TDD: User Feature

Software as a Service - Back-End Development
Session 03

Developed by Adrian Gould

Contents

Creating the User API

We will use the User Model, Migration and Factory from the existing application for this example.

The User API will, eventually, be used by an application to allow the user to update their information.

It could also be used as a way to provide an administration application with full access to users and allow for more control.

The best part of any API development is that they may be used for web, mobile and desktop implementations of applications that may be client or administration focussed.

So, we will implement the User Feature to allow for:

- Add a user
- Edit a user
- Browse Users
- Read a User
- Delete a User

In a future session we will then add security to:

- prevent users who are not logged in to access the feature(s)
- prevent ordinary (client) users from access edit, add and delete features
- allow a client to only see their details and work on their information

Create the User Controller

```
php artisan make:controller UserController
```

We are familiar with this from our previous work.

Define Routes

Edit the `routes/api.php` file to define the API routes:

```
use App\Http\Controllers\UserController;
use Illuminate\Support\Facades\Route;

Route::prefix('v1')->group(function () {
    Route::post('/users', [UserController::class, 'store']);
    Route::get('/users', [UserController::class, 'index']);
    Route::get('/users/{id}', [UserController::class, 'show']);
    Route::patch('/users/{id}', [UserController::class, 'update']);
    Route::delete('/users/{id}', [UserController::class, 'destroy']);
});
```

We will revisit the routs and think about how we can refactor these later...

Create User

Create User Test

Create a test file `tests/Feature/UserTest.php`:

```
php artisan make:test UserCreateTest
```

Add the initial test:

```
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;
use function Pest\Laravel\postJson;

uses(RefreshDatabase::class);

it('can create a user', function () {
    $response = postJson('/api/v1/users', [
        'name' => 'John Doe',
        'email' => 'john@example.com',
```

```
    ]);

    $response->assertStatus(201);
    $this->assertDatabaseHas('users', [
        'name' => 'John Doe',
        'email' => 'john@example.com',
    ]);
});
```

Implement the Create (Store) Method

Edit the `UserController.php` file and add the required code for create (store)...

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'name' => 'required|string|max:255',
            'email' => 'required|email|unique:users,email',
        ]);

        if ($validator->fails()) {
            return response()->json($validator->errors(), 400);
        }

        $user = User::create($request->all());

        return response()->json($user, 201);
    }

/* Trimmed down to reduce the length of the sample code */

}
```

Run the Tests

```
php artisan test
```

or

```
./vendor/bin/pest
```

> *Aside: we can add aliases to BASH for the above, see the article Add Bash Command Line Aliases for Git, Laravel and more on the SQuASH helpdesk.*

Create a User when Missing Data

Next we need to make sure that when we attempt to create a user but are missing data that the API responds appropriately...

Add a new test to the Create User Test file.

```
it('cannot create a user with incomplete data', function () {
    $response = postJson('/api/v1/users', [ 'name' => 'John Doe', ]);
    $response->assertStatus(422) ->assertJsonValidationErrors(['email']);

    $response = postJson('/api/v1/users', [ 'email' =>
'john@example.com', ]);
    $response->assertStatus(422) ->assertJsonValidationErrors(['name']);
});
```

Browse User

Create Browse Users Test

Create a test file `tests/Feature/UserTest.php`:

```
php artisan make:test UserCreateTest
```

Add the initial test:

```
it('can browse users', function () {
    User::factory()->count(3)->create();

    $response = $this->getJson('/api/v1/users');

    $response->assertStatus(200)
             ->assertJsonCount(3);
```

```
});
```

Implement the Browse (index) Method

Edit the `UserController.php` file and add the required code...

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{

/* Trimmed down to reduce the length of the sample code */

    public function index()
    {
        $users = User::all();
        return response()->json($users);
    }

/* Trimmed down to reduce the length of the sample code */

}
```

Read User

Create Read Single User Test

Create a test file `tests/Feature/UserTest.php`:

```
php artisan make:test UserCreateTest
```

Add the initial test:

```
it('can read a single user', function () {
    $user = User::factory()->create();

    $response = $this->getJson("/api/v1/users/{$user->id}");
```

```
    $response->assertStatus(200)
             ->assertJson([
                 'name' => $user->name,
                 'email' => $user->email,
             ]);
});
```

Implement the Read Method

Edit the `UserController.php` file and add the required code...

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{

/* Trimmed down to reduce the length of the sample code */

    public function show($id)
    {
        $user = User::findOrFail($id);
        return response()->json($user);
    }

/* Trimmed down to reduce the length of the sample code */

}
```

Question:

What happens when you try reading a user when:

- No ID provided?
- An invalid ID (ID is not found)?
- An invalid ID (random text used)?

Update User

Create Update User Test

Create a test file `tests/Feature/UserTest.php`:

```
php artisan make:test UserCreateTest
```

Add the initial test:

```
it('can update a user', function () {
    $user = User::factory()->create();

    $response = $this->patchJson("/api/v1/users/{$user->id}", [
        'name' => 'Updated Name',
    ]);

    $response->assertStatus(200);
    $this->assertDatabaseHas('users', [
        'id' => $user->id,
        'name' => 'Updated Name',
    ]);
});
```

Implement the Update Method

Edit the `UserController.php` file and add the required code...

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{

/* Trimmed down to reduce the length of the sample code */

    public function update(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $validator = Validator::make($request->all(), [
            'name' => 'sometimes|required|string|max:255',
            'email' => 'sometimes|required|email|unique:users,email,' .
$user->id,
```

```
        ]);

        if ($validator->fails()) {
            return response()->json($validator->errors(), 400);
        }

        $user->update($request->all());

        return response()->json($user);
    }

/* Trimmed down to reduce the length of the sample code */

}
```

Create Test for Updating User with Missing Data

```
it('cannot update a user with incomplete data', function () {
    $user = User::factory()->create();

    $response = patchJson("/api/v1/users/{$user->id}", [ 'email' =>
'updated@example.com', ]);

    $response->assertStatus(422) ->assertJsonValidationErrors(['name']);
});
```

Does this work as expected when you run the test?

Question:

What happens when you try updating when:

- No ID provided?
- An invalid ID (ID is not found)?
- An invalid ID (random text used)?

Delete User

Create Delete User Test

Create a test file `tests/Feature/UserTest.php`:

```
php artisan make:test UserCreateTest
```

Add the initial test:

```
it('can delete a user', function () {
    $user = User::factory()->create();

    $response = $this->deleteJson("/api/v1/users/{$user->id}");

    $response->assertStatus(204);
    $this->assertDatabaseMissing('users', [
        'id' => $user->id,
    ]);
});
```

Implement the Delete Method

Edit the `UserController.php` file and add the code for the delete method...

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{

/* Trimmed down to reduce the length of the sample code */

    public function destroy($id)
    {
        $user = User::findOrFail($id);
        $user->delete();

        return response()->json(null, 204);
    }
}
```

Test the Delete Method

Run the tests and check that delete works.

Question:

What happens when you try deleting when:

- No ID provided?
- An invalid ID (ID is not found)?
- An invalid ID (random text used)?

Refactoring the Routes

Edit the `routes/api.php` file to use resourceful routing:

```
use App\Http\Controllers\UserController;
use Illuminate\Support\Facades\Route;

Route::prefix('v1')->group(function () {
        Route::apiResource('users', UserController::class);
});
```

The `apiResource` method automatically creates the following routes:

- `GET /users` maps to `UserController@index`
- `POST /users` maps to `UserController@store`
- `GET /users/{user}` maps to `UserController@show`
- `PATCH /users/{user}` maps to `UserController@update`
- `DELETE /users/{user}` maps to `UserController@destroy`

Refactoring the Controllers

Refactoring the controller methods to reduce their complexity and improve readability can be achieved by moving validation logic to Form Requests and using Eloquent resources for response formatting.

Create the `StoreUserRequestTest` Unit Test

To Create a Unit Test using PEST you must manually create the file in the `tests/Unit` folder...

Create a `StoreUserRequestTest.php` file and when you have done this, add the Unit Test Code...

Edit this new file and add the following to test when name and email are given to update a user.

```
use App\Http\Requests\StoreUserRequest;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\ValidationException;

it('validates the store user request with valid data', function () {
    $request = new StoreUserRequest();
    $data = [
        'name' => 'John Doe',
        'email' => 'john@example.com',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->passes())->toBeTrue();
});
```

The next test tests that the validation fails when the name is missing...

```
it('fails validation for the store user request with missing name',
function () {
    $request = new StoreUserRequest();
    $data = [
        'email' => 'john@example.com',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('name'))->toBeTrue();
});
```

Next we test when the email is missing...

```
it('fails validation for the store user request with missing email',
function () {
    $request = new StoreUserRequest();
    $data = [
        'name' => 'John Doe',
    ];
```

```
    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('email'))->toBeTrue();
});
```

Next we test with an invalid email...

```
it('fails validation for the store user request with invalid email',
function () {
    $request = new StoreUserRequest();
    $data = [
        'name' => 'John Doe',
        'email' => 'invalid-email',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('email'))->toBeTrue();
});
```

Create `StoreUserRequest`

```
php artisan make:request StoreUserRequest
```

Edit the `StoreUserRequest.php` file:

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StoreUserRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'name' => 'required|string|max:255',
```

```
            'email' => 'required|email|unique:users,email',
        ];
    }
}
```

Create the `UpdateUserRequestTest` Unit Test

To Create a Unit Test using PEST you must manually create the file in the `tests/Unit` folder...

Create a `UpdateUserRequestTest.php` file and when you have done this, add the Unit Test Code...

Edit this new file and add the following to test when name and email are given to update a user.

```
use App\Http\Requests\UpdateUserRequest;
use App\Models\User;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

it('validates the update user request with valid data', function () {
    $user = User::factory()->create();
    $request = new UpdateUserRequest();
    $request->setUserResolver(function () use ($user) {
        return $user;
    });

    $data = [
        'name' => 'Updated Name',
        'email' => 'updated@example.com',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->passes())->toBeTrue();
});
```

The next test tests that the validation fails when the name is missing...

```
it('fails validation for the update user request with missing name',
function () {
    $user = User::factory()->create();
    $request = new UpdateUserRequest();
    $request->setUserResolver(function () use ($user) {
```

```
        return $user;
    });

    $data = [
        'email' => 'updated@example.com',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('name'))->toBeTrue();
});
```

Next we test when the email is missing...

```
it('fails validation for the update user request with missing email',
function () {
    $user = User::factory()->create();
    $request = new UpdateUserRequest();
    $request->setUserResolver(function () use ($user) {
        return $user;
    });

    $data = [
        'name' => 'Updated Name',
    ];

    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('email'))->toBeTrue();
});
```

Next we test with an invalid email...

```
it('fails validation for the update user request with invalid email',
function () {
    $user = User::factory()->create();
    $request = new UpdateUserRequest();
    $request->setUserResolver(function () use ($user) {
        return $user;
    });

    $data = [
        'name' => 'Updated Name',
        'email' => 'invalid-email',
    ];
```

```
    $validator = Validator::make($data, $request->rules());

    expect($validator->fails())->toBeTrue();
    expect($validator->errors()->has('email'))->toBeTrue();
});
```

Create `UpdateUserRequest`

```
php artisan make:request UpdateUserRequest
```

Edit the `UpdateUserRequest.php` file:

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class UpdateUserRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'name' => 'sometimes|required|string|max:255',
            'email' => [
                'sometimes',
                'required',
                'email',
                Rule::unique('users', 'email')->ignore($this->user),
            ],
        ];
    }
}
```

Create Eloquent `UserResource`

Eloquent resources provide a way to format JSON responses consistently.

```
php artisan make:resource UserResource
```

Edit the `UserResource.php` file:

```
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Create a Unit Test for the UserResource

To Create a Unit Test using PEST you must manually create the file in the `tests/Unit` folder...

Create a `UserResourceTest.php` file and when you have done this, add the Unit Test Code...

```
use App\Http\Resources\UserResource;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;

uses(RefreshDatabase::class);

it('can transform a user model into a JSON resource', function () {
    $user = User::factory()->create([
        'name' => 'John Doe',
        'email' => 'john@example.com',
    ]);

    $resource = new UserResource($user);

    $expectedArray = [
```

```
        'id' => $user->id,
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'created_at' => $user->created_at->toISOString(),
        'updated_at' => $user->updated_at->toISOString(),
    ];

    expect($resource->toArray(request()))->toBe($expectedArray);
});
```

Refactor the User Controller

Update the `UserController.php` to use the Form Requests and Eloquent Resources...

You will see that the code becomes much shorter and easier to read.

Store Method

```
public function store(StoreUserRequest $request)
{
        $user = User::create($request->validated());

        return new UserResource($user);
}
```

Index Method

```
public function index()
{
        $users = User::all();
        return UserResource::collection($users);
}
```

Show Method

```
public function show(User $user)
{
        return new UserResource($user);
}
```

Update Method

```
public function update(UpdateUserRequest $request, User $user)
{
        $user->update($request->validated());

        return new UserResource($user);
}
```

Destroy Method

```
public function destroy(User $user)
{
        $user->delete();

        return response()->json(null, 204);
}
```

Update Tests

Ensure your tests are still valid.

The endpoints and functionality remain the same, but the implementation has been refactored.

Run the tests to make sure!