acceptFriendshipRequest(){
                axios.post(`/accept-friendships/${this.sender.name}`)
                    .then(res => [
                        this.localFriendshipStatus = res.data.friendship_status
                    ])
                    .catch(err => [
                        console.log(err.response.data)
                    ])
            },
            denyFriendshipRequest(){
                axios.delete(`/accept-friendships/${this.sender.name}`)
                    .then(res => [
                        this.localFriendshipStatus = res.data.friendship_status
                    ])
                    .catch(err => [
                        console.log(err.response.data)
                    ])
            },
            deleteFriendship(){
                axios.delete(`/friendships/${this.sender.name}`)
                    .then(res => [
                        this.localFriendshipStatus = res.data.friendship_status
                    ])
                    .catch(err => [
                        console.log(err.response.data)
                    ])
            }
        }
    }
</script>
```

Continuamos en el FriendshipController, guardando la solicitud en una variable para luego retornar el estado usando la variable (usando fresh() para obtener una nueva copia del friendship para poder obtener el estado).
Además, mostramos el texto deleted (con operador ternario) si se ha eliminado el recurso, y nada si no se ha eliminado.
Y modificando el método destroy para que funcione con el sender y recipient a la vez, añadiendo el orWhere con los parámetros invertidos.

```
public function store(User $recipient)
{
    $friendship = Friendship::firstOrCreate([
        'sender_id' => auth()->id(),
        'recipient_id' => $recipient->id
    ]);
```

```
    return response()->json([
      'friendship_status' => $friendship->fresh()->status
    ]);
  }

  public function destroy(User $user)
  {
    $deleted = Friendship::where([
        'sender_id' => auth()->id(),
        'recipient_id' => $user->id
    ])->orWhere([
        'sender_id' => $user->id,
        'recipient_id' => auth()->id()
    ])->delete();

    return response()->json([
        'friendship_status' => $deleted ? 'deleted' : ''
    ]);
  }
```

Creamos en el CanRequestFriendshipTest un nuevo test para que compruebe que el método destroy modificado del FriendshipController, sirve tanto para el que envía como para el que recibe.
Para ello cambiamos el nombre al test can_delete_friendship_request y creando otro para el recipient:

```
  /** @test */
  public function senders_can_delete_sent_friendship_request()
  {
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id
    ]);

    $response = $this->actingAs($sender)->deleteJson(route('friendships.destroy', $recipient));

    $response->assertJson([
        'friendship_status' => 'deleted'
    ]);

    $this->assertDatabaseMissing('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id
    ]);
  }

  /** @test */
  public function recipients_can_delete_received_friendship_request()
```

```
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id
    ]);

    $response = $this->actingAs($recipient)->deleteJson(route('friendships.destroy', $sender));

    $response->assertJson([
        'friendship_status' => 'deleted'
    ]);

    $this->assertDatabaseMissing('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id
    ]);
}
```

Finalmente vamos al archivo de rutas web.php para modificar el nombre de variable que pasamos por parámetro en una de las rutas, al modificarla anteriormente:

```
// Friendships routes
Route::post('friendships/{recipient}', 'FriendshipsController@store')->name('friendships.store')->middleware('auth');
Route::delete('friendships/{user}', 'FriendshipsController@destroy')->name('friendships.destroy')->middleware('auth');
```

Compilamos los cambios realizados: npm run dev
Y ejecutamos los tests de phpunit y dusk para ver que pasan.

[Fin video 56]

Vamos a cambiar que cuando el usuario acepta la solicitud de amista muestre Eliminar amistad en vez de Solicitar amistad como muestra ahora mismo.

Comenzamos en el UsersCanRequestFriendshipTest creando un nuevo test para probar que los que envían pueden eliminar las solicitudes aceptadas:

```
/**
* @test
* @return void
*/
public function senders_can_delete_accepted_friendship_requests()
{
    $sender = factory(User::class)->create();
```

```
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'accepted'
    ]);

    $this->browse(function (Browser $browser) use ($sender, $recipient) {
        $browser->loginAs($sender)
            ->visit(route('users.show', $recipient))
            ->assertSee('Eliminar de mis amigos')
            ->press('@request-friendship')
            ->waitForText('Solicitar amistad')
            ->assertSee('Solicitar amistad')
            ->visit(route('users.show', $recipient))
            ->assertSee('Solicitar amistad')
        ;
    });
}
```

Explicación: Necesitamos un usuario que manda y otro que recibe la solicitud, una solicitud creada con estatus aceptado. Luego comprobamos que vemos el texto Eliminar de mis amigos al visitar el perfil del usuario al que solicitamos amistad, y si hacemos click en el botón que efectivamente hemos eliminado la amistad.

Ahora vamos a comprobar que el usuario no pueda eliminar una solicitud de amistad denegada (pues no tiene sentido que pueda volver a mandar la solicitud de amistad al usuario si este ya la ha denegado por no querer ser amigo).

Para ello volvemos al UsersCanRequestFriendshipTest y creamos un nuevo test con la solicitud en estado denegado y comprobamos que vemos Solicitud denegada:

```
/**
* @test
* @return void
*/
public function senders_cannot_delete_denied_friendship_requests()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'denied'
    ]);

    $this->browse(function (Browser $browser) use ($sender, $recipient) {
        $browser->loginAs($sender)
```

```
            ->visit(route('users.show', $recipient))
            ->assertSee('Solicitud denegada')
            ->press('@request-friendship')
            ->waitForText('Solicitud denegada')
            ->assertSee('Solicitud denegada')
            ->visit(route('users.show', $recipient))
            ->assertSee('Solicitud denegada')
        ;
    });
}
```

Agregamos los cambios al FriendshipBtn.vue para que pasen los tests:

```
( … )
        getMethod(){
                if (this.localFriendshipStatus === 'pending' || this.localFriendshipStatus === 'accepted'){
                    return 'delete';
                }
                return 'post';
            }
        },
        computed: {
            getText(){
                if (this.localFriendshipStatus === 'pending')
                {
                    return 'Cancelar solicitud';
                }
                if (this.localFriendshipStatus === 'accepted')
                {
                    return 'Eliminar de mis amigos';
                }
                if (this.localFriendshipStatus === 'denied')
                {
                    return 'Solicitud denegada';
                }
                return 'Solicitar amistad';
            }
        }
    }
</script>
```

Ahora para evitar que la lógica anterior se pueda modificar en el navegador del usuario, es necesario añadir la comprobación al backend.

Por ello, añadimos un nuevo test al CanRequestFriendshipTest para comprobar que los que envían no pueden eliminar las solicitudes denegadas. Mediante la creación de una solicitud con estado denegado comprobamos que al hacer el post el estado sigue denegado y en la base de datos sigue existiendo la solicitud:

```
/** @test */
public function senders_cannot_delete_denied_friendship_request()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id,
     'status' => 'denied'
  ]);

  $response = $this->actingAs($sender)->deleteJson(route('friendships.destroy', $recipient));

  $response->assertJson([
     'friendship_status' => 'denied'
  ]);

  $this->assertDatabaseHas('friendships', [
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id,
     'status' => 'denied'
  ]);
}
```

Finalmente, en el FriendshipController necesitamos que al obtener la solicitud de la base de datos, comprobar que si la solicitud está denegada devolvemos un json con el estado denied. En otros casos, si se elimina la solicitud devuelve estado deleted.

```
public function destroy(User $user)
{
  $friendship = Friendship::where([
     'sender_id' => auth()->id(),
     'recipient_id' => $user->id
  ])->orWhere([
     'sender_id' => $user->id,
     'recipient_id' => auth()->id()
  ])->first();

  if($friendship->status === 'denied')
  {
     return response()->json([
        'friendship_status' => 'denied'
     ]);
  }

  return response()->json([
     'friendship_status' => $friendship->delete() ? 'deleted' : ''
  ]);
}
```

Compilamos los cambios realizados: npm run dev
Y ejecutamos los tests de phpunit y dusk para ver que pasan.

[Fin video 57]

Vamos a actualizar laravel de 5.6.* a la versión 5.7.*

Empezamos siguiendo la documentación oficial para hacerlo:
https://laravel.com/docs/5.7/upgrade

Actualizamos la versión de laravel en el composer.json a 5.7.*:

```
"require": {
   "php": "^7.1.3",
   "fideloper/proxy": "^4.0",
   "laravel/framework": "5.7.*",
   "laravel/tinker": "^1.0"
},
```

Vamos a eliminar la carpeta assets dentro de resources, pero antes sacamos las 2 carpetas que contiene (js y sass) y las movemos a la carpeta resources.
Una vez movidas las carpetas, eliminamos la carpeta assets.

Ahora en el archivo webpack.mix.js cambiamos la ruta a las carpetas anteriores por la nueva ruta actual correcta:

```
mix.js('resources/js/app.js', 'public/js')
   .sass('resources/sass/app.scss', 'public/css');
```

Compilamos los cambios realizados: npm run dev

Modificamos el archivo .gitignore de storage/framework/cache por:

```
*
!data/
!.gitignore
```

Ejecutamos en la terminal (llevará un rato): composer update

Ejecutamos los tests de phpunit y dusk para comprobar que pasan OK.

Podemos ver la nueva versión de laravel con: php artisan -V

Si mostramos una pagina de error de la aplicacion (por ej: http://social.local/sdfsdf) vemos que ha cambiado pero que no muestra una ilustración svg.
Para añadir estas ilustraciones, dentro de la carpeta public de nuestro proyecto vamos a crear la carpeta svg y dentro de ella añadir las distintas imágenes que guardaremos a la carpeta svg creada:
https://raw.githubusercontent.com/laravel/laravel/5.7/public/svg/403.svg
https://raw.githubusercontent.com/laravel/laravel/5.7/public/svg/404.svg
https://raw.githubusercontent.com/laravel/laravel/5.7/public/svg/500.svg
https://raw.githubusercontent.com/laravel/laravel/5.7/public/svg/503.svg

En mi caso la página de error que muestra Laravel sigue siendo la misma que anteriormente, por lo que este ultimo añadido de las imágenes svg no me son necesarias.

[Fin video 58]

Vamos a solucionar el problema de que al visitar nuestro perfil de usuario, aparece el botón para solicitar amistad que si damos click nos enviamos una solicitud de amistad a nosotros mismos.

Para ello comenzamos creando un nuevo test en CanRequestFriendshipTest que usando un usuario y haciendo un post a la ruta de las solicitudes de amistad, compruebe que en la base de datos no exista un registro con los datos:

```
/** @test */
public function a_user_cannot_send_friend_request_to_itself()
{
    $sender = factory(User::class)->create();

    $this->actingAs($sender)->postJson(route('friendships.store', $sender));

    $this->assertDatabaseMissing('friendships', [
        'sender_id' => $sender->id,
        'recipient_id' => $sender->id,
        'status' => 'pending'
    ]);
}
```

Para arreglar esto, vamos al FriendshipController y hacemos la verificación en el método store, abortando con código 400:

```
public function store(User $recipient)
{
    if (auth()->id() === $recipient->id)
    {
```

```
        abort(400);
    }
    $friendship = Friendship::firstOrCreate([
        'sender_id' => auth()->id(),
        'recipient_id' => $recipient->id
    ]);

    return response()->json([
      'friendship_status' => $friendship->fresh()->status
    ]);
}
```

Para quitar el botón, primero añadimos un nuevo test a UsersCanRequestFriendshipTest para que un usuario no puede enviarse solicitudes a sí mismo, comprobando que no vemos el botón, y aparece el texto Eres tu:

```
/**
* @test
* @return void
*/
public function a_user_cannot_send_friend_request_to_itself()
{
    $user = factory(User::class)->create();

    $this->browse(function (Browser $browser) use ($user) {
        $browser->loginAs($user)
            ->visit(route('users.show', $user))
            ->assertMissing('@request-friendship')
            ->assertSee('Eres tu')
        ;
    });
}
```

Hacemos la verificación en la vista de users show.blade.php , preguntando si el id del usuario autenticado es el mismo del usuario.
En ese caso muestra no muestra el botón y muestra el texto Eres tu:

```
@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-3">
                <div class="card border-0 bg-light shadow-sm">
                    <img src="{{ $user->avatar }}" alt="{{ $user->name }}" class="card-img-top">
                    <div class="card-body">
                        @if(auth()->id() === $user->id)
                            <h5 class="card-title">{{ $user->name }} <small class="text-secondary">Eres
tu</small></h5>
                        @else
                            <h5 class="card-title">{{ $user->name }}</h5>
                            <friendship-btn
                                dusk="request-friendship"
```

```
                        class="btn btn-primary btn-block"
                        friendship-status="{{ $friendshipStatus }}"
                        :recipient="{{ $user }}"
                    ></friendship-btn>
                @endif
            </div>
        </div>
    </div>
```

Vamos a corregir que si vamos a la ruta social.local/friends/requests sin estar autenticados no esté protegida, y lo cambiamos a que redireccione al login.

Primero vamos a añadir la comprobación a uno de los tests existentes en CanRequestFriendshipTest para que redirecciona al login si es un invitado:

```
/** @test */
public function guests_users_cannot_accept_friendship_request()
{
  $user = factory(User::class)->create();

  $this->postJson(route('accept-friendships.store', $user))->assertStatus(401);

  $this->get(route('accept-friendships.index'))->assertRedirect('login');
}
```

Y luego en el archivo de rutas web.php añadimos el middleware a la ruta:

```
// Accept Friendships routes
Route::get('friends/requests', 'AcceptFriendshipsController@index')->name('accept-friendships.index')->middleware('auth');
Route::post('accept-friendships/{sender}', 'AcceptFriendshipsController@store')->name('accept-friendships.store')->middleware('auth');
Route::delete('accept-friendships/{sender}', 'AcceptFriendshipsController@destroy')->name('accept-friendships.destroy')->middleware('auth');
```

Ahora vamos a solucionar que si vamos a cualquier perfil sin estar autenticados aparezca el botón de solicitar amistad.
Para ello cambiamos que al hacer click redirecciona al login.

En el test de dusk UsersCanRequestFriendshipTest creamos un nuevo test para verificar que al presionar el botón siendo invitado, la nueva url es el login:

```
/**
* @test
* @return void
*/
public function guests_cannot_create_friendship_requests()
```

```
{
    $recipient = factory(User::class)->create();

    $this->browse(function (Browser $browser) use ($recipient) {
        $browser->visit(route('users.show', $recipient))
            ->press('@request-friendship')
            ->assertPathIs('/login')
        ;
    });
}
```

Finalmente en el FriendshipBtn.vue añadimos la lógica para que redireccione al presionar el boton siendo invitado, en el método toggle:

```
methods: {
    toggleFriendshipStatus(){
        this.redirectIfGuest();

        let method = this.getMethod();

        axios[method](`friendships/${this.recipient.name}`)
            .then(res => {
                this.localFriendshipStatus = res.data.friendship_status;
            })
            .catch(err => {
                console.log(err.response.data);
            })
    },
```

Compilamos los cambios realizados: npm run dev
Ejecutamos los tests de phpunit y dusk para ver que pasan.
Y vemos que los cambios realizados se muestran en el navegador.

[Fin video 59]

Actualmente hay un bug en la aplicación, si intentamos eliminar una solicitud de amistad que esté en estado denegado desde nuestro perfil de usuario, en vez de eliminarla lo que hace la aplicación es borrar la primera solicitud que esté en estado pendiente.

Para captar este error, vamos a reproducir este error en un test para luego solucionarlo.

El error está ocurriendo en la parte de la búsqueda en base de datos del método destroy del FriendshipController.

Vamos a aislar esta búsqueda (query) en un scope, testear el método en aislamiento y recrear el error en el test unitario.

Comenzamos sustituyendo la búsqueda en base de datos del método destroy del FriendshipController por el método betweenUsers (que crearemos) del modelo Friendship que pasamos los 2 usuarios a encontrar:

```
$friendship = Friendship::betweenUsers(auth()->user(), $user)->first();
```

Creamos un nuevo test en FriendshipTest para probar lo anterior:

```
/** @test */
public function can_find_friendships_by_sender_and_recipient()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  factory(Friendship::class, 2)->create(['recipient_id' => $recipient->id]);
  factory(Friendship::class, 2)->create(['sender_id' => $sender->id]);

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id
     ]);

  $foundFriendship = Friendship::betweenUsers($sender, $recipient)->first();

  $this->assertEquals($sender->id, $foundFriendship->sender_id);
  $this->assertEquals($recipient->id, $foundFriendship->recipient_id);

  $foundFriendship2 = Friendship::betweenUsers($recipient, $sender)->first();

  $this->assertEquals($sender->id, $foundFriendship2->sender_id);
  $this->assertEquals($recipient->id, $foundFriendship2->recipient_id);
}
```

Explicación: Usando un usuario que recibe, otro que envía, una solicitud a encontrar. Llamamos al método betweenUsers con el sender y recipient, y verificamos que el sender_id es igual al sender_id de la solicitud, e igualmente para el recipient_id.
Para reproducir el error, creamos 2 solicitudes adicionales para el recipient_id y para el sender_id. Y comprobamos que invirtiendo el recipient y el sender, seguimos recibiendo el sender_id y el recipient_id de forma correcta.

Añadimos el método betweenUsers (utilizando el prefijo scope, pues no accedemos desde una instancia sino directamente de la clase, y pasando el parámetro query por ser un scope) al modelo Friendship.php:

```
public function scopeBetweenUsers($query, $sender, $recipient)
{
  $query->where([
     ['sender_id', $sender->id],
     ['recipient_id', $recipient->id]
  ])->orWhere([
     ['sender_id', $recipient->id],
     ['recipient_id', $sender->id]
  ]);
}
```

Vamos a permitir que la persona que recibe la solicitud pueda eliminar la solicitud denegada.
Para ello en el CanRequestFriendshipTest creamos un nuevo test siguiendo el ejemplo del senders_cannot_delete_denied_friendship_request actuando como el recipient en el deleteJson con el sender como parámetro, esperando una respuesta con el estado deleted y comprobando que en la base de datos se ha eliminado:

```
/** @test */
public function recipients_can_delete_denied_friendship_request()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  Friendship::create([
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id,
     'status' => 'denied'
  ]);

  $response = $this->actingAs($recipient)->deleteJson(route('friendships.destroy', $sender));

  $response->assertJson([
     'friendship_status' => 'deleted'
  ]);

  $this->assertDatabaseMissing('friendships', [
     'sender_id' => $sender->id,
     'recipient_id' => $recipient->id,
     'status' => 'denied'
  ]);
}
```

Ahora en el FriendshipController modificamos la parte que nos impide eliminar la solicitud, ampliando la comprobación a que el sender_id es igual al id del usuario que intenta eliminarla:

```
public function destroy(User $user)
{
  $friendship = Friendship::betweenUsers(auth()->user(), $user)->first();
```

```
    if($friendship->status === 'denied' && (int) $friendship->sender_id === auth()->id())
    {
        return response()->json([
            'friendship_status' => 'denied'
        ]);
    }

    return response()->json([
        'friendship_status' => $friendship->delete() ? 'deleted' : ''
    ]);
}
```

Ejecutamos los tests de phpunit y dusk para ver que pasan.
Y vemos que los cambios realizados se muestran en el navegador.

[Fin video 60]

Vamos a hacer que cuando un usuario publique un estado, comentario o like, éste aparezca en la página de otros usuarios sin necesidad de recargar la página.

Esto va a funcionar en tiempo real mediante eventos que se lanzarán al momento de crear un estado.

Comenzamos en el CreateStatusTest, modificando el siguiente test an_authenticated_user_can_create_statuses:

```
use App\Http\Resources\StatusResource;
use App\Models\Status;
use App\User;
use Tests\TestCase;
use App\Events\StatusCreated;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\RefreshDatabase;

( … )

/** @test */
public function an_authenticated_user_can_create_statuses()
{
    Event::fake([StatusCreated::class]);

    $user = factory(User::class)->create();
    $this->actingAs($user);

    $response = $this->postJson(route('statuses.store'),['body' => 'Mi primer status']);
```

```
    Event::assertDispatched(StatusCreated::class, function ($e){
        return $e->status->id === Status::first()->id
            && get_class($e->status) === StatusResource::class;
    });

    $response->assertJson([
        'data' => ['body' => 'Mi primer status'],
    ]);

    $this->assertDatabaseHas('statuses',[
        'user_id' => $user->id,
        'body' => 'Mi primer status'
    ]);

}
```

Explicación: Verificamos que al crear el estado un evento se ha disparado o despachado mediante el facade Event y el método assertDispatch del evento StatusCreated.
Al que pasamos una función que recibe el evento como parámetro, y comprueba que el id del estado que pasamos es igual al id del único estado que tenemos en la base de datos (solo 1 estado por usar RefreshDatabase). Y añadimos tambien la verificacion que la clase del estado que recibimos sea el StatusResource.

Importamos las clases necesarias y simulamos el disparador de eventos con Event::fake, de manera que Laravel simula el evento StatusCreated de la aplicación.

Ahora en el StatusesController, hacemos que el test anterior pase disparando el evento StatusCreated con el estado que pasamos con el StatusResource.
Y reestructuramos el código para hacerlo más legible:

```
use App\Events\StatusCreated;
use App\Http\Resources\StatusResource;
use App\Models\Status;
use Illuminate\Http\Request;

class StatusesController extends Controller
{
    public function index()
    {
        return StatusResource::collection(
            Status::latest()->paginate()
        );
    }

    public function store(Request $request)
    {
```

```
        $validStatus = $request->validate(['body' => 'required|min:5']);

        $status = $request->user()->statuses()->create($validStatus);

        $statusResource = StatusResource::make($status);
        StatusCreated::dispatch($statusResource);

        return $statusResource;
    }
}
```

Creamos el evento por terminal: php artisan make:event StatusCreated
Y en el StatusCreated (app/events), por el constructor asignamos la propiedad:

```
namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class StatusCreated
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $status;

    /**
     * Create a new event instance.
     *
     * @param $status
     */
    public function __construct($status)
    {
        $this->status = $status;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

Ejecutamos los tests de phpunit y dusk para ver que pasan.

[Fin video 61]

https://laravel.com/docs/5.7/broadcasting#installing-laravel-echo

Una vez verificado que se dispara el evento en Laravel cuando creamos un evento, vamos a comprobar que un usuario puede ver en tiempo real el estado creado por otro usuario.

Comenzamos en UsersCanCreateStatusesTest creando un nuevo test siguiendo como ejemplo el test ya creado:

```
/**
* A Dusk test example.
*
* @test
* @throws \Throwable
*/
public function users_can_see_statuses_in_real_time()
{
  $user1 = factory(\App\User::class)->create();
  $user2 = factory(\App\User::class)->create();

  $this->browse(function (Browser $browser1, Browser $browser2) use ($user1, $user2) {
    $browser1->loginAs($user1)
      ->visit('/');

    $browser2->loginAs($user2)
      ->visit('/')
      ->type('body', 'Mi primer status')
      ->press('#create-status')
      ->screenshot('create-status')
      ->waitForText('Mi primer status')
      ->assertSee('Mi primer status')
      ->assertSee($user2->name);

    $browser1->waitForText('Mi primer status')
      ->assertSee('Mi primer status')
      ->assertSee($user2->name);
  });
}
```

Explicación: Probamos la funcionalidad con 2 usuarios, primero el usuario 1 hace login en una sesión de navegador. Luego el otro user hace login en otra sesión de navegador y crea un estado. Por último comprobamos que el usuario1 ve el estado que ha creado el usuario2 sin recargar la página.

Vamos a instalar Laravel Echo que nos ayudara con la implementación del lado del cliente, junto a la librería pusher para los mensajes en tiempo real, y que se guarde todo en las dependencias de desarrollo:

```
npm install laravel-echo pusher-js --save-dev
```

Al instalar, aparece un error de que hay X numero de vulnerabilidades.
Para solucionarlas ejecutamos: npm audit fix

Ahora ejecutamos el watcher: npm run watch

En mi caso se ejecuta sin problemas, pero al profesor en el video le da problemas para compilar los cambios por problemas con versiones de las dependencias de vuejs y el compilador de vuejs. Los pasos que sigue para solucionarlo son:

- Elimina la carpeta node modules: sudo rm -r node_modules/
- Utiliza yarn en vez de npm. Para instalarlo: npm install -g yarn
- Una vez instalado ejecuta: yarn
- Elimina el archivo package-lock: rm package-lock.json
- Y finalmente para compilar los cambios: yarn watch

Para inicializar los paquetes instalados, vamos al archivo bootstrap.js de resources/js y descomentamos las lineas al final:

```
import Echo from 'laravel-echo'

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    encrypted: true
});
```

Continuamos en el StatusList.vue, y en el metodo mounted abrimos el canal para escuchar el evento disparado por Laravel, el StatusCreated, y usando la propiedad publica status (del StatusCreated) agregue el estado al principio de la lista statuses :

```
mounted() {
  axios.get(this.getUrl)
      .then(res => {
          this.statuses = res.data.data
      })
      .catch(err => {
```

```
            console.log(err.response.data);
        });

    EventBus.$on('status-created', status => {
        this.statuses.unshift(status);
    });

    Echo.channel('statuses').listen('StatusCreated', ({status}) => {
        this.statuses.unshift(status);
    })
},
```

Definimos el canal statuses en el evento StatusCreated de la lección anterior, implementando la interfaz ShouldBroadcast para que el método se ejecute automaticamente al disparar el evento:

```
namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class StatusCreated implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $status;

    /**
     * Create a new event instance.
     *
     * @param $status
     */
    public function __construct($status)
    {
        $this->status = $status;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new Channel('statuses');
    }
}
```

Finalmente comprobamos con un test que el evento StatusCreated es una instancia de ShouldBroadcast. Para ello vamos a utilizar el test de la anterior lección, an_authenticated_user_can_create_statuses del CreateStatusTest:

```
/** @test */
public function an_authenticated_user_can_create_statuses()
{
  Event::fake([StatusCreated::class]);

  $user = factory(User::class)->create();
  $this->actingAs($user);

  $response = $this->postJson(route('statuses.store'),['body' => 'Mi primer status']);

  Event::assertDispatched(StatusCreated::class, function ($e){
    return $e->status->id === Status::first()->id
      && $e->status instanceof StatusResource
      && $e->status->resource instanceof Status
      && $e instanceof ShouldBroadcast;
  });

  $response->assertJson([
    'data' => ['body' => 'Mi primer status'],
  ]);

  $this->assertDatabaseHas('statuses',[
    'user_id' => $user->id,
    'body' => 'Mi primer status'
  ]);

}
```

Explicacion: Añadimos otra verificación para comprobar que el evento es una instancia de la interfaz ShouldBroadcast.
También verificamos que el status->resource es igual al modelo Status.
Y cambiamos el get_class anterior por instance of para una lectura más clara.

Al ejecutar el test de dusk creado anteriormente vemos que falla:

```
php artisan dusk --filter users_can_see_statuses_in_real_time
```

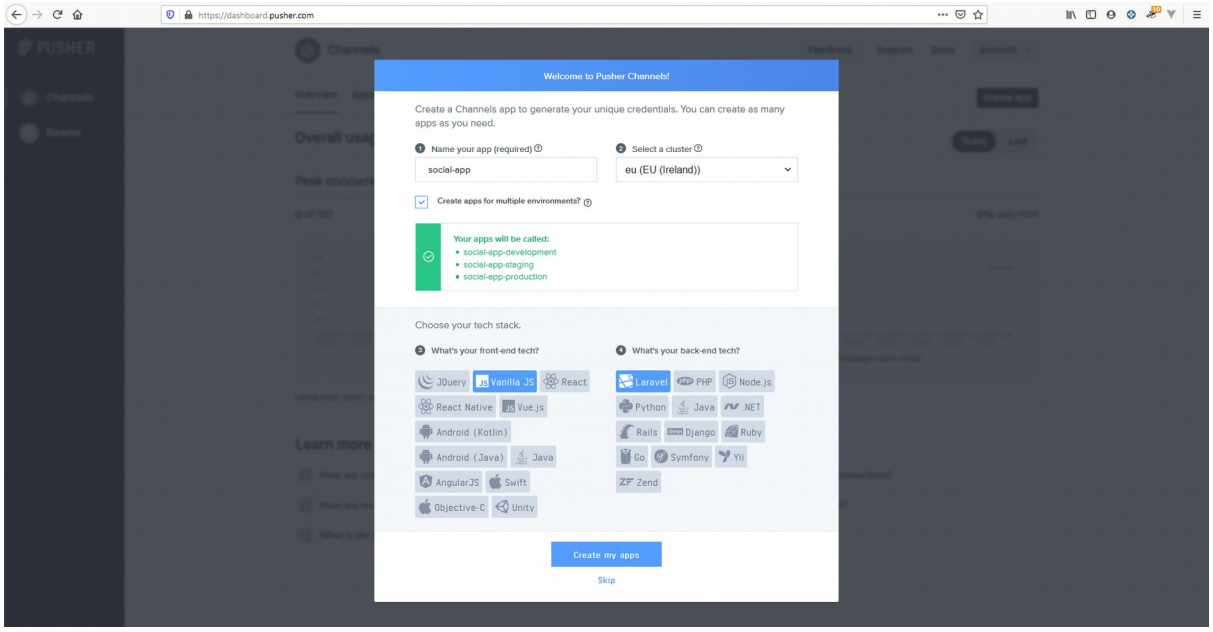
Esto es porque no hemos configurado pusher para funcionar con Laravel.

[Fin video 62]

Vamos a configurar Pusher para que escuche los eventos transmitidos por Laravel cuando se crea un estado y a su vez emita los mensajes a Laravel echo para que actualice la interfaz.

Para ello vamos a https://pusher.com/
Y creamos una cuenta nueva.

Al crear la cuenta, al no tener ninguna app nos aparece una pantalla para crear una. La configuramos de la misma manera que la imagen, con el nombre de aplicación social-app , el servidor en eu, marcando la casilla de crear múltiples app (para desarrollo, pruebas y producción), y marcando como frontend js vanilla y backend Laravel:



Ahora para ver las credenciales de desarrollo, vamos al apartado Channels, seleccionamos el apartado App y dentro de el social-app-development.
Luego vamos al apartado App Keys para ver las claves de la aplicación.

Como estamos utilizando Dusk para probar las características en tiempo real, agregamos las credenciales al archivo .env.dusk.local :

```
BROADCAST_DRIVER=pusher
CACHE_DRIVER=file
SESSION_DRIVER=file
SESSION_LIFETIME=120
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
```

```
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=1011151
PUSHER_APP_KEY=cb9990f0d5c9d2410b74
PUSHER_APP_SECRET=8d56dc5d019ee18c6b35
PUSHER_APP_CLUSTER=eu
```

Y hacemos lo mismo en el archivo .env :

```
BROADCAST_DRIVER=pusher
CACHE_DRIVER=file
SESSION_DRIVER=file
SESSION_LIFETIME=120
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=1011151
PUSHER_APP_KEY=cb9990f0d5c9d2410b74
PUSHER_APP_SECRET=8d56dc5d019ee18c6b35
PUSHER_APP_CLUSTER=eu
```

Seguidamente en la terminal compilamos los assets: npm run dev

Si ejecutamos el test de dusk vemos que sigue fallando:

```
php artisan dusk --filter users_can_see_statuses_in_real_time
```

Para solucionarlo, es necesario instalar la librería Pusher.
En terminal: composer require pusher/pusher-php-server

Comprobamos que el test de dusk ahora pasa, que en la consola de pusher en la página web vemos el mensaje, y que en el navegador si hacemos una prueba manual con 2 navegadores vemos que el estado se muestra en ambas pantallas al crearlo automáticamente.

[Fin video 63]

Vamos a solucionar el problema de la lección anterior de que al publicar un estado, se produce una duplicación de este en el navegador del usuario que lo crea.

Para ello vamos a impedir que el evento se transmita al usuario que crea el estado.

En el StatusCreated (app/events) añadimos al constructor el método dontBroadcastToCurrentUser del trait InteractsWithSockets (añade un header al evento para diferenciar a la persona que crea el evento y excluirla de recibir el broadcast):

```
public function __construct($status)
{
  $this->dontBroadcastToCurrentUser();

  $this->status = $status;
}
```

Ahora es necesario comprobar con un test que se llama al método anterior.
En CreateStatusTest, modificamos el test siguiente quitando la parte de comprobación del evento, que pasamos a un nuevo test:

```
/** @test */
public function an_authenticated_user_can_create_statuses()
{
  Event::fake([StatusCreated::class]);

  $user = factory(User::class)->create();
  $this->actingAs($user);

  $response = $this->postJson(route('statuses.store'),['body' => 'Mi primer status']);

  $response->assertJson([
     'data' => ['body' => 'Mi primer status'],
  ]);

  $this->assertDatabaseHas('statuses',[
     'user_id' => $user->id,
     'body' => 'Mi primer status'
  ]);
}
```

Y añadimos el test para la parte de comprobación del evento:

```
/** @test */
public function an_event_is_fired_when_a_status_is_created()
{
  Event::fake([StatusCreated::class]);
  Broadcast::shouldReceive('socket')->andReturn('socket-id');
```

```
    $user = factory(User::class)->create();

    $this->actingAs($user)->postJson(route('statuses.store'),['body' => 'Mi primer status']);

    Event::assertDispatched(StatusCreated::class, function ($statusCreatedEvent){

        $this->assertInstanceOf(ShouldBroadcast::class, $statusCreatedEvent);
        $this->assertInstanceOf(StatusResource::class, $statusCreatedEvent->status);
        $this->assertInstanceOf(Status::class, $statusCreatedEvent->status->resource);
        $this->assertEquals(Status::first()->id, $statusCreatedEvent->status->id);
        $this->assertEquals(
            'socket-id',
            $statusCreatedEvent->socket,
                        'The event ' . get_class($statusCreatedEvent) . ' must call the method
dontBroadcastToCurrentUser in the constructor.'
        );

        return true;
    });
}
```

Explicación: Usando el facade Broadcast, comprobamos que se llama al método socket. Luego cambiamos las verificaciones del evento anteriores por assertInstanceOf y assertEquals para facilitar la depuración de errores haciéndolos más legibles (la funcionalidad es la misma que antes). Y debemos devolver true.

Finalmente en el archivo phpunit.xml cambiamos el broadcast driver a log para acelerar la ejecución de los tests de phpunit al utilizar el driver log y no pusher:

```
        <env name="DB_CONNECTION" value="sqlite"/>
        <env name="DB_DATABASE" value=":memory:"/>
        <env name="BROADCAST_DRIVER" value="log"/>
    </php>
</phpunit>
```

[Fin video 64]

Vamos a agregar una transición al momento de crear un estado.

Para ello en StatusList.vue, utilizamos el componente de vue.js transition-group y añadimos la configuración para la animación en un style al final de la página:

```
<template>
    <div @click="redirectIfGuest">
        <transition-group name="status-list-transition">
            <status-list-item
```

```
            v-for="status in statuses"
            :status="status"
            :key="status.id"
          ></status-list-item>
      </transition-group>
    </div>
</template>

( … Y al final de la pagina … )

<style>
  .status-list-transition-move{
      transition: all 0.8s;
  }
</style>
```

Al usar transition-group se agrega una clase adicional: move.

Compilamos los cambios realizados: npm run dev
Y vemos que los cambios realizados se muestran en el navegador.

[Fin video 65]

Vamos a hacer que los comentarios aparezcan en tiempo real igual que los estados. Para ello dispararemos un evento cuando se cree un comentario.

En el CreateCommentTest creamos un nuevo test similar al anterior de los estados de la lección anterior para probar que se lanza un evento al crear comentarios e importamos las clases necesarias:

```
use App\Events\CommentCreated;
use App\Http\Resources\CommentResource;
use App\Models\Comment;
use App\User;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Support\Facades\Broadcast;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;
use App\Models\Status;
use Illuminate\Foundation\Testing\RefreshDatabase;

( … Y creamos el test …)

/** @test */
public function an_event_is_fired_when_a_comment_is_created()
{
```

```
    Event::fake([CommentCreated::class]);
    Broadcast::shouldReceive('socket')->andReturn('socket-id');

    $status = factory(Status::class)->create();
    $user = factory(User::class)->create();
    $comment = ['body' => 'Mi primer comentario'];

    $this->actingAs($user)->postJson(route('statuses.comments.store', $status), $comment);

    Event::assertDispatched(CommentCreated::class, function ($commentStatusEvent){

        $this->assertInstanceOf(ShouldBroadcast::class, $commentStatusEvent);
        $this->assertInstanceOf(CommentResource::class, $commentStatusEvent->comment);
        $this->assertInstanceOf(Comment::class, $commentStatusEvent->comment->resource);
        $this->assertEquals(Comment::first()->id, $commentStatusEvent->comment->id);
        $this->assertEquals(
            'socket-id',
            $commentStatusEvent->socket,
                    'The event ' . get_class($commentStatusEvent) . ' must call the method
dontBroadcastToCurrentUser in the constructor.'
        );

        return true;
    });
}
```

Luego creamos el evento CommentCreated por la terminal:

php artisan make:event CommentCreated

Ahora en el StatusCommentController disparamos el evento luego de crear el comentario commentResource:

```
use App\Events\CommentCreated;
use App\Models\Comment;
use App\Models\Status;
use Illuminate\Http\Request;
use App\Http\Resources\CommentResource;

class StatusCommentsController extends Controller
{
    public function store(Status $status)
    {
        request()->validate([
            'body' => 'required'
        ]);

        $comment = Comment::create([
            'user_id' => auth()->id(),
            'status_id' => $status->id,
            'body' => request('body')
        ]);
```

```
        $commentResource = CommentResource::make($comment);

        CommentCreated::dispatch($commentResource);

        return $commentResource;
    }
}
```

Y en el CommentCreated.php (app/events) utilizamos el shouldBroadcast:

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class CommentCreated implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $comment;

    /**
     * Create a new event instance.
     *
     * @param $comment
     */
    public function __construct($comment)
    {
        $this->dontBroadcastToCurrentUser();

        $this->comment = $comment;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

Si ejecutamos ahora el test creado al principio vemos que pasa.

[Fin video 66]

Vamos a crear un test para verificar que el comentario aparece en tiempo real.

En UsersCanCommentStatusTest creamos un nuevo test que usando 2 navegadores compruebe que los comentarios se muestran en tiempo real:

```
/** @test */
public function users_can_see_comments_in_real_time()
{
  $status = factory(Status::class)->create();
  $user = factory(User::class)->create();

  $this->browse(function (Browser $browser1, Browser $browser2) use ($status, $user) {
    $comment = 'Mi primer comentario';

    $browser1->visit('/');

    $browser2->loginAs($user)
      ->visit('/')
      ->waitForText($status->body)
      ->type('comment', $comment)
      ->press('@comment-btn');

    $browser1->waitForText($comment)
      ->assertSee($comment);
  });
}
```

Ahora en el StatusListItem.vue escuchamos con Laravel Echo el evento disparado, creando el método mounted y añadiendo el canal que creará para los comentarios:

```
<script>
  import LikeBtn from './LikeBtn'

  export default {
    props: {
      status: {
        type: Object,
        required: true
      }
    },
    components: { LikeBtn },
    data() {
      return {
        newComment: '',
        comments: this.status.comments
      }
    },
```

```
    mounted(){
                Echo.channel(`statuses.${this.status.id}.comments`).listen('CommentCreated',
({comment}) => {
            this.comments.push(comment);
        });
    },
    methods: {
        addComment() {
            axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
                .then(res => {
                    this.newComment = '';
                    this.comments.push(res.data.data);
                })
                .catch(err => {
                    console.log(err.response.data)
                })
        }
    }
  }
</script>
```

Finalmente en el CommentCreated añadimos el canal:

```
class CommentCreated implements ShouldBroadcast
{
  use Dispatchable, InteractsWithSockets, SerializesModels;

  public $comment;

  /**
   * Create a new event instance.
   *
   * @param $comment
   */
  public function __construct($comment)
  {
      $this->dontBroadcastToCurrentUser();

      $this->comment = $comment;
  }

  /**
   * Get the channels the event should broadcast on.
   *
   * @return \Illuminate\Broadcasting\Channel|array
   */
  public function broadcastOn()
  {
      return new Channel("statuses.{$this->comment->status_id}.comments");
  }
}
```

Si ejecutamos ahora el test creado al principio vemos que pasa.
Y si comprobamos con 2 navegadores vemos que los comentarios aparecen en tiempo real en los navegadores al crearse.

[Fin video 67]

Antes de seguir realizando cambios, vamos a reestructurar el componente StatusListItem.vue por tener demasiadas tareas a cargo.

Comenzamos extrayendo el listado de comentarios a un componente dedicado que vamos a crear. Este nuevo componente de vue lo llamaremos CommentList.

En el CommentList creado, añadimos lo siguiente:

```
<template>
<div>
  <div v-for="comment in comments" class="mb-3">
    <div class="d-flex">
                                <img class="rounded shadow-sm mr-2" height="34px"
width="34px" :src="comment.user.avatar" :alt="comment.user.name">
      <div class="flex-grow-1">
        <div class="card border-0 shadow-sm">
          <div class="card-body p-2 text-secondary">
            <a :href="comment.user.link"><strong>{{ comment.user.name }}</strong></a>
            {{ comment.body }}
          </div>
        </div>

                <small class="badge badge-pill py-1 px-2 mt-1 badge-primary float-right"
dusk="comment-likes-count">
          <i class="fa fa-thumbs-up"></i>
          {{ comment.likes_count }}
        </small>

        <like-btn
          dusk="comment-like-btn"
          :url="`/comments/${comment.id}/likes`"
          :model="comment"
          class="comments-like-btn"
        ></like-btn>
      </div>
    </div>
  </div>
</div>
</template>

<script>
```

```
  import LikeBtn from './LikeBtn';

  export default {
     components: { LikeBtn },
     props: {
        comments: {
           type: Array,
           required: true
        },
        statusId: {
           type: Number,
           required: true
        }
     },
     mounted(){
                  Echo.channel(`statuses.${this.statusId}.comments`).listen('CommentCreated',
({comment}) => {
           this.comments.push(comment);
        });

        EventBus.$on(`statuses.${this.statusId}.comments`, comment => {
           this.comments.push(comment);
        });
     }
  }
</script>

<style scoped>

</style>
```

Y el StatusListItem quedaría tras la reestructuración:

```
<template>
  <div class="card border-0 mb-3 shadow-sm">
     <div class="card-body d-flex flex-column">
        <div class="d-flex align-items-center mb-3">
              <img class="rounded mr-3 shadow-sm" width="40px" :src="status.user.avatar"
alt="status.user.name">
           <div class="">
                                                <h5   class="mb-1"><a   :href="status.user.link"
v-text="status.user.name"></a></h5>
              <div class="small text-muted" v-text="status.ago"></div>
           </div>
        </div>
        <p class="card-text text-secondary" v-text="status.body"></p>
     </div>
     <div class="card-footer pt-2 d-flex justify-content-between align-items-center">

        <like-btn
           dusk="like-btn"
           :url="`/statuses/${status.id}/likes`"
```

```
            :model="status"
        ></like-btn>

        <div class="text-secondary mr-2">
            <i class="far fa-thumbs-up"></i>
            <span dusk="likes-count">{{ status.likes_count }}</span>
        </div>
    </div>

    <div class="card-footer">
        <comment-list
            :comments="status.comments"
            :status-id="status.id"
        ></comment-list>
        <form @submit.prevent="addComment" v-if="isAuthenticated">
            <div class="d-flex align-items-center ">
                                        <img class="rounded shadow-sm float-left mr-2" width="34px" :src="currentUser.avatar" :alt="currentUser.name">
                <div class="input-group">
                        <textarea class="form-control border-0 shadow-sm" name="comment" v-model="newComment" placeholder="Escribe un comentario.." rows="1" required></textarea>

                    <div class="input-group-append">
                        <button class="btn btn-primary" dusk="comment-btn">Enviar</button>
                    </div>
                </div>
            </div>
        </form>

    </div>
  </div>
</template>

<script>
  import LikeBtn from './LikeBtn'
  import CommentList from './CommentList'

  export default {
    props: {
        status: {
            type: Object,
            required: true
        }
    },
    components: { LikeBtn, CommentList },
    data() {
        return {
            newComment: ''
        }
    },
    methods: {
        addComment() {
            axios.post(`/statuses/${this.status.id}/comments`, {body: this.newComment})
```

```
                .then(res => {
                    this.newComment = '';
                    EventBus.$emit(`statuses.${this.status.id}.comments`, res.data.data);
                })
                .catch(err => {
                    console.log(err.response.data)
                })
            }
        }
    }
</script>

<style scoped>

</style>
```

Explicación: Comenzamos extrayendo el listado de comentarios al CommentList. En el CommentList necesitamos el array de comentarios y el status id que añadimos como propiedades.

En el StatusListItem importamos el CommentList y lo utilizamos en el template pasandole las propiedades requeridas.
En el CommentList importamos el LikeBtn y registramos como componente. Nos traemos el método mounted del StatusListItem y lo modificamos para poder usar el status id.

Ahora en el StatusListItem en el método addComment, al no tener ya el array de comentarios en el componente, emitimos un evento que escucharemos en el CommentList con el comentario recién creado.

Finalmente en el CommentList escuchamos el evento creado en el StatusListItem, que recibe por parámetro el comentario creado y lo añadimos al array de comentarios.

Si ejecutamos los tests de Dusk vemos que pasan.

[Fin video 68]

Vamos a extraer el formulario para crear comentarios del StatusListItem.
Para ello creamos un nuevo componente vue llamado CommentForm.

Nos llevamos el formulario del StatusListItem al CommentForm, junto con el método addComment, el newComment, y recibimos la propiedad statusId:

```
<template>
  <form @submit.prevent="addComment" v-if="isAuthenticated" class="mb-3">
    <div class="d-flex align-items-center ">
                                        <img class="rounded shadow-sm float-left mr-2"
width="34px" :src="currentUser.avatar" :alt="currentUser.name">
      <div class="input-group">
                          <textarea class="form-control border-0 shadow-sm" name="comment" v-
model="newComment" placeholder="Escribe un comentario.." rows="1" required></textarea>

        <div class="input-group-append">
          <button class="btn btn-primary" dusk="comment-btn">Enviar</button>
        </div>
      </div>
    </div>
  </form>
  <div v-else class="text-center mb-3">
    Debes <a href="/login">autenticarte</a> para poder comentar
  </div>
</template>

<script>
  export default {
    props: {
      statusId: {
        type: Number,
        required: true
      }
    },
    data() {
      return {
        newComment: ''
      }
    },
    methods: {
      addComment() {
        axios.post(`/statuses/${this.statusId}/comments`, {body: this.newComment})
          .then(res => {
            this.newComment = '';
            EventBus.$emit(`statuses.${this.statusId}.comments`, res.data.data);
          })
          .catch(err => {
            console.log(err.response.data)
          })
      }
    }
  }
</script>
```

En el StatusListItem importamos el componente CommentForm, y lo utilizamos en el template. Y añadimos al footer de comentarios que solo se muestre si se esta autenticado o hay comentarios:

```
<template>
  <div class="card border-0 mb-3 shadow-sm">
    <div class="card-body d-flex flex-column">
      <div class="d-flex align-items-center mb-3">
            <img class="rounded mr-3 shadow-sm" width="40px" :src="status.user.avatar"
alt="status.user.name">
        <div class="">
                                                <h5   class="mb-1"><a   :href="status.user.link"
v-text="status.user.name"></a></h5>
          <div class="small text-muted" v-text="status.ago"></div>
        </div>
      </div>
      <p class="card-text text-secondary" v-text="status.body"></p>
    </div>
    <div class="card-footer pt-2 d-flex justify-content-between align-items-center">
      <like-btn
        dusk="like-btn"
        :url="`/statuses/${status.id}/likes`"
        :model="status"
      ></like-btn>

      <div class="text-secondary mr-2">
        <i class="far fa-thumbs-up"></i>
        <span dusk="likes-count">{{ status.likes_count }}</span>
      </div>
    </div>

    <div class="card-footer pb-0" v-if="isAuthenticated || status.comments.length">
      <comment-list
        :comments="status.comments"
        :status-id="status.id"
      ></comment-list>

      <comment-form
        :status-id="status.id"
      ></comment-form>
    </div>
  </div>
</template>

<script>
  import LikeBtn from './LikeBtn'
  import CommentList from './CommentList'
  import CommentForm from './CommentForm'

  export default {
    props: {
      status: {
        type: Object,
        required: true
      }
    },
```

```
    components: { LikeBtn, CommentList, CommentForm }
  }
</script>
```

Una vez hecha la reestructuración aprovechamos para solucionar el fallo de que si no estamos autenticados, aparece un hueco en blanco donde estaría el formulario para publicar un comentario.
Modificamos el CommentForm como se muestra anteriormente para añadir un div que muestre que debes hacer login para comentar, si no estás autenticado.

Finalmente compilamos los cambios realizados: npm run dev
Y ejecutamos los tests de Dusk: php artisan dusk

IMPORTANTE: Si aparece un error al ejecutar los tests de Dusk de que esperó por el texto y no lo encontró, comprobar la conexión a internet. En mi caso ocurrió por estar sincronizando archivos en la nube. Al quitar la subida de ficheros y volver a ejecutar los tests todos pasaron en verde.

[Fin video 69]

Vamos a reestructurar los tests de eventos CreateStatusTest y CreateCommentsTest para hacerlos más legibles y fáciles de reutilizar.

Comenzamos primero en el CreateStatusTest añadiendo la verificación de que el broadcastOn sea una instancia de Channel (el canal), y que el nombre del canal sea igual a statuses.
Y luego cambiamos las comprobaciones que hacíamos antes por otras que vamos a definir a continuación para facilitar la lectura y encuentro de errores.
La funcionalidad sigue siendo la misma que antes, con la adición de las 2 comprobaciones definidas en este video:

```
/** @test */
public function an_event_is_fired_when_a_status_is_created()
{
  Event::fake([StatusCreated::class]);
  Broadcast::shouldReceive('socket')->andReturn('socket-id');

  $user = factory(User::class)->create();

  $this->actingAs($user)->postJson(route('statuses.store'),['body' => 'Mi primer status']);

  Event::assertDispatched(StatusCreated::class, function ($statusCreatedEvent){
```

```
        $this->assertInstanceOf(StatusResource::class, $statusCreatedEvent->status);
        $this->assertTrue(Status::first()->is($statusCreatedEvent->status->resource));
        $this->assertEventChannelType('public', $statusCreatedEvent);
        $this->assertEventChannelName('statuses', $statusCreatedEvent);
        $this->assertDontBroadcastToCurrentUser($statusCreatedEvent);

        return true;
    });
}
```

Luego en el CreateCommentTest hacemos la misma reestructuracion de código que en el CreateStatusTest, añadiendo el nombre canal correspondiente a comentarios:

```
/** @test */
public function an_event_is_fired_when_a_comment_is_created()
{
    Event::fake([CommentCreated::class]);
    Broadcast::shouldReceive('socket')->andReturn('socket-id');

    $status = factory(Status::class)->create();
    $user = factory(User::class)->create();
    $comment = ['body' => 'Mi primer comentario'];

    $this->actingAs($user)->postJson(route('statuses.comments.store', $status), $comment);

    Event::assertDispatched(CommentCreated::class, function ($commentCreatedEvent){

        $this->assertInstanceOf(CommentResource::class, $commentCreatedEvent->comment);
        $this->assertTrue(Comment::first()->is($commentCreatedEvent->comment->resource));
        $this->assertEventChannelType('public', $commentCreatedEvent);
                $this->assertEventChannelName("statuses.{$commentCreatedEvent->comment-
>status_id}.comments", $commentCreatedEvent);
        $this->assertDontBroadcastToCurrentUser($commentCreatedEvent);

        return true;
    });
}
```

Finalmente en el TestCase.php (tests) creamos los métodos assertDontBroadcastToCurrentUser, assertEventChannelType y assertEventChannelName que hemos usado en los tests anteriores. (Los creamos aquí para que estén disponibles a usar en todos los tests, por eso no los ponemos debajo de los tests anteriores) :

```
namespace Tests;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;
```

```
    protected function assertClassUsesTrait($trait, $class)
    {
        $this->assertArrayHasKey(
            $trait,
            class_uses($class),
            "{$class} must use {$trait} trait"
        );
    }

    protected function assertDontBroadcastToCurrentUser($event, $socketId = 'socket-id')
    {
        $this->assertInstanceOf(ShouldBroadcast::class, $event);

        $this->assertEquals(
            $socketId, // Generated by Broadcast::shouldReceive('socket')->andReturn('socket-id');
            $event->socket,
             'The event ' . get_class($event) . ' must call the method dontBroadcastToCurrentUser
in the constructor.'
        );
    }

    protected function assertEventChannelType($channelType, $event)
    {
        $types = [
            'public' => \Illuminate\Broadcasting\Channel::class,
            'private' => \Illuminate\Broadcasting\PrivateChannel::class,
            'presence' => \Illuminate\Broadcasting\PresenceChannel::class,

        ];

        $this->assertEquals($types[$channelType], get_class($event->broadcastOn()));
    }

    protected function assertEventChannelName($channelName, $event)
    {
        $this->assertEquals($channelName, $event->broadcastOn()->name);
    }
}
```

Explicación: En el assertEventChannelName recibimos el nombre del canal y el evento, y hacemos la verificación que el nombre del broadcastOn del evento sea igual al nombre del canal.
En el assertEventChannelType comprobamos que el tipo de canal sea el que le pasamos por parámetro, en nuestro caso que es de tipo público.
Y en el assertDontBroadcastToCurrentUser comprobamos que se agrega el método dontBroadcastToCurrentUser en el constructor para evitar duplicaciones.

[Fin video 70]

Vamos a comenzar con la implementación de los likes en tiempo real.
Comenzamos con un feature test para comprobar el correcto funcionamiento del evento, y luego al test de dusk para verificar la interfaz.

Primero verificaremos que se dispara un evento cuando se crea un like.

Cambiamos el CommentLikesController para utilizar los métodos like y unlike del trait HasLikes (por no haberlo cambiado en lecciones anteriores):

```
namespace App\Http\Controllers;

use App\Models\Comment;

class CommentLikesController extends Controller
{
  public function store(Comment $comment)
  {
    $comment->like();
  }

  public function destroy(Comment $comment)
  {
    $comment->unlike();
  }
}
```

Para evitar tener que disparar el evento en cada controlador en el que usamos likes, vamos a disparar el evento dentro del método like del Trait para que pueda ejecutarse en cualquier modelo que ejecute este método.

Ahora vamos a añadir al HasLikesTest un nuevo test porque vamos a realizar modificaciones al trait HasLikes:

```
/** @test */
public function an_event_is_fired_when_a_model_is_liked()
{
  Event::fake([ModelLiked::class]);
  Broadcast::shouldReceive('socket')->andReturn('socket-id');

  $this->actingAs(factory(User::class)->create());

  $model = new ModelWithLike(['id' => 1]);

  $model->like();

  Event::assertDispatched(ModelLiked::class, function ($event){
```

```
        $this->assertInstanceOf(ModelWithLike::class, $event->model);
        $this->assertEventChannelType('public', $event);
        $this->assertEventChannelName($event->model->eventChannelName(), $event);
        $this->assertDontBroadcastToCurrentUser($event);

        return true;
    });
}
```

Explicación: Simulamos el evento con Event::fake (importando el facade Event), hacemos login con un usuario creado con factory, usamos un modelo al que dar like y cuando ejecutemos el like utilizamos el facade Event para verificar que se ha disparado el evento.
Y realizamos todas las verificaciones que realizamos con los eventos anteriores en las lecciones previas, modificando lo necesario para que haga referencia al ModelWithLike.

El nombre del modelo dependerá del modelo al que demos like, para testearlo creamos un nuevo test debajo, donde esperaremos el nombre del modelo pero en minúscula seguido del id y .likes:

```
/** @test */
public function can_get_the_event_channel_name()
{
    $model = new ModelWithLike(['id' => 1]);

    $this->assertEquals(
        "modelwithlikes.1.likes",
        $model->eventChannelName()
    );
}
```

Continuamos creando el evento ModelLiked por terminal:

```
php artisan make:event ModelLiked
```

Ahora en el trait HasLikes.php añadimos al método like el evento para despacharlo junto al modelo por $this ($this hace referencia a la clase que utiliza este trait)
Y el método eventChannelName que se encarga de obtener el nombre del canal usando la función class_basename (convirtiendo el modelo en plural) junto al id del modelo y .likes, y al final pasando todo a minúsculas:

```
public function like()
{
    $this->likes()->firstOrCreate([
        'user_id' => auth()->id()
    ]);
```

```
    ModelLiked::dispatch($this);
}

public function eventChannelName()
{
    return strtolower(str_plural(class_basename($this))) . "." . $this->getKey() . ".likes";
}
```

Seguimos en el ModelLiked creado añadiendo la propiedad model como en lecciones anteriores:

```
namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ModelLiked implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $model;

    /**
     * Create a new event instance.
     *
     * @param $model
     */
    public function __construct($model)
    {
        $this->dontBroadcastToCurrentUser();
        $this->model = $model;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new Channel($this->model->eventChannelName());
    }
}
```

Finalmente, en el test HasLikesTests renombramos el modelo ModelWithLikes por ModelWithLike para todas las coincidencias (Alt+J).
Al usar el modelo de prueba no hay una migración en la base de datos para la tabla model_with_likes.
Para solucionarlo añadimos la función setUp al principio de los test donde creamos la tabla, con el id que necesitamos, modificamos los tests para que creen el modelo en la base de datos, y deshabilitamos la columna updated_at en el modelo de prueba:

```
namespace Tests\Unit\Traits;

use App\Events\ModelLiked;
use App\Models\Like;
use App\Traits\HasLikes;
use App\User;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Broadcast;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Schema;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class HasLikesTest extends TestCase
{
  use RefreshDatabase;

  function setUp() //crea la tabla model_with_likes solo para estos tests, luego se borra
  {
      parent::setUp();

      Schema::create('model_with_likes', function ($table){
          $table->increments('id');
      });
  }

  /** @test */
  function a_model_morph_many_likes()
  {
      $model = new ModelWithLike(['id' => 1]);

      factory(Like::class)->create([
          'likeable_id' => $model->id,          // Id del comentario: 1
          'likeable_type' => get_class($model), // Se guardara: App\\Models\\Comment

      ]);

      $this->assertInstanceOf(Like::class, $model->likes->first());
  }

  /** @test */
```

```
    function a_model_can_be_liked_and_unlike()
    {
        $model = ModelWithLike::create();

        $this->actingAs( factory(User::class)->create() );

        $model->like();

        $this->assertEquals(1, $model->likes()->count());

        $model->unlike();

        $this->assertEquals(0, $model->likes()->count());

    }

    /** @test */
    function a_model_can_be_liked_once()
    {
        $model = ModelWithLike::create();

        $this->actingAs( factory(User::class)->create() );

        $model->like();

        $this->assertEquals(1, $model->likes()->count());

        $model->like();

        $this->assertEquals(1, $model->likes()->count());

    }

    /** @test */
    function a_model_knows_if_it_has_been_liked()
    {
        $model = ModelWithLike::create();

        $this->assertFalse($model->isLiked());

        $this->actingAs(factory(User::class)->create());

        $this->assertFalse($model->isLiked());

        $model->like();

        $this->assertTrue($model->isLiked());
    }

    /** @test */
    function a_model_knows_how_many_likes_it_has()
    {
        $model = new ModelWithLike(['id' => 1]);
```

```
        $this->assertEquals(0, $model->likesCount());

        factory(Like::class, 2)->create([
            'likeable_id' => $model->id,          // 1
            'likeable_type' => get_class($model), // App\\Models\\Status

        ]);

        $this->assertEquals(2, $model->likesCount());
    }

    /** @test */
    public function an_event_is_fired_when_a_model_is_liked()
    {
        Event::fake([ModelLiked::class]);
        Broadcast::shouldReceive('socket')->andReturn('socket-id');

        $this->actingAs(factory(User::class)->create());

        $model = new ModelWithLike(['id' => 1]);

        $model->like();

        Event::assertDispatched(ModelLiked::class, function ($event){

            $this->assertInstanceOf(ModelWithLike::class, $event->model);
            $this->assertEventChannelType('public', $event);
            $this->assertEventChannelName($event->model->eventChannelName(), $event);
            $this->assertDontBroadcastToCurrentUser($event);

            return true;
        });
    }

    /** @test */
    public function can_get_the_event_channel_name()
    {
        $model = new ModelWithLike(['id' => 1]);

        $this->assertEquals(
            "modelwithlikes.1.likes",
            $model->eventChannelName()
        );
    }

}

class ModelWithLike extends Model
{
    use HasLikes;

    public $timestamps = false;
```

```
    protected $fillable = ['id'];
}
```

Finalmente comprobamos que los tests de phpunit pasan.

[Fin video 71]

Vamos a crear un test de dusk para comprobar que se actualiza la interfaz en tiempo real al dar un like.

En el UsersCanLikeStatusesTest, añadimos un nuevo test empleando 2 browsers:

```
/**
* @test
* @throws \Throwable
*/
public function users_can_see_likes_in_real_time()
{
    $user = factory(User::class)->create();
    $status = factory(Status::class)->create();

    $this->browse(function (Browser $browser1, Browser $browser2) use ($user, $status){
        $browser1->visit('/');

        $browser2->loginAs($user)
            ->visit('/')
            ->waitForText($status->body)
            ->assertSeeIn('@likes-count', 0)
            ->press('@like-btn')
            ->waitForText('TE GUSTA');

        $browser1->assertSeeIn('@likes-count', 1);
    });
}
```

Y en el StatusListItem, añadimos el método mounted para escuchar el evento con Laravel echo, escuchando el canal statuses + id del estado actual + .likes (nombre del canal por que hacemos el broadcast en el evento ModelLiked) e incrementando el conteo de likes:

```
<script>
    import LikeBtn from './LikeBtn'
    import CommentList from './CommentList'
    import CommentForm from './CommentForm'

    export default {
```

```
        props: {
            status: {
                type: Object,
                required: true
            }
        },
        components: { LikeBtn, CommentList, CommentForm },
        mounted() {
            Echo.channel(`statuses.${this.status.id}.likes`)
                .listen('ModelLiked', e => {
                    this.status.likes_count++;
                });
        }
    }
</script>
```

Al ejecutar el test comprobamos que pasa.

php artisan dusk --filter users_can_see_likes_in_real_time

Y usando 2 navegadores comprobamos que también funciona.

[Fin video 72]

Vamos a implementar la actualización del conteo de likes en tiempo real cuando se quita un like.

Empezamos en el HasLikesTest, creando un nuevo test que verifique que se dispara un evento al quitar un like:

```
/** @test */
public function an_event_is_fired_when_a_model_is_unliked()
{
    Event::fake([ModelUnliked::class]);
    Broadcast::shouldReceive('socket')->andReturn('socket-id');

    $this->actingAs(factory(User::class)->create());

    $model = ModelWithLike::create(); //guardamos modelo en base de datos

    $model->likes()->where([ //asi evitamos disparar el evento ModelLiked
        'user_id' => auth()->id()
    ])->delete();

    $model->unlike();

    Event::assertDispatched(ModelUnliked::class, function ($event){
```

```
        $this->assertInstanceOf(ModelWithLike::class, $event->model);
        $this->assertEventChannelType('public', $event);
        $this->assertEventChannelName($event->model->eventChannelName(), $event);
        $this->assertDontBroadcastToCurrentUser($event);

        return true;
    });
}
```

Creamos por terminal el evento ModelUnliked y lo importamos al test anterior:

php artisan make:event ModelUnliked

Ahora en el trait HasLikes, despachamos el evento en el método unlike e importamos el evento:

```
public function unlike()
{
    $this->likes()->where([
        'user_id' => auth()->id()
    ])->delete();

    ModelUnliked::dispatch($this);
}
```

Continuamos en el ModelUnlike creado, añadiendo la propiedad model, cambiando el canal a público y el nombre al del modelo->eventChannelName, e implementando ShouldBroadcast y el metodo dontBroadcastToCurrentUser:

```
namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ModelUnliked implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $model;

    /**
     * Create a new event instance.
     *
     * @param $model
     */
    public function __construct($model)
```

```
  {
      $this->dontBroadcastToCurrentUser();
      $this->model = $model;
  }

  /**
   * Get the channels the event should broadcast on.
   *
   * @return \Illuminate\Broadcasting\Channel|array
   */
  public function broadcastOn()
  {
      return new Channel($this->model->eventChannelName());
  }
}
```

Ahora realizamos la verificación con laravel dusk en el test users_can_see_likes_and_unlikes_in_real_time del UsersCanLikeStatusesTest que modificamos a lo siguiente para añadir la comprobación al quitar los likes:

```
public function users_can_see_likes_and_unlikes_in_real_time()
{
  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();

  $this->browse(function (Browser $browser1, Browser $browser2) use ($user, $status){
      $browser1->visit('/');

      $browser2->loginAs($user)
          ->visit('/')
          ->waitForText($status->body)
          ->assertSeeIn('@likes-count', 0)
          ->press('@like-btn')
          ->waitForText('TE GUSTA');

      $browser1->assertSeeIn('@likes-count', 1);

      $browser2->press('@like-btn')
          ->waitForText('ME GUSTA');

      $browser1->assertSeeIn('@likes-count', 0);
  });
}
```

Y en el componente StatusListItem, añadimos un nuevo canal para escuchar los eventos ModelUnliked, disminuyendo el conteo de likes:

```
mounted() {
        Echo.channel(`statuses.${this.status.id}.likes`)
            .listen('ModelLiked', e => {
                this.status.likes_count++;
```

```
            });

        Echo.channel(`statuses.${this.status.id}.likes`)
            .listen('ModelUnliked', e => {
                this.status.likes_count--;
            });
    }
  }
</script>
```

Finalmente, al ejecutar el test comprobamos que pasa.

php artisan dusk --filter users_can_see_likes_and_unlikes_in_real_time

Y usando 2 navegadores comprobamos que también funciona.

[Fin video 73]

Vamos a implementar los likes en tiempo real para los comentarios.

En el UsersCanLikeCommentsTest, añadimos otro test probar los likes en tiempo real, similar al test anterior pero usando 2 browsers:

```
/**
* @test
* @throws \Throwable
*/
public function users_can_see_likes_in_real_time()
{
  $user = factory(User::class)->create();
  $comment = factory(Comment::class)->create();

  $this->browse(function (Browser $browser1, Browser $browser2) use ($user, $comment){
    $browser1->visit('/');

    $browser2->loginAs($user)
        ->visit('/')
        ->waitForText($comment->body)
        ->assertSeeIn('@comment-likes-count', 0)
        ->press('@comment-like-btn')
        ->waitForText('TE GUSTA');

    $browser1->assertSeeIn('@comment-likes-count', 1);

    $browser2->press('@comment-like-btn')
        ->waitForText('ME GUSTA');

    $browser1->pause(2000)->assertSeeIn('@comment-likes-count', 0);
```

```
    });
}
```

Vamos a realizar ahora la extraccion de codigo del CommentList a un nuevo componente de vue, que llamaremos CommentListItem.
El CommentListItem quedaría:

```
<template>
  <div class="d-flex">
                                <img class="rounded shadow-sm mr-2" height="34px" width="34px" :src="comment.user.avatar" :alt="comment.user.name">
      <div class="flex-grow-1">
        <div class="card border-0 shadow-sm">
          <div class="card-body p-2 text-secondary">
            <a :href="comment.user.link"><strong>{{ comment.user.name }}</strong></a>
            {{ comment.body }}
          </div>
        </div>

          <small dusk="comment-likes-count" class="badge badge-pill py-1 px-2 mt-1 badge-primary float-right">
            <i class="fa fa-thumbs-up"></i>
            {{ comment.likes_count }}
        </small>

        <like-btn
          dusk="comment-like-btn"
          :url="`/comments/${comment.id}/likes`"
          :model="comment"
          class="comments-like-btn"
        ></like-btn>
      </div>
  </div>
</template>

<script>
  import LikeBtn from './LikeBtn';

  export default {
    components: { LikeBtn },
    props: {
      comment: {
        type: Object,
        required: true
      }
    },
    mounted() {
      Echo.channel(`comments.${this.comment.id}.likes`)
        .listen('ModelLiked', comment => {
          this.comment.likes_count++;
        });
```

```
            Echo.channel(`comments.${this.comment.id}.likes`)
                .listen('ModelUnliked', comment => {
                    this.comment.likes_count--;
                });
        }
    }
</script>
```

Explicación: Cortamos lo que está dentro del v-for del CommentList y lo pegamos en el template. Añadimos el objeto comment como propiedad. Importamos el componente LikeBtn y registramos el componente.
Creamos el método mounted y utilizamos laravel echo para escuchar los eventos de crear like (incrementando el contador de comentarios) y quitar like (disminuyendo el contador).
Siendo el canal: comments + id del comentario + .likes. Y los eventos ModelLiked y ModelUnliked.

Luego en el CommentList, importamos el componente CommentListItem y lo registramos. Y usamos el componente comment-list-item en el template, al que le pasamos por parámetros el objeto comment y una llave única para diferenciar los elementos.
Finalmente quedaría el archivo CommentList así:

```
<template>
<div>
  <comment-list-item
    v-for="comment in comments"
    :comment="comment"
    :key="comment.id"
    class="mb-3"
  ></comment-list-item>
</div>
</template>

<script>
  import CommentListItem from './CommentListItem';

  export default {
    components: { CommentListItem },
    props: {
      comments: {
        type: Array,
        required: true
      },
      statusId: {
        type: Number,
        required: true
      }
```

```
        },
        mounted(){
                        Echo.channel(`statuses.${this.statusId}.comments`).listen('CommentCreated',
({comment}) => {
                this.comments.push(comment);
            });

            EventBus.$on(`statuses.${this.statusId}.comments`, comment => {
                this.comments.push(comment);
            });
        }
    }
</script>
```

IMPORTANTE: Añadir en el test UserCanLikeCommentsTest **->pause(2000)** luego de modificar el conteo de likes al presionar el botón (tanto para dar como para quitar el like). Esto es para que espere un poco antes de verificar el conteo, para que dé tiempo a que se actualice el conteo de likes.

Finalmente, al ejecutar el test comprobamos que pasa.

```
php artisan dusk --filter users_can_see_likes_in_real_time
```

Y usando 2 navegadores comprobamos que también funciona.

[Fin video 74]

Vamos ahora a usar listeners para escuchar cuando se dispare el evento ModelLiked y envíe una notificación al usuario dueño del modelo que recibió el like.

Comenzamos creando un test unitario para este listener:

```
php artisan make:test Listeners/SendNewLikeNotificationTest --unit
```

Abrimos el SendNewLikeNotificationTest y agregamos un test para comprobar que una notificación es enviada cuando un usuario recibe un like:

```
namespace Tests\Unit\Listeners;

use App\Events\ModelLiked;
use App\Models\Status;
use App\Notifications\NewLikeNotification;
use App\User;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;
```

```
class SendNewLikeNotificationTest extends TestCase
{
   use RefreshDatabase;

   /** @test */
   public function a_notification_is_sent_when_a_user_receives_a_new_like()
   {
      Notification::fake();

      $statusOwner = factory(User::class)->create();

      $status = factory(Status::class)->create(['user_id' => $statusOwner->id]);

      ModelLiked::dispatch($status);

      Notification::assertSentTo($statusOwner, NewLikeNotification::class);

   }
}
```

Explicación: Creamos un modelo de estado con factory, guardamos el dueño del estado en una variable, disparamos un evento ModelLiked, y verificamos que se envia la notificacion al dueno del estado que creó el like cuando se dispara el evento (usando facade Notification).
Para poder usar el assertSentTo debemos interceptar las notificaciones (Notification::*fake*()).

Creamos ahora el listener mediante la terminal:
php artisan make:listener SendNewLikeNotification -e App\\Events\\ModelLiked

Y creamos también la notificacion:
php artisan make:notification NewLikeNotification

Ahora en el listener SendNewNotification enviamos la notificación
(el método notify lo obtenemos porque en el modelo User usamos el trait Notifiable, que a su vez usa el trait RoutesNotifications donde está el método notify):

```
class SendNewLikeNotification
{
   /**
    * Create the event listener.
    *
    * @return void
    */
   public function __construct()
   {
      //
   }
```

```
    /**
     * Handle the event.
     *
     * @param  ModelLiked  $event
     * @return void
     */
    public function handle(ModelLiked $event)
    {
        $event->model->user->notify(new NewLikeNotification());
    }
}
```

Continuamos conectando el modelo con el listener, debemos decirle a Laravel que cuando se dispare el evento ModelLiked ejecute este listener.
Para ello, en el archivo EventServiceProvider (app/providers) conectamos el evento ModelLiked con el listener SendNewLikeNotification:

```
class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        'App\Events\ModelLiked' => [
            'App\Listeners\SendNewLikeNotification',
        ],
    ];

( … )
```

Para que pasen los tests, es necesario en el HasLikesTest no probemos el envío de notificaciones al no haber necesidad.
Por ello hacemos que no se ejecuten los listeners añadiéndolos al setUp y quitando el evento modelLiked y modelUnliked de los tests an_event_is_fired_when_a_model_is_liked y an_event_is_fired_when_a_model_is_unliked respectivamente:

```
use App\Events\ModelLiked;
use App\Events\ModelUnliked;
use App\Models\Like;
use App\Traits\HasLikes;
use App\User;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Broadcast;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Schema;
```

```
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class HasLikesTest extends TestCase
{
  use RefreshDatabase;

  function setUp()
  {
    parent::setUp();

    Schema::create('model_with_likes', function ($table){
      $table->increments('id');
    });

    Event::fake([ModelLiked::class, ModelUnliked::class]);
  }

( … Mas adelante …)

public function an_event_is_fired_when_a_model_is_liked()
{
  Broadcast::shouldReceive('socket')->andReturn('socket-id');

  $this->actingAs(factory(User::class)->create());

  $model = new ModelWithLike(['id' => 1]);

  $model->like();

  Event::assertDispatched(ModelLiked::class, function ($event){

    $this->assertInstanceOf(ModelWithLike::class, $event->model);
    $this->assertEventChannelType('public', $event);
    $this->assertEventChannelName($event->model->eventChannelName(), $event);
    $this->assertDontBroadcastToCurrentUser($event);

    return true;
  });
}

/** @test */
public function an_event_is_fired_when_a_model_is_unliked()
{
  Broadcast::shouldReceive('socket')->andReturn('socket-id');

  $this->actingAs(factory(User::class)->create());

  $model = ModelWithLike::create();

  $model->likes()->where([
    'user_id' => auth()->id()
```

```
    ])->delete();

    $model->unlike();

    Event::assertDispatched(ModelUnliked::class, function ($event){

        $this->assertInstanceOf(ModelWithLike::class, $event->model);
        $this->assertEventChannelType('public', $event);
        $this->assertEventChannelName($event->model->eventChannelName(), $event);
        $this->assertDontBroadcastToCurrentUser($event);

        return true;
    });
}
```

Y también es necesario desactivar temporalmente el envío de mails, pues estamos mandando la notificación por email (y esto no está configurado todavía).
En el NewLikeNotification (app/notifications):

```
public function via($notifiable)
{
    return [];
}
```

Finalmente ejecutamos todos los test de phpunit: vendor/phpunit/phpunit/phpunit
Y todos los tests de dusk: php artisan dusk
Comprobamos que todo funciona correctamente.

[Fin video 75]

Vamos a verificar ahora el funcionamiento de la notificación. Tendremos un array de notificaciones en forma de lista en el menú de la pagina, y que cada notificación tenga un link al evento ocurrido junto a un mensaje que diga que al usuario X le gusto tu (estado o comentario).

Para ello, en el SendNewNotificationTest verificamos que se envía el modelo y usuario que dio like a la notificación:

```
/** @test */
public function a_notification_is_sent_when_a_user_receives_a_new_like()
{
    Notification::fake();

    $statusOwner = factory(User::class)->create();
    $likeSender = factory(User::class)->create();
```

```
        $status = factory(Status::class)->create(['user_id' => $statusOwner->id]);

        $status->likes()->create([
            'user_id' => $likeSender->id
        ]);

        ModelLiked::dispatch($status, $likeSender);

        Notification::assertSentTo(
            $statusOwner,
            NewLikeNotification::class,
            function($notification, $channels) use ($likeSender, $status){
                $this->assertTrue($notification->model->is($status));
                $this->assertTrue($notification->likeSender->is($likeSender));
                return true;
            });

    }
```

Explicación: En la verificación del envío de la notificación pasamos como parámetro una función con la notificación y los canales. Dentro, comprobamos que la propiedad model de la notificación sea el estado $status, y que la notificación recibe el usuario que dio el like (likeSender).
Creando para ello en el mismo test, un like donde el user_id sea el de otro usuario que creamos llamado likeSender.

Ahora en NewLikeNotification agregamos la propiedad model y likeSender a la notificación:

```
namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class NewLikeNotification extends Notification
{
    use Queueable;

    public $model;
    public $likeSender;

    /**
     * Create a new notification instance.
     *
     * @param $model
     * @param $likeSender
     */
    public function __construct($model, $likeSender)
```

```
    {
        //
        $this->model = $model;
        $this->likeSender = $likeSender;
    }

    /**
     * Get the notification's delivery channels.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function via($notifiable)
    {
        return [];
    }

    /**
     * Get the mail representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return \Illuminate\Notifications\Messages\MailMessage
     */
    public function toMail($notifiable)
    {
        return (new MailMessage)
                    ->line('The introduction to the notification.')
                    ->action('Notification Action', url('/'))
                    ->line('Thank you for using our application!');
    }

    /**
     * Get the array representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function toArray($notifiable)
    {
        return [
            //
        ];
    }
}
```

Continuamos en SendNewLikeNotification agregando parámetros a la notificación:

```
public function handle(ModelLiked $event)
{
        $event->model->user->notify(new NewLikeNotification($event->model, $event->likeSender));
}
```

Ahora es necesario añadir la propiedad likeSender al evento.
Por lo que primero verificamos en el HasLikesTest que event->likeSender es el mismo usuario que envía el like:

```
/** @test */
public function an_event_is_fired_when_a_model_is_liked()
{
  Broadcast::shouldReceive('socket')->andReturn('socket-id');

  $this->actingAs($likeSender = factory(User::class)->create());

  $model = new ModelWithLike(['id' => 1]);

  $model->like();

  Event::assertDispatched(ModelLiked::class, function ($event) use ($likeSender){
    $this->assertInstanceOf(ModelWithLike::class, $event->model);
    $this->assertTrue($event->likeSender->is($likeSender));
    $this->assertEventChannelType('public', $event);
    $this->assertEventChannelName($event->model->eventChannelName(), $event);
    $this->assertDontBroadcastToCurrentUser($event);

    return true;
  });
}
```

Ahora en el evento ModelLiked, agregamos el likeSender:

```
class ModelLiked implements ShouldBroadcast
{
  use Dispatchable, InteractsWithSockets, SerializesModels;

  public $model;
  public $likeSender;

  /**
   * Create a new event instance.
   *
   * @param $model
   * @param $likeSender
   */
  public function __construct($model, $likeSender)
  {
    $this->dontBroadcastToCurrentUser();
    $this->model = $model;
    $this->likeSender = $likeSender;
  }

  /**
   * Get the channels the event should broadcast on.
```

```
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new Channel($this->model->eventChannelName());
    }
}
```

Y finalmente en el trait HasLikes pasamos como parámetro el usuario actualmente autenticado (el que crea el like), cuando creamos el evento:

```
public function like()
{
    $this->likes()->firstOrCreate([
        'user_id' => auth()->id()
    ]);

    ModelLiked::dispatch($this, auth()->user());
}
```

[Fin video 76]

Vamos a verificar que los datos de la notificación se guardan correctamente en la base de datos.
Primero verificamos que la notificación usa el canal ‘database’.

En SendNewLikeNotificationTest, comprobamos que el canal database se encuentre dentro de los canales de la notificación:

```
public function a_notification_is_sent_when_a_user_receives_a_new_like()
{
    Notification::fake();

    $statusOwner = factory(User::class)->create();
    $likeSender = factory(User::class)->create();

    $status = factory(Status::class)->create(['user_id' => $statusOwner->id]);

    $status->likes()->create([
        'user_id' => $likeSender->id
    ]);

    ModelLiked::dispatch($status, $likeSender);

    Notification::assertSentTo(
```

```
        $statusOwner,
        NewLikeNotification::class,
        function($notification, $channels) use ($likeSender, $status){
            $this->assertContains('database', $channels);
            $this->assertTrue($notification->model->is($status));
            $this->assertTrue($notification->likeSender->is($likeSender));
            return true;
        });

    }
```

Para verificar que la notificación se guarda en la base de datos, creamos un nuevo test unitario en la terminal dentro de la carpeta Notifications:

php artisan make:test Notifications/NewLikeNotificationTest --unit

Ahora en el NewLikeNotificationTest (tests/unit/notifications) añadimos el nuevo test:

```
namespace Tests\Unit\Notifications;

use App\Models\Status;
use App\Notifications\NewLikeNotification;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class NewLikeNotificationTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function the_notification_is_stored_in_the_database()
  {
      $statusOwner = factory(User::class)->create();
      $likeSender = factory(User::class)->create();

      $status = factory(Status::class)->create(['user_id' => $statusOwner->id]);

      $status->likes()->create(['user_id' => $likeSender->id]);

      $statusOwner->notify(new NewLikeNotification($status, $likeSender));

      $this->assertCount(1, $statusOwner->notifications);

      $notificationsData = $statusOwner->notifications->first()->data;

      $this->assertEquals($status->path(), $notificationsData['link']);
          $this->assertEquals("Al usuario {$likeSender->name} le gustó tu publicación.",
$notificationsData['message']);
  }
}
```

Explicación: Usamos un usuario que crea el estado, otro usuario que da like al estado, el estado, y el like.
Luego usamos al dueño del estado para enviarle la notificación, y revisamos que se ha guardado en la base de datos.
Y verificamos que recibimos el link del modelo, obteniendo la ruta a través del estado con una función path que definiremos más adelante, y un texto específico en el campo mensaje de la notificación.

Necesitamos crear ahora la tabla notifications pues no existe, para ello ejecutamos en la terminal: php artisan notifications:table

Seguidamente, en el modelo Status creamos el método path para usarlo en los tests anteriores, donde devolvemos la ruta statuses.show con el estado como parametro:

```
public function path()
{
  return route('statuses.show', $this);
}
```

Ahora hace falta crear la ruta statuses.show en el archivo de rutas web.php:

```
// Statuses routes
Route::get('statuses', 'StatusesController@index')->name('statuses.index');
Route::get('statuses/{status}', 'StatusesController@show')->name('statuses.show');
Route::post('statuses', 'StatusesController@store')->name('statuses.store')->middleware('auth');
```

Luego en el NewLikeNotification es necesario añadir el canal database, y la llave link (con el path del modelo) y message (con el texto definido en el test junto al nombre del usuario que envía la notificación) dentro del array de la notificación:

```
class NewLikeNotification extends Notification
{
  use Queueable;

  public $model;
  public $likeSender;

  /**
   * Create a new notification instance.
   *
   * @param $model
   * @param $likeSender
   */
  public function __construct($model, $likeSender)
  {
    //
    $this->model = $model;
```

```
        $this->likeSender = $likeSender;
    }

    /**
     * Get the notification's delivery channels.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function via($notifiable)
    {
        return ['database'];
    }

    /**
     * Get the mail representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return \Illuminate\Notifications\Messages\MailMessage
     */
    public function toMail($notifiable)
    {
        return (new MailMessage)
                    ->line('The introduction to the notification.')
                    ->action('Notification Action', url('/'))
                    ->line('Thank you for using our application!');
    }

    /**
     * Get the array representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function toArray($notifiable)
    {
        return [
            'link' => $this->model->path(),
            'message' => "Al usuario {$this->likeSender->name} le gustó tu publicación."
        ];
    }
}
```

Si ejecutamos el test the_notification_is_stored_in_the_database del NewLikeNotificationTest vemos que ahora pasa.

Vamos a implementar ahora la ruta para ver un estado individual.
En el ListStatusesTest, creamos un nuevo test con un estado creado y hacemos un path comprobando que vemos el cuerpo del estado:

```
/** @test */
```

```
public function can_see_individual_status()
{
   $status = factory(Status::class)->create();

   $this->get($status->path())->assertSee($status->body);
}
```

En el StatusesController, creamos el método show que hemos definido el la ruta anteriormente. Recibiendo como parámetro el estado y retornando la vista statuses.show con el estado transformado por StatusResource como parámetro:

```
public function show(Status $status)
{
   return view('statuses.show', [
       'status' => StatusResource::make($status)
   ]);
}
```

Creamos la vista statuses.show, para ello en la carpeta resources/views creamos una nueva carpeta llamada statuses, y dentro el archivo show.blade.php en el que imprimimos el cuerpo del estado:

```
{{ $status->body }}
```

Finalmente al ejecutar todos los tests de phpunit, falla el test an_authenticated_user_can_like_and_unlike_comments del CanLikeCommentsTest porque no hemos definido el método path() para el modelo comment, solo para el status.

Como no nos interesa que en este test se envie la notificacion, capturamos las notificaciones para que no se envíen:

```
function an_authenticated_user_can_like_and_unlike_comments()
{
   \Notification::fake();

    // Para que phpunit ignore el test al ejecutarse: $this->markTestIncomplete();
    $this->withoutExceptionHandling();

( … )
```

Finalmente ejecutamos todos los test de phpunit: vendor/phpunit/phpunit/phpunit
Todos los tests de dusk: php artisan dusk
Y comprobamos en el navegador que todo funciona correctamente.

[Fin video 77]

Vamos a implementar una url para obtener las notificaciones del usuario actualmente autenticado.

Creamos un nuevo test por la terminal:

```
php artisan make:test CanGetNotificationsTest
```

En el CanGetNotificationsTest creamos un test para comprobar qué usuarios autenticados obtienen sus notificaciones (usando el modelo DatabaseNotification que viene con Laravel).
Y otro para que los invitados no tienen acceso (para comprobar el middleware auth de la ruta):

```
namespace Tests\Feature;

use App\User;
use Illuminate\Notifications\DatabaseNotification;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanGetNotificationsTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function guests_users_cannot_access_notifications()
  {
      $this->getJson(route('notifications.index'))->assertStatus(401);
  }

  /** @test */
  public function authenticated_users_can_get_their_notifications()
  {
      $user = factory(User::class)->create();

      $notification = factory(DatabaseNotification::class)->create(['notifiable_id' => $user->id]);

      $this->actingAs($user)->getJson(route('notifications.index'))
          ->assertJson([
              [
                  'data' => [
                      'link' => $notification->data['link'],
                      'message' => $notification->data['message'],
                  ]
              ]
          ]);
  }
}
```

Creamos el factory para el modelo DatabaseNotification:

php artisan make:factory DatabaseNotificationFactory

Abrimos el DatabaseNotificationFactory y usando los campos de la tabla de migración de las notificaciones, añadimos los campos necesarios:

```
use App\User;
use Faker\Generator as Faker;
use Illuminate\Notifications\DatabaseNotification;
use Illuminate\Support\Str;

$factory->define(DatabaseNotification::class, function (Faker $faker) {
  return [
     'id' => Str::uuid()->toString(), //genera un identificador unico
     'type' => 'App\\Notifications\\ExampleNotification',
     'notifiable_type' => 'App\\User',
     'notifiable_id' => factory(User::class)->create(),
     'data' => [
         'link' => url('/'),
         'message' => 'Mensaje de la notificacion'
     ],
     'read_at' => null,
  ];
});
```

Ahora es necesario crear la ruta notifications.index puesta en el test.
Para ello en el archivo de rutas web.php añadimos:

```
// Notification routes
Route::get('notifications', 'NotificationsController@index')->name('notifications.index')->middleware('auth');
```

Creamos el controlador que hemos definido en la ruta:
php artisan make:controller NotificationsController

Y añadimos el método index en el NotificationsController para retornar las notificaciones del usuario actualmente autenticado:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class NotificationsController extends Controller
{
  public function index()
  {
     return auth()->user()->notifications;
  }
}
```

Finalmente ejecutamos todos los test de phpunit: vendor/phpunit/phpunit/phpunit
Y comprobamos que los tests pasan.

[Fin video 78]

Vamos a implementar un botón en el navbar para mostrar el listado de notificaciones que tenemos.

Creamos primero un test de dusk:

```
php artisan dusk:make UsersCanGetTheirNotificationsTest
```

Y en el UsersCanGetTheirNotificationsTest creamos un test para comprobar que los usuarios pueden ver sus notificaciones en el navbar:

```
use App\Models\Status;
use App\User;
use Illuminate\Notifications\DatabaseNotification;
use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class UsersCanGetTheirNotificationsTest extends DuskTestCase
{
  use DatabaseMigrations;
  /**
   * @test
   * @throws \Throwable
   */
  public function user_can_see_their_notifications_in_the_nav_bar()
  {
     $user = factory(User::class)->create();
     $status = factory(Status::class)->create();

     $notification = factory(DatabaseNotification::class)->create([
        'notifiable_id' => $user->id,
        'data' => [
           'message' => 'Has recibido un like',
           'link' => route('statuses.show', $status)
        ]
     ]);

     $this->browse(function (Browser $browser) use ($user, $status, $notification) {
        $browser->loginAs($user)
           ->visit('/')
           ->resize(1024, 768) //para que no se colapse el navbar
           ->click('@notifications')
           ->assertSee('Has recibido un like')
```

```
            ->click("@{$notification->id}")
            ->assertUrlIs($status->path())
        ;
    });
  }
}
```

Explicación: Necesitamos un usuario, un estado, y una notificación con el id del usuario creado y con un mensaje concreto y un link que lleve al estado.

Luego comprobamos que al hacer un login con el usuario, visitar el home y hacer click en un elemento notifications, vemos el mensaje en el navbar.
Y también que hacer click en la notificación (con un identificador único de dusk) nos redirige a la url del link del estado.

Vamos ahora a extraer la barra de navegación a un partial. Para ello, en el layouts app.blade.php quitamos el navbar y lo reemplazamos por un include.
Además añadimos el include junto al main en un div con el elemento app para que vue pueda ejecutarse también en el partial nav y no solo en el main como antes:

```
( … )
</head>
<body>

<div id="app">
  @include('partials.nav')

  <main class="py-4">
      @yield('content')
  </main>
</div>

<script src="{{ mix('js/app.js') }}"></script>
</body>
</html>
```

Creamos ahora el partial nav en la carpeta partials de resources/views, el archivo lo llamaremos nav.blade.php
Y pegamos navbar, y añadimos el elemento notification-list para mostrar el dropdown donde mostraremos las notificaciones en el navbar:

```
<nav class="navbar navbar-expand-lg navbar-light navbar-socialapp">
  <div class="container">

        <a class="navbar-brand" href="{{ route('home') }}"><i class="fa fa-address-book text-primary mr-1"></i> SocialApp</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
```

```
            <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="navbarSupportedContent">
            <ul class="navbar-nav mx-auto">
                {{--            <li class="nav-item active">--}}
                {{--                <a class="nav-link" href="#">Inicio <span class="sr-only">(current)</
span></a>--}}
                {{--            </li>--}}
                {{--            <li class="nav-item">--}}
                {{--                <a class="nav-link" href="#">Link</a>--}}
                {{--            </li>--}}
            </ul>

            <ul class="navbar-nav ml-auto">
                @guest()
                                        <li class="nav-item"><a href="{{ route('register') }}"
class="nav-link">Register</a></li>
                    <li class="nav-item"><a href="{{ route('login') }}" class="nav-link">Login</a></li>
                @else
                    <notification-list><i class="fa fa-bell"></i></notification-list>

                    <li class="nav-item dropdown">
                                <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                            {{ Auth::user()->name }}
                        </a>
                        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
                                                <a class="dropdown-item" href="{{ route('users.show',
Auth::user()) }}">Perfil</a>
                            <div class="dropdown-divider"></div>
                            <a onclick=" document.getElementById('logout').submit()" class="dropdown-
item" href="#">Cerrar sesión</a>
                        </div>
                    </li>

                    <form id="logout" action="{{ route('logout') }}" method="POST">@csrf</form>
                @endguest
            </ul>

        </div>
    </div>
</nav>
```

Ahora creamos el componente de vue NotificationList:

```
<template>
    <li class="nav-item dropdown">
        <a dusk="notifications"
            href="#"
            class="nav-link dropdown-toggle"
            id="dropdownNotifications"
```

```
        role="button"
        data-toggle="dropdown"
        aria-haspopup="true"
        aria-expanded="false">
         <slot></slot>
      </a>
                                    <div class="dropdown-menu dropdown-menu-right" aria-
labelledby="dropdownNotifications">
        <a v-for="notification in notifications"
          :dusk="notification.id"
          :href="notification.data.link"
          class="dropdown-item"
        >{{ notification.data.message}}</a>
      </div>
   </li>
</template>

<script>
   export default {
      data(){
         return {
            notifications: []
         }
      },
      created() {
         axios.get('/notifications')
            .then(res => {
               this.notifications = res.data;
            })
      }
   }
</script>
```

Explicación: Creamos el dropdown de las notificaciones (que es similar al dropdown del partial nav) y hacemos un v-for para mostrar todas las notificaciones del usuario.
En el componente de notificaciones, cuando se cree, hacemos un llamado a la url de las notificaciones, obtenemos las notificaciones y las guardamos en una variable que luego usamos en el v-for para mostrar el contenido de la notificación.

Usamos <slot> para poder cambiar el nombre que aparece en el desplegable del navbar en el partial nav, añadiendo un icono. Y añadimos al enlace del dropdown el bind al id de la notificación, y el bind al enlace de la notificación.

Finalmente importamos el nuevo componente de vue al app.js de resources/js:

```
Vue.component('friendship-btn', require('./components/FriendshipBtn'));
Vue.component('accept-friendship-btn', require('./components/AcceptFriendshipBtn'));
Vue.component('notification-list', require('./components/NotificationList'));
```

Compilamos los assets: npm run dev
Ejecutamos el test creado en esta leccion:

```
php artisan dusk --filter user_can_see_their_notifications_in_the_nav_bar
```

Y comprobamos que pasa.

[Fin video 79]

Vamos a agregar estilo a las notificaciones, y luego poder marcar las notificaciones como leídas y no leídas.

En el archivo show.blade.php de resources/views/statuses, agregamos el estilo:

```
@extends('layouts.app')

@section('content')

<div class="container">
  <status-list-item
     :status="{{ json_encode($status) }}"
  ></status-list-item>
</div>
@endsection
```

Pero no hemos registrado el componente de vue StatusListItem para poder usarlo de forma global. Por ello lo agregamos en el app.js de resources/js:

```
Vue.component('status-list', require('./components/StatusList'));
Vue.component('status-list-item', require('./components/StatusListItem'));
Vue.component('friendship-btn', require('./components/FriendshipBtn'));
```

Y ahora podemos quitar el import del componente anterior en el StatusList.vue:

```
<script>
( aqui estaba el import del StatusListItem )
  export default {
     props: {
        url: String
     },
     data() {
```

Continuaremos creando unos tests para marcar las notificaciones como leídas y no leídas, así como comprobar que un invitado no puede marcar notificaciones.

Empezamos con creando un nuevo test en el CanGetNotificationsTest, pero antes le cambiamos el nombre a CanManageNotificationsTest (shift+F6 en el nombre del test):

```
/** @test */
public function guests_users_cannot_mark_notifications()
{
  $notification = factory(DatabaseNotification::class)->create();

  $this->postJson(route('read-notifications.store', $notification))->assertStatus(401);
  $this->deleteJson(route('read-notifications.destroy', $notification))->assertStatus(401);
}

/** @test */
public function authenticated_users_can_mark_notifications_as_read()
{
  $user = factory(User::class)->create();

  $notification = factory(DatabaseNotification::class)->create([
     'notifiable_id' => $user->id,
     'read_at' => null
     ]);

          $response    =       $this->actingAs($user)->postJson(route('read-notifications.store',
$notification));

  $response->assertJson($notification->fresh()->toArray());

  $this->assertNotNull($notification->fresh()->read_at);
}

/** @test */
public function authenticated_users_can_mark_notifications_as_unread()
{
  $user = factory(User::class)->create();

  $notification = factory(DatabaseNotification::class)->create([
     'notifiable_id' => $user->id,
     'read_at' => now()
  ]);

        $response    =      $this->actingAs($user)->deleteJson(route('read-notifications.destroy',
$notification));

  $response->assertJson($notification->fresh()->toArray());

  $this->assertNull($notification->fresh()->read_at);
}
```

Explicación: Creamos primero el test para probar que usuarios autenticados pueden marcar las notificaciones como leídas.

Necesitamos un usuario y notificación, luego actuando como el usuario hacemos un postJson a la nueva ruta read-notification con metodo store junto a la notificación. Verificamos que el campo read_at no sea nulo, y que recibimos la notificación

transformada en un array.

Luego nos basamos en el test anterior para crear otro para notificaciones marcadas como no leídas. Haciendo un deleteJson a la misma url anterior pero al método destroy, verificando que recibimos la notificación en forma de array, y finalmente que el campo read_at es nulo.

Y finalmente creamos un test para comprobar que los invitados no puedan marcar las notificaciones. Comprobando que al hacer un postJson a las rutas anteriores recibimos un código de error 401.

Creamos las nuevas rutas en el archivo de rutas web.php, en una nueva acepción:

```
// Notification routes
Route::get('notifications', 'NotificationsController@index')->name('notifications.index')->middleware('auth');

// Read Notification routes
Route::post('read-notifications/{notification}', 'ReadNotificationsController@store')->name('read-notifications.store')->middleware('auth');
Route::delete('read-notifications/{notification}', 'ReadNotificationsController@destroy')->name('read-notifications.destroy')->middleware('auth');
```

Creamos el nuevo controlador ReadNotificationsController:

```
php artisan make:controller ReadNotificationsController
```

Y le añadimos el metodo store y destroy al controlador creado, donde recibimos la notificación, la marcamos como leída o como no leida dependiendo el método, y la retornamos:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Notifications\DatabaseNotification;

class ReadNotificationsController extends Controller
{
  public function store(DatabaseNotification $notification)
  {
      $notification->markAsRead();

      return $notification;
  }

  public function destroy(DatabaseNotification $notification)
  {
      $notification->markAsUnread();
```

```
        return $notification;
    }
}
```

Compilamos los assets: npm run dev
Ejecutamos el test creado en esta lección y todos los de phpunit
Y comprobamos que pasan.

[Fin video 80]

Vamos a agregar un botón al lado de las notificaciones para poder marcarlas como leídas o como no leídas.

Comenzamos en el UsersCanGetTheirNotificationsTest, añadiendo la comprobacion de que si presionamos un botón mark-as-read-{id notificación} la notificación pasa a ser leída, y si presionamos otro botón mark-as-unread lo contrario:

```
public function user_can_see_their_notifications_in_the_nav_bar()
{
    $user = factory(User::class)->create();
    $status = factory(Status::class)->create();

    $notification = factory(DatabaseNotification::class)->create([
        'notifiable_id' => $user->id,
        'data' => [
            'message' => 'Has recibido un like',
            'link' => route('statuses.show', $status)
        ]
    ]);

    $this->browse(function (Browser $browser) use ($user, $notification, $status) {
        $browser->loginAs($user)
            ->visit('/')
            ->resize(1024, 768)
            ->click('@notifications')
            ->assertSee('Has recibido un like')
            ->click("@{$notification->id}")
            ->assertUrlIs($status->path())

            ->click('@notifications')
            ->pause(1000) // para que de tiempo a que carge el boton
            ->press("@mark-as-read-{$notification->id}")
            ->waitFor("@mark-as-unread-{$notification->id}")
            ->assertMissing("@mark-as-read-{$notification->id}")

            ->press("@mark-as-unread-{$notification->id}")
```

```
            ->waitFor("@mark-as-read-{$notification->id}")
            ->assertMissing("@mark-as-unread-{$notification->id}")
        ;
    });
}
```

Continuamos en el NotificationList, llevando la parte del dropdown de las notificaciones a un nuevo componente y haciendo el v-for en este elemento junto a una llave para la notificación y otra para el id de ésta:

```
                                <div class="dropdown-menu dropdown-menu-right" aria-labelledby="dropdownNotifications">
            <notification-list-item
                v-for="notification in notifications"
                :notification="notification"
                :key="notification.id"
            ></notification-list-item>
        </div>
    </li>
</template>

<script>
    import NotificationListItem from "./NotificationListItem";

    export default {
        components: { NotificationListItem },
        data(){
            return {
                notifications: []
            }
        },
        created() {
            axios.get('/notifications')
                .then(res => {
                    this.notifications = res.data;
                })
        }
    }
</script>
```

Y creamos el nuevo componente de vue NotificationListItem en resources/js/components, donde añadimos la lógica para el dropdown:

```
<template>
    <div>
        <a :dusk="notification.id"
           :href="notification.data.link"
           class="dropdown-item"
        >{{ notification.data.message}}</a>
                    <button v-if="isRead" @click.stop="markAsUnread" :dusk="`mark-as-unread-${notification.id}`">Marcar como No leida</button>
```

```
                    <button v-else @click.stop="markAsRead" :dusk="`mark-as-read-${notification.id}`">Marcar como leida</button>
  </div>
</template>

<script>
  export default {
      props: {
          notification: Object
      },
      data() {
          return {
              isRead: !! this.notification.read_at
          }
      },
      methods: {
          markAsRead(){
              axios.post(`/read-notifications/${this.notification.id}`)
                  .then(res => {
                      this.isRead = true;
                  })
          },
          markAsUnread(){
              axios.delete(`/read-notifications/${this.notification.id}`)
                  .then(res => {
                      this.isRead = false;
                  })
          }
      }
  }
</script>
```

Explicación: Creamos 2 botones, uno para marcar las notificaciones y otro para desmarcarlas. Estos botones contendrán un selector de dusk con el id de la notificación, los mostramos con un v-if dependiendo de si la notificación se ha leído o no. Y cuando hacemos click que efectivamente se marque o no como leída (usamos stop para prevenir que el dropdown se cierre al hacer click al botón).

En el script recibimos la notificación como propiedad, creamos un valor isRead que será un booleano dependiendo el campo read_at de la notificación (si read_at es null sera falso al no haberse leido).

Finalmente creamos los métodos para marcar la notificación como leída o como no leída, haciendo un post a la url de las notificaciones y poniendo la variable isRead como corresponda.

Compilamos los assets: npm run dev
Ejecutamos el test usado en esta lección:

php artisan dusk --filter user_can_see_their_notifications_in_the_nav_bar

Y comprobamos que pasan.

[Fin video 81]

Vamos a mejorar la apariencia del dropdown de notificaciones añadiendo estilo con bootstrap.

Empezamos en el NotificationListItem:

```
<template>
  <div class="dropdown-item d-flex align-items-center"
       :class="isRead ? '' : 'bg-light'"
  >
      <a :dusk="notification.id"
        :href="notification.data.link"
        class="dropdown-item"
      >{{ notification.data.message}}</a>
                                    <button v-if="isRead" class="btn btn-link mr-2"
@click.stop="markAsUnread" :dusk="`mark-as-unread-${notification.id}`">
          <i class="far fa-circle"></i>
            <span class="position-absolute bg-dark text-white ml-2 py-1 px-2 rounded">Marcar
como NO leída</span>
      </button>
          <button v-else class="btn btn-link mr-2" @click.stop="markAsRead" :dusk="`mark-as-
read-${notification.id}`">
          <i class="fas fa-circle"></i>
            <span class="position-absolute bg-dark text-white ml-2 py-1 px-2 rounded">Marcar
como leída</span>
      </button>
  </div>
</template>

<script>
  export default {
      props: {
          notification: Object
      },
      data() {
          return {
              isRead: !! this.notification.read_at
          }
      },
      methods: {
          markAsRead(){
              axios.post(`/read-notifications/${this.notification.id}`)
                  .then(res => {
                      this.isRead = true;
```

```
                    EventBus.$emit('notification-was-read');
                })
            },
            markAsUnread(){
                axios.delete(`/read-notifications/${this.notification.id}`)
                    .then(res => {
                        this.isRead = false;
                        EventBus.$emit('notification-was-unread');
                    })
            }
        }
    }
</script>

<style lang="scss" scoped>
    button > span {
        display: none;
    }

    button i {
        &:hover {
            & + span {
                display: inline;
            }
        }
    }
</style>
```

Explicación: Usamos iconos de font-awesome para mostrar si está leída o no leída la notificación. Creamos con mensaje que salga sobre el icono de la notificación explicando que hace el botón, con css al final de página en el style.

Añadimos clases de bootstrap para dar el formato que queremos.
Y mostramos el número de notificaciones al lado del icono de la campana, haciendo que el icono se actualice en tiempo real al hacer click en los iconos para marcar como leída o no las notificaciones.
Finalmente disparamos un evento cuando se actualiza el estado de la notificación.

Y en el NotificationList:

```
<template>
    <li class="nav-item dropdown">
        <a dusk="notifications"
            href="#"
            class="nav-link dropdown-toggle"
            :class="count ? 'text-primary font-weight-bold' : ''"
            id="dropdownNotifications"
            role="button"
            data-toggle="dropdown"
```

```
        aria-haspopup="true"
        aria-expanded="false">
        <slot></slot> {{ count }}
      </a>
                                    <div class="dropdown-menu dropdown-menu-right" aria-
labelledby="dropdownNotifications">
        <div class="dropdown-header text-center">Notificaciones</div>
        <notification-list-item
          v-for="notification in notifications"
          :notification="notification"
          :key="notification.id"
        ></notification-list-item>
      </div>
  </li>
</template>

<script>
  import NotificationListItem from "./NotificationListItem";

  export default {
    components: { NotificationListItem },
    data(){
      return {
        notifications: [],
        count: ''
      }
    },
    created() {
      axios.get('/notifications')
        .then(res => {
          this.notifications = res.data;
          this.unreadNotifications();
        });

      EventBus.$on('notification-was-read', () => {
        if (this.count === 1){
          return this.count = '';
        }
        this.count--;
      })

      EventBus.$on('notification-was-unread', () => {
        this.count++;
      })
    },
    methods: {
      unreadNotifications(){
        this.count = this.notifications.filter(notification => {
          return notification.read_at === null;
        }).length || ''
      }
    }
  }
```

```
</script>
```

Explicación: Al lado del slot imprimimos el conteo de notificaciones. Añadimos un header a las notificaciones.
Creamos un método para contar el número de notificaciones que no están leídas (notificaciones con read_at = null). Y usando el evento creado al modificar la notificación, actualizamos el valor del número de notificaciones sin leer.

Compilamos los assets: npm run dev
Y comprobamos que en la página se muestran correctamente los cambios.

[Fin video 82]

Vamos a actualizar la versión de Laravel de la 5.7 a la 5.8.
https://laravel.com/docs/5.8/upgrade

Siguiendo las instrucciones de la guia de actualizacion de Laravel:
Primero actualizamos la versión de Laravel y sus dependencias, en el archivo composer.json:

```
"require": {
    "php": "^7.1.3",
    "fideloper/proxy": "^4.0",
    "laravel/framework": "5.8.*",
    "laravel/tinker": "^1.0",
    "pusher/pusher-php-server": "^4.1"
},
"require-dev": {
    "beyondcode/laravel-dump-server": "^1.0",
    "filp/whoops": "^2.0",
    "fzaninotto/faker": "^1.4",
    "laravel/dusk": "^5.0",
    "mockery/mockery": "^1.0",
    "nunomaduro/collision": "^3.0",
    "phpunit/phpunit": "^7.5"
},
```

Luego en el archivo phpunit.xml es necesario cambiar el nombre de las variables de env, a server. También es necesario añadir la variable bcrypt_rounds, y cambiar el nombre al queue_driver por queue_connection:

```
<php>
    <server name="APP_ENV" value="testing"/>
    <server name="BCRYPT_ROUNDS" value="4"/>
```

```
        <server name="CACHE_DRIVER" value="array"/>
        <server name="SESSION_DRIVER" value="array"/>
        <server name="QUEUE_CONNECTION" value="sync"/>
        <server name="MAIL_DRIVER" value="array"/>
        <server name="DB_CONNECTION" value="sqlite"/>
        <server name="DB_DATABASE" value=":memory:"/>
        <server name="BROADCAST_DRIVER" value="log"/>
    </php>
</phpunit>
```

Ahora en el .env y en el .env.dusk.local cambiamos tambien el queue_driver por queue_connection:

```
SESSION_LIFETIME=120
QUEUE_CONNECTION=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
```

Y en el archivo de configuración queue.php (config) también cambiamos el nombre:

```
'default' => env('QUEUE_CONNECTION', 'sync'),
```

Ahora si actualizamos las dependencias que hemos cambiado, en la terminal:

```
composer update
```

Solucionamos ahora los errores que da si ejecutamos phpunit.
En el HasLikesTest, que hace falta que el método setUp retorne void:

```
class HasLikesTest extends TestCase
{
    use RefreshDatabase;

    function setUp(): void
    {
        parent::setUp();

        Schema::create('model_with_likes', function ($table){
            $table->increments('id');
        });
( … )
```

Y en el RegistrationTest al parecer el email que pusimos ahora es válido por lo que lo cambiamos por otro valor inválido para que pase el test:

```
/** @test */
public function the_email_must_be_a_valid_address()
{
    $this->post(
        route('register'),
```

```
            $this->userValidData(['email' => 'invalidemail'])
        )->assertSessionHasErrors('email');

    }
```

Una vez solucionados los errores de los tests de phpunit, revisamos el contenido del archivo package.json y lo cambiamos por el del github de la versión de laravel 5.8

https://github.com/laravel/laravel/blob/5.8/package.json

El archivo package.json quedaría de la siguiente forma:

```
{
    "private": true,
    "scripts": {
        "dev": "npm run development",
        "development": "cross-env NODE_ENV=development node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js",
        "watch": "npm run development -- --watch",
        "watch-poll": "npm run watch -- --watch-poll",
        "hot": "cross-env NODE_ENV=development node_modules/webpack-dev-server/bin/webpack-dev-server.js --inline --hot --config=node_modules/laravel-mix/setup/webpack.config.js",
        "prod": "npm run production",
        "production": "cross-env NODE_ENV=production node_modules/webpack/bin/webpack.js --no-progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js"
    },
    "devDependencies": {
        "axios": "^0.19",
        "bootstrap": "^4.1.0",
        "browser-sync": "^2.26.7",
        "browser-sync-webpack-plugin": "^2.0.1",
        "cross-env": "^5.1",
        "jquery": "^3.2",
        "laravel-mix": "^4.0.7",
        "lodash": "^4.17.13",
        "popper.js": "^1.12",
        "resolve-url-loader": "^2.3.1",
        "sass": "^1.15.2",
        "sass-loader": "^7.1.0",
        "vue": "^2.5.17",
        "vue-template-compiler": "^2.6.11"
    },
    "dependencies": {
        "laravel-echo": "^1.8.0",
        "pusher-js": "^6.0.3"
    }
}
```

Ahora en la terminal, eliminamos la carpeta node_modules:

sudo rm -r node_modules

Instalamos todo nuevamente: **npm install**
Añadimos laravel-echo y pusher-js: **npm install laravel-echo pusher-js**
Instalamos dependencias adicionales: **npm run watch**
Y compilamos los cambios: **npm run dev**

Si comprobamos ahora nuestra aplicación en el navegador, muestra un error de vue.js de que fallo al montar un componente.
Para solucionarlo, en el archivo app.js de resources/js es necesario agregar default a los requires para que se compilen los componentes:

```
Vue.component('status-form', require('./components/StatusForm').default);
Vue.component('status-list', require('./components/StatusList').default);
Vue.component('status-list-item', require('./components/StatusListItem').default);
Vue.component('friendship-btn', require('./components/FriendshipBtn').default);
Vue.component('accept-friendship-btn', require('./components/AcceptFriendshipBtn').default);
Vue.component('notification-list', require('./components/NotificationList').default);
```

Si ejecutamos ahora los tests de dusk aparece un error: Chrome version must be between 70 and 73.
Para solucionarlo, en el DuskTestCase cambio la ip del RemoteWebDriver por http://localhost:9515
Y luego en Vagrant, en la terminal ssh del proyecto instalo el chrome:

```
sudo apt-get update
sudo apt-get install chromium-browser
```

Si al ejecutar ahora php artisan dusk falla probar a ejecutar: php artisan dusk:chrome-driver
https://github.com/laravel/dusk/issues/683

Ahora si, ejecutamos los tests de dusk y vemos que algunos fallan.
Para solucionar el error del test UsersCanLikeCommentTests users_can_like_and_unlike_comment, temporalmente vamos a agregar el metodo path al modelo Comment.php en (app/models):

```
public function path()
{
  // TODO
}
```

Y en el UsersCanGetTheirNotificationsTest en user_can_see_their_notifications_in_the_nav_bar es necesario añadir una pequeña pausa para que encuentre el texto:

```
$this->browse(function (Browser $browser) use ($user, $notification, $status) {
  $browser->loginAs($user)
```

```
        ->visit('/')
        ->resize(1024, 768)
        ->click('@notifications')
        ->pause(1000)
        ->assertSee('Has recibido un like')
        ->click("@{$notification->id}")
        ->assertUrlIs($status->path())
```

[Fin video 83]

Vamos a implementar las notificaciones en tiempo real.

Comenzamos con un nuevo test en el UsersCanGetTheirNotificationsTest, donde necesitamos 2 usuarios con 2 browsers:

```
/**
* @test
* @throws \Throwable
*/
public function users_can_see_their_notifications_in_real_time()
{
   $user1 = factory(User::class)->create();
   $user2 = factory(User::class)->create();

   $status = factory(Status::class)->create(['user_id' => $user1->id]);

     $this->browse(function (Browser $browser1, Browser $browser2) use ($user1, $user2,
$status) {
       $browser1->loginAs($user1)
           ->visit('/')
           ->resize(1024, 768);

       $browser2->loginAs($user2)
           ->visit('/')
           ->pause(1000) // para que pueda localizar el boton
           ->press('@like-btn')
           ->pause(1000);

       $browser1->assertSeeIn('@notifications-count', 1);
   });
}
```

Necesitamos crear ahora el elemento notifications-count usado en el test.
Para ello, en el NotificationList añadimos el elemento:

```
<template>
   <li class="nav-item dropdown">
       <a dusk="notifications"
```

```
        href="#"
        class="nav-link dropdown-toggle"
        :class="count ? 'text-primary font-weight-bold' : ''"
        id="dropdownNotifications"
        role="button"
        data-toggle="dropdown"
        aria-haspopup="true"
        aria-expanded="false">
         <slot></slot> <span dusk="notifications-count">{{ count }}</span>
      </a>
      ( … )
```

Debemos ahora emitir la notificación para poder escucharla con laravel echo. Por ello debemos asegurarnos que la notificación lleva el método broadcast. Añadimos la verificación en el SendNewNotificationTest:

```
Notification::assertSentTo(
  $statusOwner,
  NewLikeNotification::class,
  function($notification, $channels) use ($likeSender, $status){
     $this->assertContains('database', $channels);
     $this->assertContains('broadcast', $channels);
     $this->assertTrue($notification->model->is($status));
     $this->assertTrue($notification->likeSender->is($likeSender));
     return true;
  });
```

Y ahora en el NewLikeNotification (app/notifications) debemos agregar la llave broadcast, e implementar el metodo toBroadcast que retorna una instancia de BroadcastMessage al que le pasamos la información de la notificación a enviar:

```
( … )
public function via($notifiable)
{
  return ['database', 'broadcast'];
}

( … )

public function toArray($notifiable)
{
  return [
     'link' => $this->model->path(),
     'message' => "Al usuario {$this->likeSender->name} le gustó tu publicación."
  ];
}

public function toBroadcast($notifiable)
{
  return new BroadcastMessage($this->toArray($notifiable));
```

```
}
```

[Fin video 84]

Vamos a verificar en pusher que es lo que emite la notificación broadcast.
https://dashboard.pusher.com/apps/1011151/console/realtime_messages

Damos like a un estado y revisamos pusher, vemos que se dispara un evento ModelLiked cuando se crea un like, y otro api message que es el broadcast de la notificación (con canal private-App.User.3, y nombre del evento BroadcastNotificationCreated).

En nuestra aplicación queremos cada usuario solo tenga acceso a su propio canal de notificaciones.
Automáticamente laravel echo va a hacer una comprobación en el servidor cuando un usuario intente acceder al canal de notificaciones para comprobar que el usuario está autorizado.
Esta verificación se realiza en el archivo channels.php de (routes), donde comprobamos que el id del usuario es igual al id del canal al que queremos acceder.

Ahora necesitamos habilitar estas rutas broadcast, debemos registrar el BroadcastServiceProvider para que funcionen.
En el app.php (/config) descomentamos la linea del BroadcastServiceProvider:

```
/*
* Application Service Providers...
*/
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\BroadcastServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
```

Y ahora en el NotificationList, usamos un canal privado para que cuando el usuario esté autenticado aumente el conteo de notificaciones y hacemos un push de la nueva notificación (usando el método notification de laravel echo):

```
<script>
  import NotificationListItem from "./NotificationListItem";

  export default {
     components: { NotificationListItem },
     data(){
        return {
```

```
            notifications: [],
            count: ''
        }
    },
    created() {
        if (this.isAuthenticated)
        {
            Echo.private(`App.User.${this.currentUser.id}`)
                .notification(notification => {
                    this.count++;
                    this.notifications.push({
                        id: notification.id,
                        data: {
                            link: notification.link,
                            message: notification.message
                        }
                    })
                })
        }

        axios.get('/notifications')
            .then(res => {
                this.notifications = res.data;
                this.unreadNotifications();
            });

( … )
```

Compilamos los cambios realizados: npm run dev
Comprobamos que el test de la lección anterior pasa:

```
php artisan dusk --filter users_can_see_their_notifications_in_real_time
```

Y usando 2 navegadores con 2 usuarios que las notificaciones en tiempo real funcionan correctamente.

[Fin video 85]

En lecciones anteriores vimos que algunos tests de dusk fallaban por no tener implementado el método path en el modelo Comment.

Vamos a añadir que los tests de phpunit también comprueben que falta este método path. Para hacerlo, debemos forzar la implementación de este método path en los componentes que usen el trait HasLikes.

Por ello en el trait HasLikes añadimos el método path como un método abstracto.

También nos damos cuenta que el método str_plural será deprecado en futuras versiones de Laravel, para solucionarlo debemos cambiar str_plural por la clase Str y como método plural.

```
public function eventChannelName()
{
  return strtolower(Str::plural(class_basename($this))) . "." . $this->getKey() . ".likes";
}
```

E importar la clase Str al principio: use Illuminate\Support\Str;

Agregamos ahora el método abstracto path al trait HasLikes, en el final:

```
abstract public function path();
```

Si ejecutamos ahora los tests de phpunit vemos que fallan porque falta por implementar el método path en el HasLikesTest:

```
class ModelWithLike extends Model
{
  use HasLikes;

  public $timestamps = false;

  protected $fillable = ['id'];

  public function path()
  {
    // TODO: Implement path() method.
  }
}
```

[Fin video 86]

Vamos ahora a comprobar que el método path definido anteriormente, devuelve el link al comentario.

En el CommentTest creamos un test para probar que un comentario debe tener un path o link, y que recibe la url correcta de este método path:

```
/** @test */
public function a_comment_must_have_a_path()
{
  $comment = factory(Comment::class)->create();

        $this->assertEquals(route('statuses.show', $comment->status_id) . '#comment-' . $comment->id, $comment->path());
}
```

Explicación: No vamos a crear una vista específica para enviar el comentario. Lo enviamos al estado que contiene el comentario, resaltamos el comentario y hacemos scroll a la ubicación del mismo.
La url es de la forma: *statuses/id del estado#comment-id del comentario*
La ruta del estado del comentario, seguido del #comment y el id del comentario.

Implementamos esto en el modelo Comment:

```
public function path()
{
  return route('statuses.show', $this->status_id) . '#comment-' . $this->id;
}
```

Y en el componente CommentListItem, resaltamos el comentario y agregamos el scroll al comentario:

```
<template>
  <div :class="highlight" :id="`comment-${comment.id}`" class="d-flex">
                              <img class="rounded shadow-sm mr-2" height="34px"
width="34px" :src="comment.user.avatar" :alt="comment.user.name">
    <div class="flex-grow-1">

( … )

<script>
  import LikeBtn from './LikeBtn';

  export default {
    components: { LikeBtn },
    props: {
      comment: {
        type: Object,
        required: true
      }
    },
    mounted() {
      Echo.channel(`comments.${this.comment.id}.likes`)
        .listen('ModelLiked', comment => {
          this.comment.likes_count++;
        });

      Echo.channel(`comments.${this.comment.id}.likes`)
        .listen('ModelUnliked', comment => {
          this.comment.likes_count--;
        });
    },
    computed: {
      highlight() {
        if(window.location.hash === `#comment-${this.comment.id}`)
```

```
            {
                return 'highlight';
            }
        }
    }
  }
</script>

<style>
  .highlight {
      background-color: #ececec;
      padding: 10px;
      border-left: 4px solid #ff8d00;
  }
</style>
```

Explicación: Usamos un v-bind para las clases para añadir una propiedad calculada llamada highlight, que verifica que si el hash de la url es igual al hash del comentario le agrega al comentario la clase highlight.
Esta clase highlight la definimos abajo dándole un estilo diferente al comentario.
Y hacemos que se haga scroll al comentario automáticamente al agregar un id al elemento que sea igual al hash de la url.

Compilamos los cambios realizados: npm run dev
Comprobamos que el test del CommentTest (a_comment_must_have_a_path) pasa.
Y en el navegador comprobamos que los cambios se han aplicado.

[Fin video 87]

Vamos a hacer que cuando un usuario comente un estado, se mande una notificación al dueño del estado.

Necesitamos para ello un listener, es decir: disparamos un evento, luego un listener escucha ese evento y envia la notificacion.

Creamos un test unitario para que nos ayude a crear el listener:

```
php artisan make:test Listeners/SendNewCommentNotificationTest --unit
```

Abrimos el nuevo test SendNewCommentNotificationTest:

```
namespace Tests\Unit\Listeners;

use App\Events\CommentCreated;
use App\Models\Comment;
```

```
use App\Models\Status;
use App\Notifications\NewCommentNotification;
use Notification;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class SendNewCommentNotificationTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function a_notification_is_sent_when_a_status_receives_a_new_comment()
  {
    Notification::fake();

    $status = factory(Status::class)->create();
    $comment = factory(Comment::class)->create(['status_id' => $status->id]);

    CommentCreated::dispatch($comment);

    Notification::assertSentTo(
      $status->user,
      NewCommentNotification::class,
      function ($notification, $channels) use ($comment) {
        $this->assertContains('database', $channels);
        $this->assertTrue($notification->comment->is($comment));
        return true;
      }
    );

  }
}
```

Explicación: Hacemos un fake de la notificación, necesitamos un estado y un comentario que pertenece al estado. Despachamos el evento CommentCreated.
Verificamos que una notificación NewCommentNotification fue enviada al dueño del estado.
Y pasamos una función para verificar la notificación y los canales. Verificando que existe la llave database en los canales de la notificación, y que el comentario que recibimos de la notificación es el mismo que se envía a través del evento CommentCreated.

Creamos la notificación:

```
php artisan make:notification NewCommentNotification
```

Creamos el listener, al que le pasamos el evento CommentCreated:

```
php artisan make:listener SendNewCommentNotification -e CommentCreated
```

Ahora para conectar el evento con el listener, debemos ir al EventServiceProvider (app/providers) y añadirlo:

```
protected $listen = [
  'App\Events\ModelLiked' => [
     'App\Listeners\SendNewLikeNotification',
  ],
  'App\Events\CommentCreated' => [
     'App\Listeners\SendNewCommentNotification',
  ],
];
```

Enviamos ahora la notificación en el SendNewCommentNotification al dueño del estado (accedemos al comentario a través del evento, luego el estado a través del comentario y finalmente al usuario) :

```
public function handle(CommentCreated $event)
{
  $statusOwner = $event->comment->status->user;

  $statusOwner->notify(
     new NewCommentNotification($event->comment)
  );
}
```

Como no hemos definido todavía la relacion status en el comentario, vamos al CommentTest y creamos un nuevo test:

```
/** @test */
public function a_comment_belongs_to_a_status()
{
  $comment = factory(Comment::class)->create();

  $this->assertInstanceOf(Status::class, $comment->status);
}
```

Luego en el modelo Comment, creamos la relación con el status:

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

Si ejecutamos ahora el test creado en el CommentTest vemos que pasa.

Continuamos en el NewCommentNotification añadiendo la llave database, agregando el comentario por el constructor, el index link del link del comentario, y la

llave message:

```
namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class NewCommentNotification extends Notification
{
  use Queueable;

  public $comment;

  /**
   * Create a new notification instance.
   *
   * @param $comment
   */
  public function __construct($comment)
  {
      $this->comment = $comment;
  }

  /**
   * Get the notification's delivery channels.
   *
   * @param  mixed  $notifiable
   * @return array
   */
  public function via($notifiable)
  {
      return ['database'];
  }

  /**
   * Get the mail representation of the notification.
   *
   * @param  mixed  $notifiable
   * @return \Illuminate\Notifications\Messages\MailMessage
   */
  public function toMail($notifiable)
  {
      return (new MailMessage)
                  ->line('The introduction to the notification.')
                  ->action('Notification Action', url('/'))
                  ->line('Thank you for using our application!');
  }

  /**
   * Get the array representation of the notification.
```

```
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function toArray($notifiable)
    {
        return [
            'link' => $this->comment->path(),
            'message' => "{$this->comment->user->name} comentó tu publicación."
        ];
    }
}
```

Finalmente creamos un nuevo test unitario para verificar cosas específicas del NewCommentNotification:

php artisan make:test Notifications/NewCommentNotificationTest --unit

Y en el NewCommentNotificationTest creado verificamos que la notificacion se guarda en la base de datos:

```
namespace Tests\Unit\Notifications;

use App\Models\Status;
use App\Notifications\NewLikeNotification;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class NewLikeNotificationTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function the_notification_is_stored_in_the_database()
    {
        $statusOwner = factory(User::class)->create();
        $likeSender = factory(User::class)->create();

        $status = factory(Status::class)->create(['user_id' => $statusOwner->id]);

        $status->likes()->create(['user_id' => $likeSender->id]);

        $statusOwner->notify(new NewLikeNotification($status, $likeSender));

        $this->assertCount(1, $statusOwner->notifications);

        $notificationsData = $statusOwner->notifications->first()->data;

        $this->assertEquals($status->path(), $notificationsData['link']);
                $this->assertEquals("Al usuario {$likeSender->name} le gustó tu publicación.",
$notificationsData['message']);
```

```
  }
}
```

Explicación: Enviamos la notificación al dueño del estado. Y de forma similar al NewLikeNotificationTest, comprobamos que recibimos 1 al llamar a las notificaciones del usuario. Luego obtenemos la información de la primera notificación, y verificamos que recibimos el link del comentario y un mensaje.

Por último comprobamos que los tests creados en (SendNewCommentNotificationTest y NewCommentNotificationTest) pasan.
Y que en el navegador a comentar un estado aparece la notificacion.

[Fin video 88]

Vamos a permitir que aparezca la notificación en tiempo real cuando un usuario recibe un comentario.

Siguiendo los mismos pasos que para las notificaciones de los likes, empezamos creando unos tests para que nos guíen en el proceso:

En el SendNewCommentNotificationTest, añadimos una verificación para el canal broadcast, y que recibimos una instancia de la clase BroadcastMessage al llamar al metodo toBroadcast (importando la clase BroadcastMessage):

```
/** @test */
public function a_notification_is_sent_when_a_status_receives_a_new_comment()
{
  Notification::fake();

  $status = factory(Status::class)->create();
  $comment = factory(Comment::class)->create(['status_id' => $status->id]);

  CommentCreated::dispatch($comment);

  Notification::assertSentTo(
    $status->user,
    NewCommentNotification::class,
    function ($notification, $channels) use ($comment, $status) {
      $this->assertContains('database', $channels);
      $this->assertContains('broadcast', $channels);
      $this->assertTrue($notification->comment->is($comment));
      $this->assertInstanceOf(BroadcastMessage::class, $notification->toBroadcast($status->user));
      return true;
```

```
        }
    );

}
```

Ahora en el NewCommentNotification agregamos el broadcast, y el metodo toBroadcast (importando la clase también):

```
( … )
public function via($notifiable)
{
  return ['database', 'broadcast'];
}

( … )

public function toBroadcast($notifiable)
{
  return new BroadcastMessage($this->toArray($notifiable));
}
```

Si ejecutamos ahora el test anterior a_notification_is_sent_when_a_status_receives_a_new_comment, vemos que pasa.

Finalmente agregamos una verificación con laravel dusk.
En el UsersCanGetTheirNotificationsTest, cambiamos el nombre al test users_can_see_their_notifications_in_real_time por users_can_see_their_like_notifications_in_real_time. Y creamos un nuevo test muy similar al que le hemos cambiado el nombre, donde verificamos las notificaciones de los comentarios en tiempo real:

```
/**
* @test
* @throws \Throwable
*/
public function users_can_see_their_comment_notifications_in_real_time()
{
  $user1 = factory(User::class)->create();
  $user2 = factory(User::class)->create();

  $status = factory(Status::class)->create(['user_id' => $user1->id]);

    $this->browse(function (Browser $browser1, Browser $browser2) use ($user1, $user2,
$status) {
      $browser1->loginAs($user1)
          ->visit('/')
          ->resize(1024, 768);

      $browser2->loginAs($user2)
          ->visit('/')
```

```
            ->pause(1000)
            ->type('comment', 'Mi comentario') // escribimos el comentario
            ->pause(2000)
            ->press('@comment-btn') // enviamos el comentario
            ->pause(2000);

        $browser1->assertSeeIn('@notifications-count', 1);
    });
}
```

Las pausas en el test son necesarias para que el test me pase.

Si ejecutamos el test de dusk creado vemos que pasa sin modificar nada, esto es porque en el NotificationList.vue estamos escuchando las notificaciones de un usuario específico, pero de cualquier tipo (de estados, comentarios, y otras que creemos a futuro).

[Fin video 89]

Vamos a hacer reestructuraciones a los controladores para mejorar la legibilidad y usabilidad del código.

En el FriendshipsController, al momento de enviar una solicitud de amistad no está claro el uso del modelo Friendship. Y en el AcceptFriendshipsController igual.

Comenzamos en el FriendshipsController extrayendo la lógica del modelo Friendship a una nueva función que manda la solicitud al recipient, y usando request para acceder al usuario actualmente autenticado.

```
use App\User;
use App\Models\Friendship;
use Illuminate\Http\Request;

class FriendshipsController extends Controller
{
    public function store(Request $request, User $recipient)
    {
        if (auth()->id() === $recipient->id)
        {
            abort(400);
        }

        $friendship = $request->user()->sendFriendRequestTo($recipient);

        return response()->json([
            'friendship_status' => $friendship->fresh()->status
```

```
        ]);
    }
```

Agregamos en el UserTest un test unitario para probar que el método sendFriendRequestTo devuelve el sender y recipient iguales a los creados arriba:

```
/** @test */
public function user_can_send_friend_requests()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    $friendship = $sender->sendFriendRequestTo($recipient);

    $this->assertTrue($friendship->sender->is($sender));
    $this->assertTrue($friendship->recipient->is($recipient));
}
```

Implementamos ahora en el modelo User el metodo sendFriendRequestTo (importando el modelo Friendship):

```
public function sendFriendRequestTo($recipient)
{
    return Friendship::firstOrCreate([
        'sender_id' => $this->id, //accedemos al mismo objeto -> id
        'recipient_id' => $recipient->id
    ]);
}
```

Continuamos en el AcceptFriendshipsController extrayendo la logica del Friendship, y en su lugar acceder al usuario y usar el metodo acceptFriendRequestFrom con el sender como parametro:

```
public function store(Request $request, User $sender)
{
    $request->user()->acceptFriendRequestFrom($sender);

    return response()->json([
        'friendship_status' => 'accepted'
    ]);
}
```

Agregamos en el UserTest un nuevo test para probar que el método acceptFriendRequestFrom devuelve accepted en el estado de la solicitud:

```
/** @test */
public function user_can_accept_friend_requests()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();
```

```
    $sender->sendFriendRequestTo($recipient);

    $friendship = $recipient->acceptFriendRequestFrom($sender);

    $this->assertEquals('accepted', $friendship->status);
}
```

Y en el modelo User agregamos el nuevo método acceptFriendRequestFrom, guardamos el resultado en una variable, actualizamos el estado de la solicitud, y retornamos el objeto:

```
public function acceptFriendRequestFrom($sender)
{
    $friendship = Friendship::where([
        'sender_id' => $sender->id,
        'recipient_id' => $this->id
    ])->first();

    $friendship->update(['status' => 'accepted']);

    return $friendship;
}
```

Finalmente si ejecutamos todos los tests de phpunit vemos que pasan.

[Fin video 90]

Continuamos con la reestructuración.

En el AcceptFriendshipsController de forma similar a lo hecho anteriormente, reestructuramos el destroy e index usando nuevos métodos que definiremos en el modelo User:

```
namespace App\Http\Controllers;

use App\Models\Friendship;
use App\User;
use Illuminate\Http\Request;

class AcceptFriendshipsController extends Controller
{
    public function index(Request $request)
    {
        return view('friendships.index', [
            'friendshipRequests' => $request->user()->friendshipRequestsReceived
        ]);
```

```
    }

    public function store(Request $request, User $sender)
    {
        $request->user()->acceptFriendRequestFrom($sender);

        return response()->json([
            'friendship_status' => 'accepted'
        ]);
    }

    public function destroy(Request $request, User $sender)
    {
        $request->user()->denyFriendRequestFrom($sender);

        return response()->json([
            'friendship_status' => 'denied'
        ]);
    }
}
```

Para el método index, es importante que se carguen solo las solicitudes en las que el recipient_id es el usuario actualmente autenticado. Para probar que esto ocurre, creamos un nuevo test en el CanRequestFriendshipTest:

```
/** @test */
public function can_get_all_friendship_requests_received()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    $sender->sendFriendRequestTo($recipient);
    factory(Friendship::class, 2)->create();

    $this->actingAs($recipient);

    $response = $this->get(route('accept-friendships.index'));

    $this->assertCount(1, $response->viewData('friendshipRequests'));
}
```

Explicación: A través del sender enviamos la solicitud al recipient, creamos un par de solicitudes más para probar que recibimos solo la que nos pertenece.
Actuamos como el recipient, accedemos a la ruta, y verificamos que solamente hay una solicitud en la respuesta (aunque haya más en la base de datos).

En el UserTest creamos los tests para probar los métodos nuevos usados:

```
/** @test */
```

```
public function user_can_deny_friend_requests()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $sender->sendFriendRequestTo($recipient);

  $friendship = $recipient->denyFriendRequestFrom($sender);

  $this->assertEquals('denied', $friendship->status);
}

/** @test */
public function user_can_get_all_their_friend_requests()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $sender->sendFriendRequestTo($recipient);

  $this->assertCount(0, $recipient->friendshipRequestsSent);
  $this->assertCount(1, $recipient->friendshipRequestsReceived);
        $this->assertInstanceOf(Friendship::class,  $recipient->friendshipRequestsReceived->first());

  $this->assertCount(1, $sender->friendshipRequestsSent);
  $this->assertCount(0, $sender->friendshipRequestsReceived);
  $this->assertInstanceOf(Friendship::class, $sender->friendshipRequestsSent->first());

}
```

Explicación: En el test para comprobar que un usuario puede obtener sus notificaciones de amistad, verificamos que recibimos una instancia del modelo Friendship al llamar al friendshipRequestsReceived y al friendshipRequestsSent del recipient y sender respectivamente.
También verificamos que devuelva 1 en el número de solicitudes recibidas del recipient, y 1 en el número de solicitudes enviadas por el sender.

En el modelo User, añadimos los métodos para realizar las reestructuraciones que hemos definido en los tests anteriores:

```
namespace App;

use App\Models\Friendship;
use App\Models\Status;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
```

```
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $guarded = [];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    protected $appends = ['avatar'];

    public function getRouteKeyName()
    {
        return 'name';
    }

    public function link()
    {
        return route('users.show', $this);
    }

    public function avatar()
    {
        return 'https://aprendible.com/images/default-avatar.jpg';
    }

    public function getAvatarAttribute()
    {
        return $this->avatar();
    }

    public function statuses()
    {
        return $this->hasMany(Status::class);
    }

    public function friendshipRequestsReceived()
    {
        return $this->hasMany(Friendship::class, 'recipient_id');
    }

    public function friendshipRequestsSent()
    {
```

```
        return $this->hasMany(Friendship::class, 'sender_id');
    }

    public function sendFriendRequestTo($recipient)
    {
        return $this->friendshipRequestsSent()
            ->firstOrCreate(['recipient_id' => $recipient->id]);
    }

    public function acceptFriendRequestFrom($sender)
    {
        $friendship = $this->friendshipRequestsReceived()
            ->where(['sender_id' => $sender->id])
            ->first();

        $friendship->update(['status' => 'accepted']);

        return $friendship;
    }

    public function denyFriendRequestFrom($sender)
    {
        $friendship = $this->friendshipRequestsReceived()
            ->where(['sender_id' => $sender->id])
            ->first();

        $friendship->update(['status' => 'denied']);

        return $friendship;
    }
}
```

Explicación: Finalmente para mejorar la legibilidad del código, en el `sendFriendRequestTo` y `acceptFriendRequestFrom`, en lugar de usar el modelo Friendship usamos la relación con las solicitudes enviadas y la relación con las solicitudes recibidas respectivamente.

Finalmente si ejecutamos todos los tests de phpunit vemos que pasan.

[Fin video 91]

Vamos a arreglar que en el navegador cuando iniciamos sesión en la aplicación se mantenga el valor del email.

En la vista login.blade.php (resources/views/auth):

```
( … )
```

```
<div class="form-group">
  <label>Email:</label>
  <input class="form-control border-0" type="email" name="email" placeholder="Tu email..."
        value="{{ old('email') }}">
</div>
( … )
```

En la vista del registro register.blade.php hacemos lo mismo para los otros campos:

```
( … )
<form action="{{ route('register') }}" method="POST">
  @csrf
  <div class="card-body">
    <div class="form-group">
      <label>Username:</label>
            <input class="form-control border-0" type="text" name="name" placeholder="Tu
nombre de usuario..." value="{{ old('name') }}">
    </div>
    <div class="form-group">
      <label>Nombre:</label>
        <input class="form-control border-0" type="text" name="first_name" placeholder="Tu
nombre..." value="{{ old('first_name') }}">
    </div>
    <div class="form-group">
      <label>Apellido:</label>
        <input class="form-control border-0" type="text" name="last_name" placeholder="Tu
apellido..." value="{{ old('last_name') }}">
    </div>
    <div class="form-group">
      <label>Email:</label>
          <input class="form-control border-0" type="email" name="email" placeholder="Tu
email..." value="{{ old('email') }}">
    </div>
    <div class="form-group">
      <label>Contraseña:</label>
                <input class="form-control border-0" type="password" name="password"
placeholder="Tu contraseña...">
    </div>
    <div class="form-group">
      <label>Repite la contraseña:</label>
       <input class="form-control border-0" type="password" name="password_confirmation"
placeholder="Repite tu contraseña...">
    </div>

    <button class="btn btn-primary btn-block" dusk="register-btn">Registrarse</button>
  </div>
</form>
( … )
```

Ahora agregaremos un link en el navbar que nos lleve a la página que contiene todas las solicitudes de amistad. Y además le daremos estilo a esta página.

En el partial nav.blade.php (resources/views/partials) añadimos el link a las solicitudes de amistad:

```
( … )
<ul class="navbar-nav ml-auto">
  @guest()
    <li class="nav-item"><a href="{{ route('register') }}" class="nav-link">Register</a></li>
    <li class="nav-item"><a href="{{ route('login') }}" class="nav-link">Login</a></li>
  @else
        <li class="nav-item"><a href="{{ route('accept-friendships.index') }}" class="nav-link">Solicitudes</a></li>
    <notification-list><i class="fa fa-bell"></i></notification-list>

    <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          {{ Auth::user()->name }}
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="{{ route('users.show', Auth::user()) }}">Perfil</a>
          <div class="dropdown-divider"></div>
            <a onclick="document.getElementById('logout').submit()" class="dropdown-item" href="#">Cerrar sesión</a>
        </div>
    </li>

    <form id="logout" action="{{ route('logout') }}" method="POST">@csrf</form>
  @endguest
</ul>
( … )
```

Y en el componente AcceptFriendshipBtn (resources/js/components) que se encarga de mostrar esta vista de las solicitudes, le damos formato:

```
<template>
<div class="d-flex justify-content-between bg-light p-3 rounded mb-3 shadow-sm">
  <div>
    <div v-if="localFriendshipStatus === 'pending'">
      <span v-text="sender.name"></span> te ha enviado una solicitud de amistad
    </div>

    <div v-if="localFriendshipStatus === 'accepted'">
      Tu y <span v-text="sender.name"></span> son amigos
    </div>

    <div v-if="localFriendshipStatus === 'denied'">
      Solicitud denegada de <span v-text="sender.name"></span>
    </div>

    <div v-if="localFriendshipStatus === 'deleted'">
```

```
            Solicitud eliminada de <span v-text="sender.name"></span>
        </div>
    </div>

    <div>
            <button class="btn btn-sm btn-success" v-if="localFriendshipStatus === 'pending'"
@click="acceptFriendshipRequest">Aceptar solicitud</button>
            <button class="btn btn-sm btn-secondary" v-if="localFriendshipStatus === 'pending'"
dusk="deny-friendship" @click="denyFriendshipRequest">Denegar solicitud</button>
                <button class="btn btn-sm btn-dark" v-if="localFriendshipStatus !== 'deleted'"
dusk="delete-friendship" @click="deleteFriendship">Eliminar</button>
    </div>
</div>
</template>
```

Finalmente comprobamos que los tests de esta lección, todos los de phpunit y los de dusk pasan.

[Fin video 92]

Vamos a crear una sección para mostrar el listado de amigos.

Empezamos creando un test:

```
php artisan make:test CanSeeFriendsTest
```

Abrimos el CanSeeFriendTest y agregamos un test para probar que un usuario puede ver el listado de amigos, y otro para que no lo puedan ver los invitados:

```
namespace Tests\Feature;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanSeeFriendsTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function guests_cannot_access_the_list_of_friends()
    {
        $this->get(route('friends.index'))->assertRedirect('login');
    }

    /** @test */
```

```
    public function a_user_can_see_a_list_of_friends()
    {
        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        factory(Friendship::class)->create([
            'recipient_id' => $recipient->id,
            'sender_id' => $sender->id,
            'status' => 'accepted'
        ]);

        $this->actingAs($sender)->get(route('friends.index'))->assertSee($recipient->name);
        $this->actingAs($recipient)->get(route('friends.index'))->assertSee($sender->name);
    }
}
```

Explicación: Creamos un usuario sender, un recipient y una amistad entre los dos. Luego actuando como el sender, accedemos a la ruta friends.index (no creada todavía) y comprobamos que vemos el nombre del amigo.
Y hacemos la misma comprobación a la inversa.

Creamos la ruta friends.index en el archivo de rutas web.php:

```
// Friends routes
Route::get('friends', 'FriendsController@index')->name('friends.index')->middleware('auth');
```

Creamos el FriendsController en la terminal:

```
php artisan make:controller FriendsController
```

Agregamos el método index al FriendsController, donde retornamos la vista friends.index (todavía no creada) con la variable friends donde le pasamos todos los amigos con un método friends (también NO creado):

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class FriendsController extends Controller
{
    public function index()
    {
        return view('friends.index', [
            'friends' => auth()->user()->friends()
        ]);
    }
}
```

Creamos la vista friends en resources/views, en un nuevo archivo llamado

index.blade.php en una nueva carpeta llamada friends.
En esta vista index.blade.php imprimimos los nombres de los amigos:

```
@extends('layouts.app')

@section('content')

<div class="container">
  @forelse($friends as $friend)
  <p>{{ $friend->name }}</p>
  @empty
     No tienes amigos :(
  @endforelse
</div>

@endsection
```

Ahora vamos al UserTest para crear un nuevo test para probar el método friends():

```
/** @test */
public function user_can_get_their_friends()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $sender->sendFriendRequestTo($recipient);

  $this->assertCount(0, $recipient->friends());
  $this->assertCount(0, $sender->friends());

  $recipient->acceptFriendRequestFrom($sender);

  $this->assertCount(1, $recipient->friends());
  $this->assertCount(1, $sender->friends());

  $this->assertEquals($recipient->name, $sender->friends()->first()->name);
  $this->assertEquals($sender->name, $recipient->friends()->first()->name);
}
```

Explicación: Usando el sender y el recipient, agregamos una solicitud de amistad. Comprobamos que al llamar al método friends recibimos 0. Luego al aceptar la solicitud el recipient, comprobamos que recibimos 1 al llamar al método friends.

Finalmente, comprobamos que recibimos un usuario al comprobar que el nombre del recipient es igual al nombre del primer usuario del sender (porque solo hay 1), y a la inversa.

Continuamos en el modelo User creando el método friends:

```
public function friends()
{
        $senderFriends = $this->belongsToMany(User::class, 'friendships', 'sender_id', 'recipient_id')
    ->wherePivot('status', 'accepted')
    ->get();

      $recipientFriends = $this->belongsToMany(User::class, 'friendships', 'recipient_id', 'sender_id')
    ->wherePivot('status', 'accepted')
    ->get();

  return $senderFriends->merge($recipientFriends);
}
```

Explicación: Creamos una relación belongsToMany del modelo User, a traves de la tabla friendships, con el nombre del campo sender_id y recipient_id para obtener los usuarios que aceptaron las solicitudes de amistad del sender .
Restringimos las solicitudes que obtenemos con wherePivot, obteniendo solo las que tengan el estado accepted.
Luego hacemos lo mismo para obtener los amigos del recipient.
Y hacemos una mezcla las relaciones del sender y del recipient.

Y en el nav.blade.php de resources/views/partials agregamos un link para ir al listado de amigos:

```
( … )
<ul class="navbar-nav ml-auto">
  @guest()
    <li class="nav-item"><a href="{{ route('register') }}" class="nav-link">Register</a></li>
    <li class="nav-item"><a href="{{ route('login') }}" class="nav-link">Login</a></li>
  @else
                    <li class="nav-item"><a href="{{ route('friends.index') }}" class="nav-link">Amigos</a></li>
          <li class="nav-item"><a href="{{ route('accept-friendships.index') }}" class="nav-link">Solicitudes</a></li>
    <notification-list><i class="fa fa-bell"></i></notification-list>
( … )
```

Compilamos los cambios: npm run dev
Finalmente comprobamos que los tests de esta lección, todos los de phpunit y los de dusk pasan.
Y realizamos una prueba en el navegador de que se muestra todo correctamente.

[Fin video 93]

Vamos a solucionar problemas con los tests de dusk.

En el video le ocurre un problema de que al ejecutar los tests de dusk: la sesión no fue creada, la versión de chromedriver solo soporta la versión 74.
Para solucionarlo actualiza dependencias de composer: **composer update**
Luego instala el chrome-driver: **php artisan dusk:chrome-driver**
Estos errores no me aparecen, pero los que siguen si.

Debido a problemas de tiempos de ejecución (hacer un llamado de ajax a la base de datos, y esperar a recibir una respuesta), vamos a intentar minificar los archivos.

Compilamos los cambios con **npm run prod** , para compilar y minificar los archivos.

Al ejecutar ahora los tests de dusk, vemos que pasa a otro error. El error es porque laravel dusk no ve el texto del comentario en el navegador y no lo puede verificar, al ser la ventana muy pequeña.
Para solucionarlo, en el UsersCanLikeCommentsTest users_can_see_likes_in_real_time expandimos la ventana del navegador con maximize:

```
public function users_can_see_likes_in_real_time()
{
  $user = factory(User::class)->create();
  $comment = factory(Comment::class)->create();

  $this->browse(function (Browser $browser1, Browser $browser2) use ($user, $comment){
    $browser1->visit('/');

    $browser2->loginAs($user)
      ->visit('/')
      ->maximize()
      ->waitForText($comment->body)
      ->assertSeeIn('@comment-likes-count', 0)
      ->press('@comment-like-btn')
      ->waitForText('TE GUSTA');

    $browser1->assertSeeIn('@comment-likes-count', 1);

    $browser2->press('@comment-like-btn')
      ->waitForText('ME GUSTA');

    $browser1->pause(2000)->assertSeeIn('@comment-likes-count', 0);
  });
}
```

Pasamos al siguiente error de varios tests, que no encuentra el 1 en el elemento

likes-count. Esto es porque estamos incrementando de más el conteo de likes en el LikeBtn.vue.

Lo solucionamos cambiando este cálculo del frontend a devolver este conteo de likes del llamado de axios.

Comenzamos en el CanLikeStatusesTest verificando que recibimos un fragmento Json con la llave likes_count:

```
function an_authenticated_user_can_like_and_unlike_statuses()
{
  $user = factory(User::class)->create();
  $status = factory(Status::class)->create();

  $this->assertCount(0, $status->likes);

  $response = $this->actingAs($user)->postJson( route('statuses.likes.store', $status));

  $response->assertJsonFragment([
     'likes_count' => 1
  ]);

  $this->assertCount(1, $status->fresh()->likes);

  $this->assertDatabaseHas('likes', ['user_id' => $user->id]);

  // Validacion de quitar like del test an_authenticated_user_can_unlike_statuses borrado
  $response = $this->actingAs($user)->deleteJson( route('statuses.likes.destroy', $status));

  $response->assertJsonFragment([
     'likes_count' => 0
  ]);

  $this->assertCount(0, $status->fresh()->likes);

  $this->assertDatabaseMissing('likes', ['user_id' => $user->id]);

}
```

Luego en el StatusLikesController devolvemos la respuesta tipo Json:

```
class StatusLikesController extends Controller
{
  public function store(Status $status)
  {
     $status->like();

     return response()->json([
        'likes_count' => $status->likesCount()
     ]);
```

```
    }

    public function destroy(Status $status)
    {
        $status->unlike();

        return response()->json([
            'likes_count' => $status->likesCount()
        ]);
    }
}
```

Y hacemos lo mismo para los comentarios. En el CommentLikesCommentTest:

```
function an_authenticated_user_can_like_and_unlike_comments()
{
    \Notification::fake();

    // Para que phpunit ignore el test al ejecutarse: $this->markTestIncomplete();
    $this->withoutExceptionHandling();

    $user = factory(User::class)->create();
    $comment = factory(Comment::class)->create();

    $this->assertCount(0, $comment->likes);

    $response = $this->actingAs($user)->postJson( route('comments.likes.store', $comment));

    $response->assertJsonFragment([
        'likes_count' => 1
    ]);

    $this->assertCount(1, $comment->fresh()->likes);

    $this->assertDatabaseHas('likes', ['user_id' => $user->id]);

    // Comprobamos que podemos quitar likes
        $response = $this->actingAs($user)->deleteJson( route('comments.likes.destroy',
$comment));

    $response->assertJsonFragment([
        'likes_count' => 0
    ]);

    $this->assertCount(0, $comment->fresh()->likes);

    $this->assertDatabaseMissing('likes', ['user_id' => $user->id]);
}
```

Y en el CommentLikesController:

```
class CommentLikesController extends Controller
```

```
{
  public function store(Comment $comment)
  {
    $comment->like();

    return response()->json([
      'likes_count' => $comment->likesCount()
    ]);
  }

  public function destroy(Comment $comment)
  {
    $comment->unlike();

    return response()->json([
      'likes_count' => $comment->likesCount()
    ]);
  }
}
```

Finalmente realizamos la reestructuración del LikeBtn.vue para eliminar el conteo de likes en el propio componente, al haberlo realizado en los controladores:

```
<script>
  export default {
    props: {
      model: {
        type: Object,
        required: true
      },
      url: {
        type: String,
        required: true
      }
    },
    methods: {
      toggle() {
        let method = this.model.is_liked ? 'delete' : 'post';
        axios[method](this.url)
          .then(res => {
            this.model.is_liked = !this.model.is_liked;
            this.model.likes_count = res.data.likes_count;
          })
      }
    },
( … )
```

Por último aparece un error de que no encuentra el botón accept-friendship debido a los cambios que realizamos en lecciones anteriores.

En la vista index.blade de la carpeta friendships, nos llevamos el selector de dusk

del botón para aceptar la solicitud al componente AcceptFriendshipBtn.vue:
En index.blade.php quedaría:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    @foreach($friendshipRequests as $friendshipRequest)
      <accept-friendship-btn
        :sender="{{ $friendshipRequest->sender }}"
        friendship-status="{{ $friendshipRequest->status }}"
      ></accept-friendship-btn>
    @endforeach
  </div>
@endsection
```

Y el AcceptFriendshipBtn:

```
<template>
<div class="d-flex justify-content-between bg-light p-3 rounded mb-3 shadow-sm">
  <div>
    <div v-if="localFriendshipStatus === 'pending'">
      <span v-text="sender.name"></span> te ha enviado una solicitud de amistad
    </div>

    <div v-if="localFriendshipStatus === 'accepted'">
      Tu y <span v-text="sender.name"></span> son amigos
    </div>

    <div v-if="localFriendshipStatus === 'denied'">
      Solicitud denegada de <span v-text="sender.name"></span>
    </div>

    <div v-if="localFriendshipStatus === 'deleted'">
      Solicitud eliminada de <span v-text="sender.name"></span>
    </div>
  </div>

  <div>
      <button class="btn btn-sm btn-success" v-if="localFriendshipStatus === 'pending'"
dusk="accept-friendship" @click="acceptFriendshipRequest">Aceptar solicitud</button>
      <button class="btn btn-sm btn-secondary" v-if="localFriendshipStatus === 'pending'"
dusk="deny-friendship" @click="denyFriendshipRequest">Denegar solicitud</button>
        <button class="btn btn-sm btn-dark" v-if="localFriendshipStatus !== 'deleted'"
dusk="delete-friendship" @click="deleteFriendship">Eliminar</button>
  </div>
</div>
</template>
```

Compilamos los cambios: npm run prod
Comprobamos que todos los de phpunit y los de dusk pasan.

[Fin video 94]

Vamos a diseñar la vista de los amigos, donde usaremos la misma tarjeta para mostrar los amigos que usamos en el perfil de usuario.

Empezamos en la vista show.blade de la carpeta users, donde extraemos esta tarjeta a un nuevo partial para poder reutilizarla.

Dentro de la carpeta partials creamos un nuevo archivo user.blade.php, pegamos la tarjeta (tambien quitamos el friendship-status porque vamos a reestructurarlo), y añadimos un link al nombre del usuario para que lleve a su perfil:

```
<div class="card border-0 bg-light shadow-sm">
  <img src="{{ $user->avatar }}" alt="{{ $user->name }}" class="card-img-top">
  <div class="card-body">
    @if(auth()->id() === $user->id)
            <h5 class="card-title"><a href="{{ route('users.show', $user) }}">{{ $user->name
}}</a><small class="text-secondary">Eres tu</small></h5>
    @else
            <h5 class="card-title"><a href="{{ route('users.show', $user) }}">{{ $user->name
}}</a></h5>
      <friendship-btn
        dusk="request-friendship"
        class="btn btn-primary btn-block"
        :recipient="{{ $user }}"
      ></friendship-btn>
    @endif
  </div>
</div>
```

Y en la vista show.blade de users, incluimos el partial creado:

```
@extends('layouts.app')

@section('content')
  <div class="container">
    <div class="row">
      <div class="col-md-3">
        @include('partials.user')
      </div>
      <div class="col-md-9">
          <status-list
            url="{{route('users.statuses.index', $user)}}"
          ></status-list>
        </div>
      </div>
```

```
    </div>
  </div>

@endsection
```

Luego incluimos el partial también en la vista index.blade de friends, y le pasamos el user para que esté disponible en el include:

```
@extends('layouts.app')

@section('content')

<div class="container">
  <div class="row">
    @forelse($friends as $friend)
      <div class="col-md-3">
        @include('partials.user', ['user' => $friend])
      </div>
    @empty
      No tienes amigos :(
    @endforelse
  </div>
</div>

@endsection
```

A continuación vamos a quitar la propiedad friendshipStatus del componente FriendshipBtn para obtenerla a través de un llamado de ajax.

Primero crearemos un test para verificar la url que devolvera el estado de la solicitud, en la terminal: php artisan make:test CanGetFriendshipTest

Abrimos el test CanGetFriendshipTest creado:

```
namespace Tests\Feature;

use App\Models\Friendship;
use App\User;
use Tests\TestCase;
use Illuminate\Foundation\Testing\WithFaker;
use Illuminate\Foundation\Testing\RefreshDatabase;

class CanGetFriendshipTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function guests_cannot_get_friendships()
  {
    $this->getJson(route('friendships.show', 'angel'))->assertStatus(401);
```

```
    }

    /** @test */
    public function can_get_friendship()
    {
        $sender = factory(User::class)->create();
        $recipient = factory(User::class)->create();

        $friendship = Friendship::create([
            'sender_id' => $sender->id,
            'recipient_id' => $recipient->id
        ]);

        $response = $this->actingAs($sender)->getJson(route('friendships.show', $recipient));

        $response->assertJsonFragment([
            'friendship_status' => $friendship->fresh()->status
        ]);
    }
}
```

Explicación: Necesitamos un usuario que envía la solicitud, otro que recibe, y la solicitud con el modelo Friendship.
Luego estando autenticados como el sender, probamos una ruta friendships.show (no creada aun), y comprobamos que recibimos el friendship_status igual al de la solicitud creada arriba (obteniendo una copia fresca de la base de datos).
Y comprobamos que los invitados no puedan obtener solicitudes de amistad.

Creamos la ruta en el archivo de rutas web.php, que use el método show:

```
// Friendships routes
Route::get('friendships/{recipient}',  'FriendshipsController@show')->name('friendships.show')->middleware('auth');
Route::post('friendships/{recipient}', 'FriendshipsController@store')->name('friendships.store')->middleware('auth');
Route::delete('friendships/{user}', 'FriendshipsController@destroy')->name('friendships.destroy')->middleware('auth');
```

Continuamos en el FriendshipsController agregando el método show, obteniendo la solicitud entre ambos usuarios, y devolviendo como Json el friendship_status:

```
public function show(Request $request, User $recipient)
{
    $friendship = Friendship::betweenUsers($request->user(), $recipient)->first();
    return response()->json([
        'friendship_status' => $friendship->status
    ]);
}
```

Ahora en el componente FriendshipBtn:

```
<script>
  export default {
    props: {
      recipient: {
        type: Object,
        required: true
      }
    },
    data: () => ({          // otra forma de escribir el data()
      friendshipStatus: ''
    }),
    created() {
      axios.get(`/friendships/${this.recipient.name}`)
        .then(res => {
          this.friendshipStatus = res.data.friendship_status
        })
    },
    methods: {
      toggleFriendshipStatus(){
        this.redirectIfGuest();

        let method = this.getMethod();

        axios[method](`friendships/${this.recipient.name}`)
          .then(res => {
            this.friendshipStatus = res.data.friendship_status;
          })
          .catch(err => {
            console.log(err.response.data);
          })
      },
      getMethod(){
        if (this.friendshipStatus === 'pending' || this.friendshipStatus === 'accepted'){
          return 'delete';
        }
        return 'post';
      }
    },
    computed: {
      getText(){
        if (this.friendshipStatus === 'pending')
        {
          return 'Cancelar solicitud';
        }
        if (this.friendshipStatus === 'accepted')
        {
          return 'Eliminar de mis amigos';
        }
        if (this.friendshipStatus === 'denied')
        {
```

```
                return 'Solicitud denegada';
            }
            return 'Solicitar amistad';
        }
    }
  }
</script>
```

Explicación: Ya no necesitamos la propiedad friendshipStatus.
Definimos el método created para hacer una petición a la url de las amistades, y asignar el valor del estado a friendshipStatus.
Y cambiamos el nombre de la variable localFriendshipStatus a friendshipStatus por no haber ya necesidad del prefijo local.

En el UserController tampoco necesitamos el friendship status:

```
namespace App\Http\Controllers;

use App\User;

class UsersController extends Controller
{
  public function show(User $user)
  {
      return view('users.show', compact('user'));
  }
}
```

Finalmente compilamos los cambios: npm run prod
Comprobamos que los tests de phpunit pasan.
Y al comprobar los tests de dusk vemos que algunos de ellos fallan porque estamos realizando un llamado de ajax para obtener la solicitud, debemos esperar a que se realice la petición al momento de visitar la ruta.

Para solucionarlo, en el UserCanRequestFriendshipTest debemos esperar a que aparezca el texto:

```
public function senders_can_create_and_delete_friendship_requests()
{
  $sender = factory(User::class)->create();
  $recipient = factory(User::class)->create();

  $this->browse(function (Browser $browser) use ($sender, $recipient) {
      $browser->loginAs($sender)
              ->visit(route('users.show', $recipient))
              ->press('@request-friendship')
              ->waitForText('Cancelar solicitud')
```

```
            ->assertSee('Cancelar solicitud')
            ->visit(route('users.show', $recipient))
            ->waitForText('Cancelar solicitud')
            ->assertSee('Cancelar solicitud')
            ->press('@request-friendship')
            ->waitForText('Solicitar amistad')
            ->assertSee('Solicitar amistad')
        ;
    });
}

( … )

public function senders_can_delete_accepted_friendship_requests()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'accepted'
    ]);

    $this->browse(function (Browser $browser) use ($sender, $recipient) {
        $browser->loginAs($sender)
            ->visit(route('users.show', $recipient))
            ->waitForText('Eliminar de mis amigos')
            ->assertSee('Eliminar de mis amigos')
            ->press('@request-friendship')
            ->waitForText('Solicitar amistad')
            ->assertSee('Solicitar amistad')
            ->visit(route('users.show', $recipient))
            ->waitForText('Solicitar amistad')
            ->assertSee('Solicitar amistad')
        ;
    });
}

( … )

public function senders_cannot_delete_denied_friendship_requests()
{
    $sender = factory(User::class)->create();
    $recipient = factory(User::class)->create();

    Friendship::create([
        'sender_id' => $sender->id,
        'recipient_id' => $recipient->id,
        'status' => 'denied'
    ]);

    $this->browse(function (Browser $browser) use ($sender, $recipient) {
```

```
        $browser->loginAs($sender)
            ->visit(route('users.show', $recipient))
            ->waitForText('Solicitud denegada')
            ->assertSee('Solicitud denegada')
            ->press('@request-friendship')
            ->waitForText('Solicitud denegada')
            ->assertSee('Solicitud denegada')
            ->visit(route('users.show', $recipient))
            ->waitForText('Solicitud denegada')
            ->assertSee('Solicitud denegada')
        ;
    });
}
```

Ahora si, todos los test de dusk pasan correctamente.

[Fin video 95]

< < < AMPLIACIÓN DEL PROYECTO > > >

[Paso 96 Actualización Laravel 6]

Lo primero es revisar la página de Laravel de los pasos a realizar para actualizar Laravel 5.8.x a Laravel 6.x.
https://laravel.com/docs/6.x/upgrade

El github de la version Laravel 6: https://github.com/laravel/laravel/tree/6.x

Se requiere php en la versión 7.2 o superior, y actualizar las dependencias del composer.json y del package.json.

Esto puede comprobarse al comparar las versiones de los 2 archivos anteriores de nuestro proyecto, con los de la nueva versión de Laravel 6 en Github.

El composer.json quedaría:

```
( … )

"require": {
    "php": "^7.2",
```

```
    "fideloper/proxy": "^4.0",
    "laravel/framework": "6.2",
    "laravel/tinker": "^2.0",
    "pusher/pusher-php-server": "^4.1"
  },
  "require-dev": {
    "facade/ignition": "^1.4",
    "beyondcode/laravel-dump-server": "^1.0",
    "filp/whoops": "^2.0",
    "fzaninotto/faker": "^1.9.1",
    "laravel/dusk": "^6.0",
    "mockery/mockery": "^1.0",
    "nunomaduro/collision": "^3.0",
    "phpunit/phpunit": "^8.0"
  },

  ( … )
```

Y el package.json:

```
  ( … )

  "devDependencies": {
    "axios": "^0.19",
    "bootstrap": "^4.3.1",
    "browser-sync": "^2.26.7",
    "browser-sync-webpack-plugin": "^2.0.1",
    "cross-env": "^7.0",
    "jquery": "^3.2",
    "laravel-mix": "^5.0.1",
    "lodash": "^4.17.13",
    "popper.js": "^1.12",
    "resolve-url-loader": "^3.1.0",
    "sass": "^1.15.2",
    "sass-loader": "^8.0.0",
    "vue": "^2.5.17",
    "vue-template-compiler": "^2.6.11"
  },
  "dependencies": {
    "laravel-echo": "^1.8.0",
    "pusher-js": "^6.0.3"
  }
```

Instalamos el paquete laravel/helpers, pues es necesario para seguir usando los helpers que empiezan por str_ y array_: https://laravel.com/docs/6.x/upgrade#helpers

composer require laravel/helpers

(También se puede actualizar las llamadas a estos métodos como marca la documentación de Laravel. Por ejemplo: str_slug se pasa a Str::slug.
Aunque lo más sencillo es instalar el helper)

Ahora si, actualizamos la versión de Laravel y dependencias: **composer update**

Ahora en la terminal, eliminamos la carpeta node_modules:
sudo rm -r node_modules

Instalamos todo nuevamente: **npm install**
Arreglamos los errores con: **npm audit fix**
Y compilamos los cambios: **npm run prod**

Una vez que termine la actualización, comprobamos que los tests de phpunit pasan.
Pero al ejecutar los tests de dusk aparece un error:
session not created: Chrome version must be between 70 and 73

Esto se soluciona instalando el driver de chrome: **php artisan dusk:chrome-driver**

Finalmente ejecutamos de nuevo los tests de dusk y comprobamos que pasan.

[Paso 97 Personalizando el estilo]

En este paso voy a modificar el estilo de la aplicación.

Agrego un avatar propio en la carpeta public, dónde creo una carpeta llamada img y dentro añado la imagen default-avatar.png.

En app/User.php añado el avatar:

```
public function avatar()
{
  return '/img/default-avatar.png';
}
```

Luego en la vista resources/views/auth/login.blade.php añado unos iconos, cambio los nombres de los input para ser más descriptivo, y agrego estilo con bootstrap:

```
( … )

<form action="{{ route('login') }}" method="POST">
  @csrf
  <div class="card-body">

    <div class="form-group">
          <i class="fas fa-envelope mr-2"></i><label class="font-weight-bold">Correo Electrónico:</label>
        <input class="form-control border-1" type="email" name="email" placeholder="Tu
```

```
email..."
                value="{{ old('email') }}">
        </div>

        <div class="form-group">
          <i class="fas fa-key mr-2"></i><label class="font-weight-bold">Contraseña:</label>
                    <input class="form-control border-1" type="password" name="password"
placeholder="Tu contraseña...">
        </div>

          <button class="btn btn-primary btn-block font-weight-bold btn-success" dusk="login-
btn">Iniciar sesión</button>
    </div>
</form>

( … )
```

En el resources/views/auth/register.blade.php hago algo parecido al login:

```
( … )

<form action="{{ route('register') }}" method="POST">
    @csrf
    <div class="card-body">
        <div class="form-group pb-2">
            <i class="fas fa-user mr-2"></i><label class="font-weight-bold">Apodo:</label>
                <input class="form-control border-1" type="text" name="name" placeholder="Tu
nombre de usuario..." value="{{ old('name') }}">
        </div>
        <div class="form-group pb-2">
            <i class="fas fa-user mr-2"></i><label class="font-weight-bold">Nombre:</label>
              <input class="form-control border-1" type="text" name="first_name" placeholder="Tu
nombre..." value="{{ old('first_name') }}">
        </div>
        <div class="form-group pb-2">
            <i class="fas fa-user mr-2"></i><label class="font-weight-bold">Apellidos:</label>
              <input class="form-control border-1" type="text" name="last_name" placeholder="Tu
apellido..." value="{{ old('last_name') }}">
        </div>
        <div class="form-group pb-2">
                    <i class="fas fa-envelope mr-2"></i><label class="font-weight-bold">Correo
Electrónico:</label>
                <input class="form-control border-1" type="email" name="email" placeholder="Tu
email..." value="{{ old('email') }}">
        </div>
        <div class="form-group pb-2">
            <i class="fas fa-lock mr-2"></i><label class="font-weight-bold">Contraseña:</label>
                    <input class="form-control border-1" type="password" name="password"
placeholder="Tu contraseña...">
        </div>
        <div class="form-group pb-3">
                    <i class="fas fa-unlock mr-2"></i><label class="font-weight-bold">Repetir
```

```
contraseña:</label>
            <input class="form-control border-1" type="password" name="password_confirmation"
placeholder="Repite tu contraseña...">
        </div>

          <button class="btn btn-primary btn-block font-weight-bold btn-success" dusk="register-
btn">Registrarse</button>
    </div>
</form>

( … )
```

En el resources/views/layouts/app.blade.php cambio el nombre al titulo de pestaña:

```
<title>Fakebook</title>
```

En resources/views/partials/nav.blade.php cambio el nombre de la aplicacion que aparece en el navbar, y el de los links de registro e iniciar sesion:

```
<a class="navbar-brand" href="{{ route('home') }}"><i class="fab fa-facebook-square text-
primary mr-2"></i>Fakebook</a>

( … )

@guest()
            <li class="nav-item"><a href="{{ route('register') }}" class="nav-link text-
primary">Registrarse</a></li>
            <li class="nav-item"><a href="{{ route('login') }}" class="nav-link text-primary">Iniciar
sesión</a></li>
@else
```

Y otros pequeños cambios que pueden verse en:
https://github.com/AngelCanovas/social/commit/1121a81afd8ca1f7ff5952cf68ed09429dc8ca2d

[Paso 98 Actualización Laravel 7]

* Incompleto *

https://laravel.com/docs/7.x/upgrade
https://github.com/laravel/laravel
https://styde.net/guia-de-actualizacion-a-laravel-7/

composer update

composer require laravel/ui