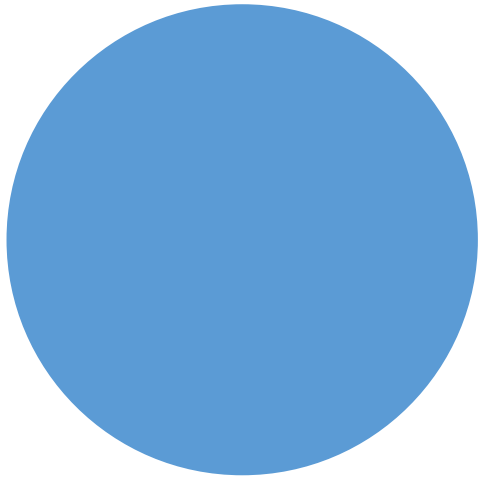
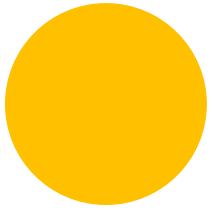



Creating Data

Debugging in Laravel

Debugging

- When we start to create views which allow users to create data things life comes at you fast!
- So we will need to use debugging tools – USE DEBUG TOOLS!!
- Debugging can be tricky because you can’t just output something to a command line
- Luckily Laravel has a nice tool to help you

Debugging with dd

- Laravel provides the command dd.
- This is dying dump. It stops the servicing of a request and instead returns a web page that just displays the argument of the variable.
- If you ever don't really know what the content of a variable is in a controller, then just dd it:

```
$animal = Animal::findOrFail($id);

dd($animal);
```

- Great little tool!!!

Resulting webpage:

```
Animal {#214 ▼
  #connection: "mysql"
  #table: null
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #attributes: array:7 [▼
    "id" => 1
    "name" => "Leo"
    "weight" => 351.6
    "date_of_birth" => null
    "enclosure_id" => 1
    "created_at" => "2018-10-22 19:47:27"
    "updated_at" => "2018-10-22 19:47:27"
  ]
  #original: array:7 [▶]
  #changes: []
  #casts: []
  #dates: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  #hidden: []
  #visible: []
  #fillable: []
  #guarded: array:1 [▶]
}
```

HTTP Verbs Recap and REST

Various HTTP Methods

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred as HTTP verbs.

- GET:
  - The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
- HEAD:
  - The HEAD method asks for a response identical to that of a GET request, but without the response body.
- POST:
  - The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Various HTTP Methods

- PUT
  - The PUT method replaces all current representations of the target resource with the request payload.
- PATCH
  - The PATCH method is used to apply partial modifications to a resource.
- DELETE:
  - The DELETE method deletes the specified resource.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

HTTP Responses

- The response from the sever begins with a status line
- Example – HTTP/1.1 200 OK
- Version
- Status code
  - 1xx – information that things are still going on, usually not used by browsers
  - 2xx – success of some kind, but not all look like success
    i.e. 204 is 'no content'
  - 3xx – Redirection, either things have moved or a proxy server is involved but it means the client needs to do more
  - 4xx – Client error (e.g. 404 not found) – you (client) screwed up
  - 5xx – Server error – I (server) screwed up
- Reason Phrase – English language description of status code

REST

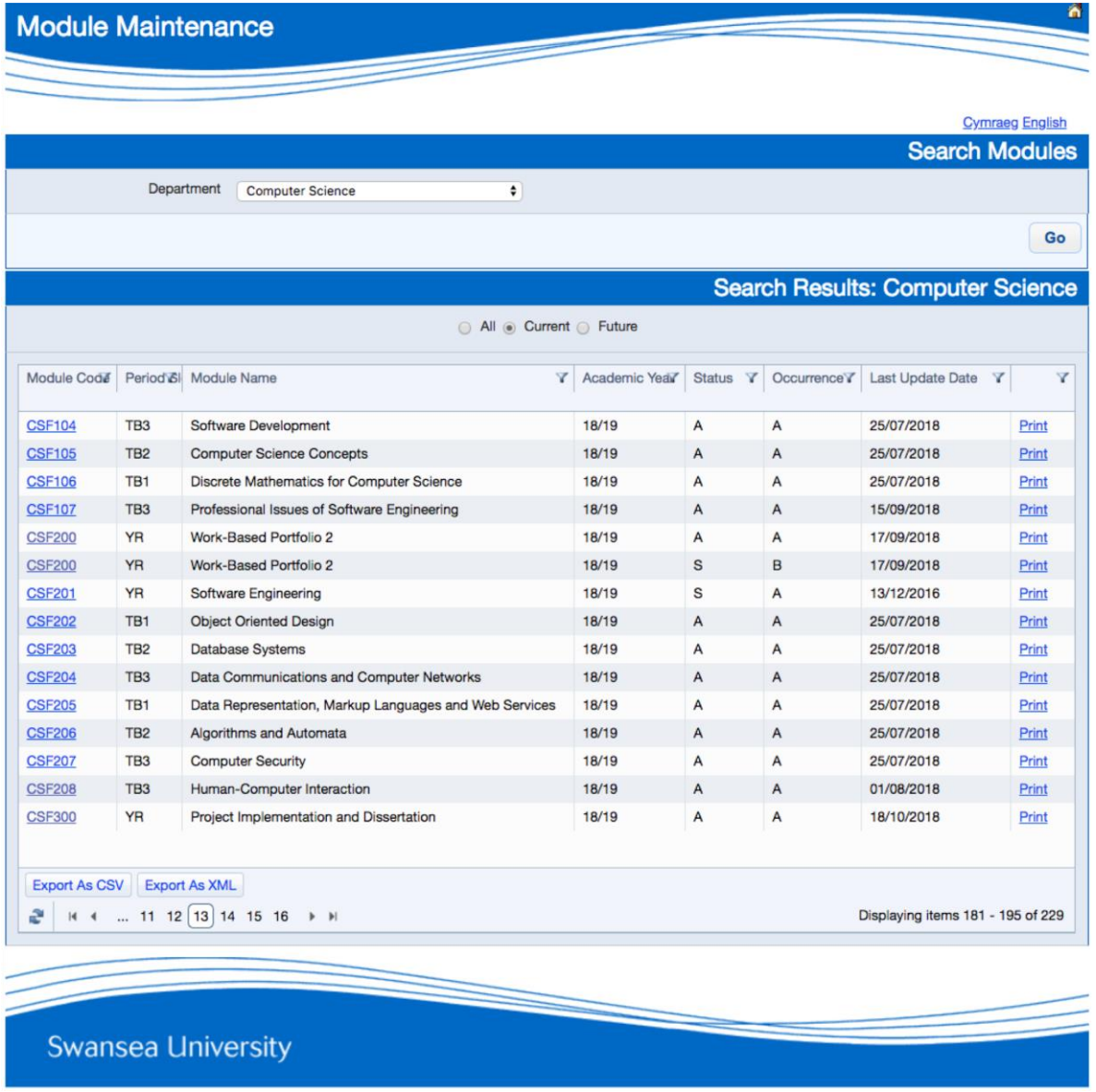
REST (REpresentational State Transfer):

- an architectural style for web page and systems on the web for making communication between systems easier.
- REST-compliant systems (RESTful systems), are characterised by being stateless and separate the concerns of client and server.
- RESTful systems do not store state on the server.

- Every request contains all the information needed to carry out the operation.
  - E.g., “Update animal 42 to have the name to ‘Percy Pig’ ”.

Example of a Non-Restful System

- Swansea University Module Maintenance.
  - The system we use to centrally administer modules and their details.



Example of a Non-Restful System (2)

Consider this sequence of events:

- I open CSF101 to make a change to the syllabus.
- I am about to make the change and think “wait, is this topic I am about to add already covered in CSF102”.
- I open a new tab and navigate to CSF102. I look up the info.
- I go back to CSF101, make the change and click submit.

Guess what happens?

I have just made changes to CSF102!!!!!!!!!!!

- The back-end system kept track that the last page I opened was CSF102.
- When I submitted my changes it assumed it was for CSF102.

- Absolutely awful system!

Reasons for Designing RESTful Systems

- Easier to work with.
- Easier to program.
- No state to keep track of on the back-end
  - Except maybe who is logged on. But normally you only log on as one user. Still – can be tricky.

Reasons for NOT Designing RESTful Systems

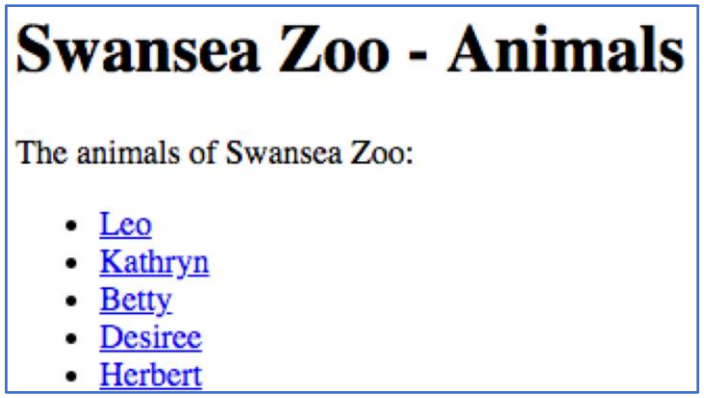
- Consider a Web Game of Connect 4.
- If we use REST then no state is stored on server.
- All state is stored on Client (in Javascript maybe).
- Malicious user can manipulate the client state (e.g., chrome developer tools) and cheat. E.g., change the grid.
- When they submit their move, they will submit all state information, including their changes to the grid.

With a Non-RESTful system, the game state would be kept track of on the server.

Creating Resources

Recap – Displaying All Animals - animals.index

Page to show all animals



Route in routes/web.php:

```
Route::get('animals', 'AnimalController@index');
```

Controller: app/Http/Controllers/AnimalController.php

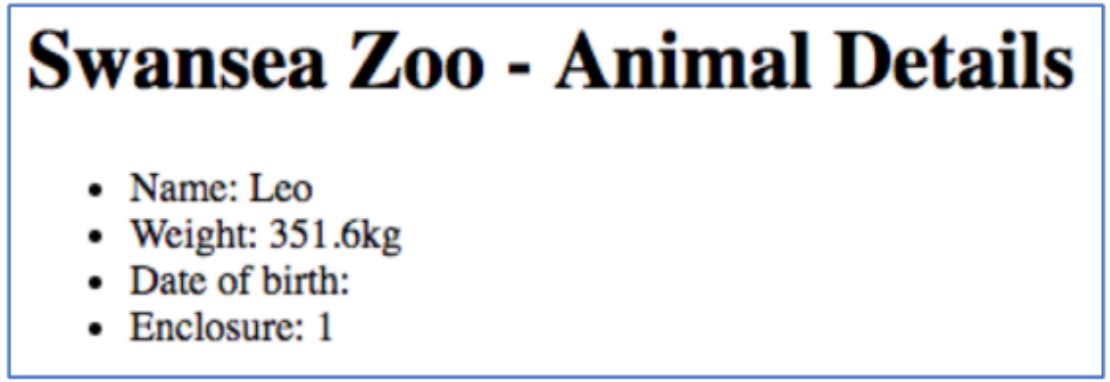
```php
public function index()
{
    $animals = Animal::all();
    return view('animals.index', ['animals' => $animals]);
}
```

Recap – Displaying All Animals - animals.index (Cont.)

View: resources/views/animals/index

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


@section('title', 'Animals')


@section('content')
    <p>The animals of Swansea Zoo:</p>
    <ul>
        @foreach ($animals as $animal)
            <li><a href="{{ route('animals.show', ['id' => $animal->id]) }}">{{ $animal->name }}</a></li>
        @endforeach
    </ul>
@endsection
```

Page to show an individual animals



Route in routes/web.php:

```
Route::get('animals/{id}', 'AnimalController@show');
```

Controller: app/Http/Controllers/AnimalController.php

```
public function show($id)
{
    $animal = Animal::findOrFail($id);
    return view('animals.show', ['animal' => $animal]);
}
```

View: resources/views/animals/show

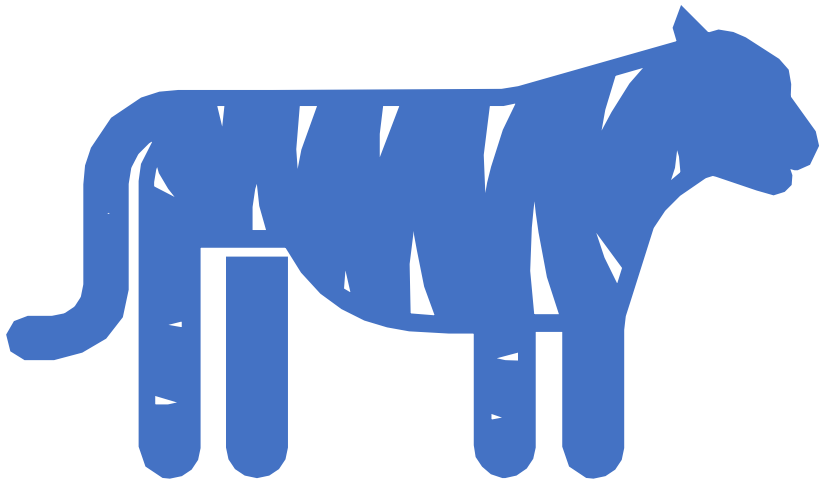
```
@extends('layouts.app')


@section('title', 'Animal Details')


@section('content')
    <ul>
        <li>Name: {{ $animal->name }}</li>
        <li>Weight: {{ $animal->weight }}kg</li>
        <li>Date of Birth: {{ $animal->date_of_birth ?? 'Unknown' }}</li>
        <li>Enclosure: {{ $animal->enclosure->name }}</li>
    </ul>
@endsection
```

Resources and Routes

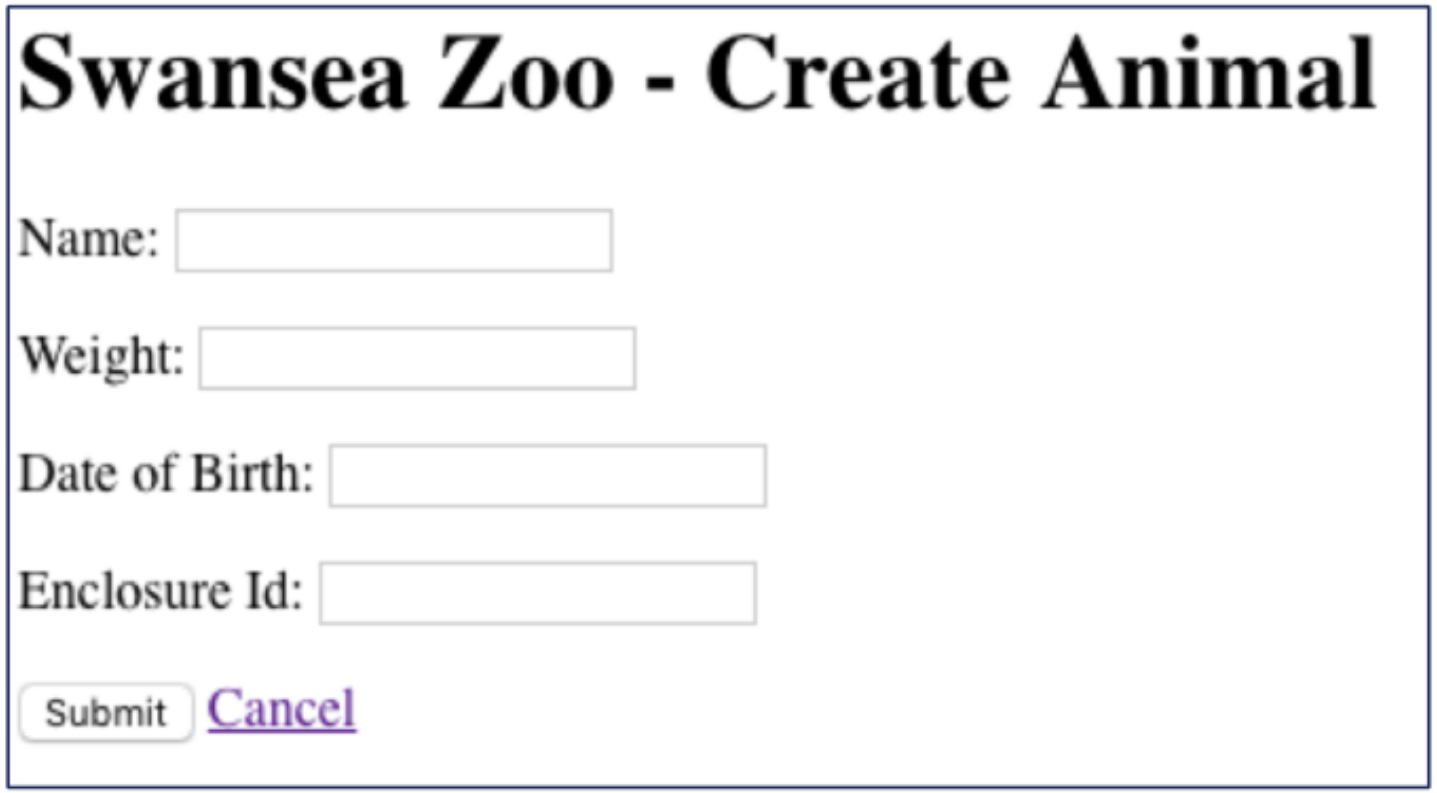
Laravel advises a structure on your routes for resources.

For example, for a Photos resource uses:

Verb	URI	Action	Route Name	Description
GET	/photos	index	photos.index	Get all photos
GET	/photos/create	create	photos.create	Display the page to create a new photo
POST	/photos	store	photos.store	Store a photo (create page will POST to here)
GET	/photos/{photo}	show	photos.show	Display a specified photo
GET	/photos/{photo}/edit	edit	photos.edit	Display the edit page for a specified photo.
PUT/PATCH	/photos/{photo}	update	photos.update	Update an existing photo (edit page will PUT/PATCH to here)
DELETE	/photos/{photo}	destroy	photos.destroy	Delete a photo



Creating Animals



Creating Animals

- This will involve creating a view with a form for the user to enter data
- Writing a controller which will handle that data and enter it into a database

The Routes

To create new animals we need to add the following routes:

Verb	URI	Action	Route Name	Description
GET	/animals/create	create	animals.create	Display the page to create a new animal
POST	/animals	store	animals.store	Store an animal (create page will POST to here)

Routes in routes/web.php:

```
Route::get('animals', 'AnimalController@index')->name('animals.index');
Route::get('animals/create', 'AnimalController@create')->name('animals.create');
Route::post('animals', 'AnimalController@store')->name('animals.store');
Route::get('animals/{id}', 'AnimalController@show')->name('animals.show');
```

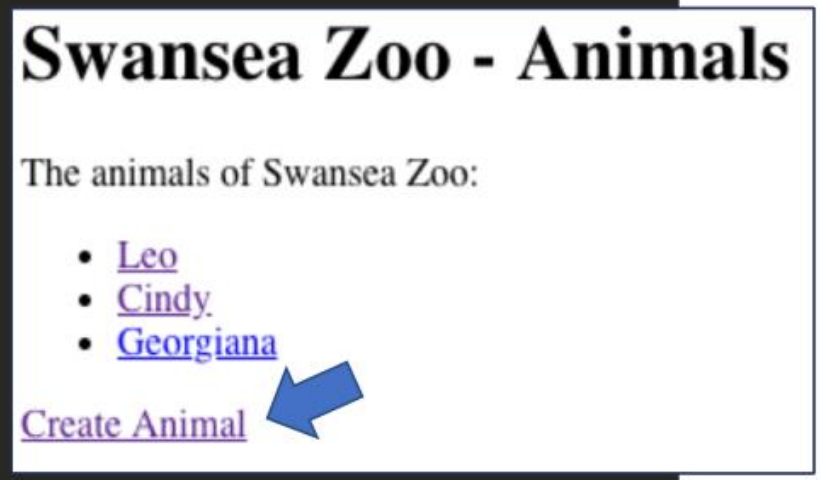
Note that animals.show must come last due to the parameter.

- If it came before animals.create then the /create would be captured by the parameter in animals.show

First: A Link To Get to the Create Page

We will add a link to get to the Create Page (animals.create) from our index page (animals.index)

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


@section('title', 'Animals')


@section('content')
    <p>The animals of Swansea Zoo:</p>
    <ul>
        @foreach ($animals as $animal)
            <li><a href="{{ route('animals.show', ['id' => $animal->id]) }}">
                {{ $animal->name }}</a></li>
        @endforeach
    </ul>
    <a href="{{ route('animals.create' )}}">Create Animal</a>
@endsection
```



The Create Route

- Remember each route will fire a controller action
- The create route simply needs to present the user with the form for entering data
- So all the controller method needs to do is return a view

Controller:

app/Http/Controllers/AnimalController.php

```php
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    return view('animals.create');
}
```

The Create Route (View)

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


@section('title', 'Create Animal')


@section('content')
    <form method="POST" action="{{ route('animals.store') }}">
        @csrf
        <p>Name: <input type="text" name="name" ></p>
        <p>Weight: <input type="text" name="weight"></p>
        <p>Date of Birth: <input type="text" name="date_of_birth"></p>
        <p>Enclosure Id: <input type="text" name="enclosure_id"></p>
        <input type="submit" value="Submit">
        <a href="{{ route('animals.index') }}">Cancel</a>
    </form>


@endsection
```

We use a form to POST the data to the receiving route.

The action attribute specifies the receiving URI.

Laravel protection against cross-site request forgery (CSRF) attacks. Ignore for now – but you need it

One input element for each attribute of our Animal.

Finally, a submit button and a cancel link.

The Create Route (View)

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


@section('title', 'Create Animal')


@section('content')
    <form method="POST" action="{{ route('animals.store') }}">
        @csrf
        <p>Name: <input type="text" name="name" ></p>
        <p>Weight: <input type="text" name="weight"></p>
        <p>Date of Birth: <input type="text" name="date_of_birth"></p>
        <p>Enclosure Id: <input type="text" name="enclosure_id"></p>
        <input type="submit" value="Submit">
        <a href="{{ route('animals.index') }}">Cancel</a>
    </form>


@endsection
```

When the user clicks submit, the data will be posted to the animals.store route (which will process the request and add the animal).

The Store Route

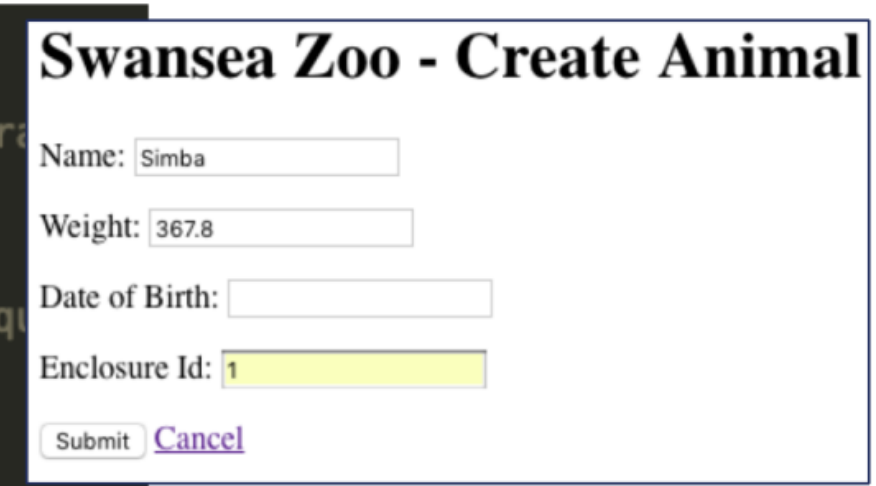
- When the user clicks submit a HTTP POST request will be sent to the animals.store route
- That request will include the form data
- The AnimalController.store method will process this request
  - It will validate it
  - If the validation fails, it will redirect back to the create page with error messages
  - If the validation passes, it will add the animal and redirect back to the animals.index page with a message saying adding the animal was successful
- …none of this needs a new view – it all uses existing views

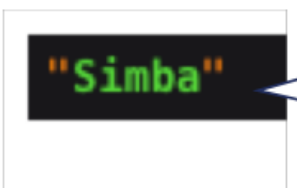
The Store Route Controller – Checking We Have Data

First, lets check we actual receive the data from the form.

Controller:
app/Http/Controllers/AnimalController.php

```php
/**
 * Store a newly created resource in stor
 *
 * @param  \Illuminate\Http\Request  $req
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    dd($request['name']);
}
```





We can access the form data. Yay ☺

The Store Route Controller – Validation

So now we can access the data from the HTTP request.

Next, we should validate it.

- Laravel has great validation tools built in.

- The $request parameter has a method validate that takes a list of rules and checks the post data against the rules.

```
$validatedData = $request->validate([
    'name' => 'required|max:255',
    'weight' => 'required|numeric',
    'date_of_birth' => 'nullable|date',
    'enclosure_id' => 'required|integer',
]);
```

Rules to validate POST data.

Just an associative array from field names to rules.

See next slide.

The Store Route Controller – Validation (Cont.)

```php
$validatedData = $request->validate([
    'name' => 'required|max:255',
    'weight' => 'required|numeric',
    'date_of_birth' => 'nullable|date',
    'enclosure_id' => 'required|integer',
]);
```

- name is required and must be at most 255 characters.
- weight is required and must be numeric (integer or decimal).
- date_of_birth can either be null or a PHP date (empty strings get turned into null).
- enclosure_id must be an integer.
- Find more rules at https://laravel.com/docs/validation

The Store Route Controller – Validation (Cont.)

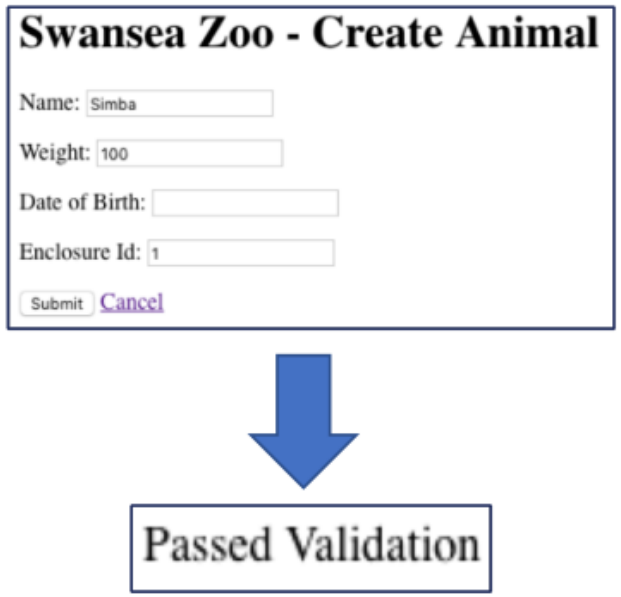
```
$validatedData = $request->validate([
    'name' => 'required|max:255',
```

- If the validation rules pass,
  - then the returned value ($validatedData) will containg only the validated data from the HTTP request (any other non-validated data will be filtered out).
  - We can then do something with the validated data.
    - E.g., Create the animal.

- If validation fails then the user will be redirected back to where they came from (the create page) with error messages (flashed to the session data – to discussed in a moment).

Testing the Validation – Valid Data

```php
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'weight' => 'required|numeric',
        'date_of_birth' => 'nullable|date',
        'enclosure_id' => 'required|integer',
    ]);

    return "Passed Validation";
}
```

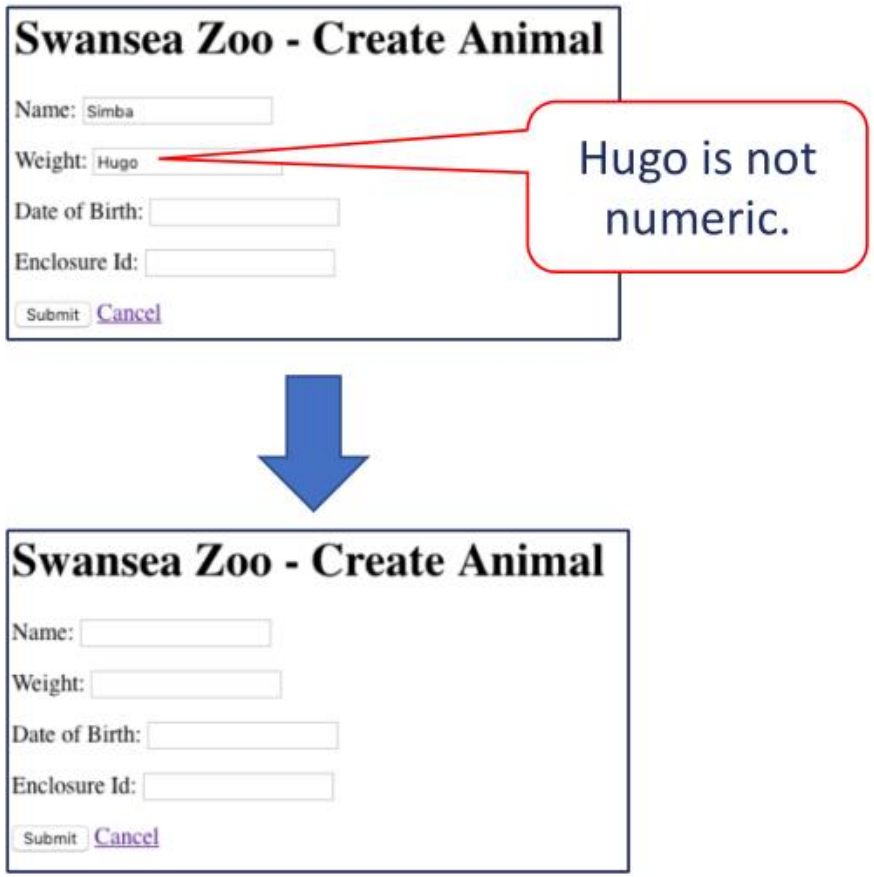


We enter valid data and we successfully execute the line that returns the string. We could now continue to write code to create a model and save it.

But first, let's test failing the validation.

Testing the Validation – Bad Data

```php
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'weight' => 'required|numeric',
        'date_of_birth' => 'nullable|date',
        'enclosure_id' => 'required|integer',
    ]);

    return "Passed Validation";
}
```



We enter invalid data.

The redirection worked – we went back to the create page.

But where are the error messages?

Answer: We have no code in our views to display them!

Displaying Error Messages

- Laravel automatically redirects the user back to the page they came from when validation fails
- Laravel also flashes error messages to the session data
  - Flashed data is only stored for exactly the next request, when the redirection happens, then the are cleared
- We need to add something to our view to display the error messages
- To avoid writing the same code multiple times we put this in the layout file

Displaying Error Messages (Cont.)

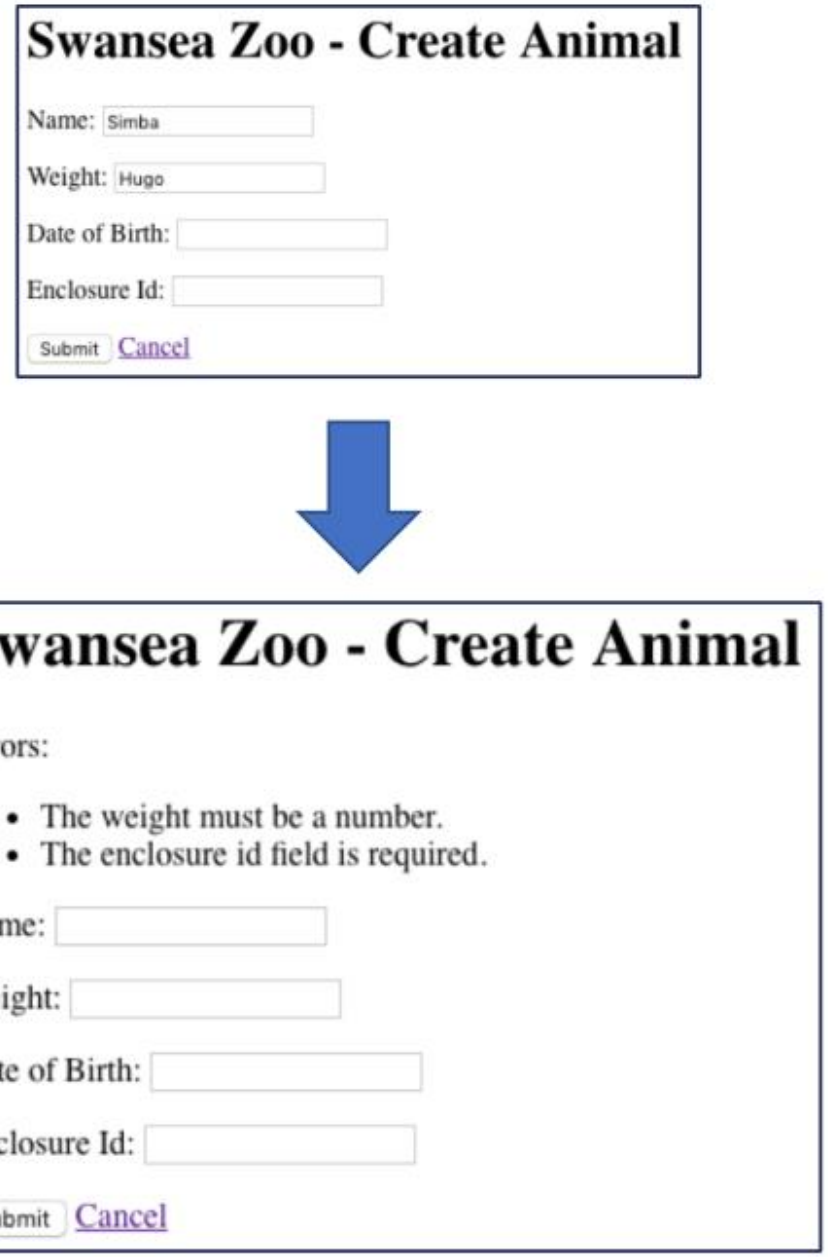
We change our layout file:

- $errors will hold the errors. The variable will always exist thanks to middleware.
- Here, we check if there are any errors.
  - If so we display them in a simple list.
- After the errors, we will display the content of the actual page.

resources/views/layouts/app.blade.php

```
@if ($errors->any())
    <div>
        Errors:
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif


<div>
  @yield('content')
</div>
```

- This time we get error messages
- Notice how Laravel removed the underscores of our property names and turned them into spaces!
- But we lost the data we already entered into the form…



Why Did We Lose The Data

- We entered invalid data.
- Data validation failed.
- We were redirected back to the create page.
- When we display the form:
  - We now display the flashed error messages.
  - We then just display a blank form.

- Luckily Laravel provide us with a helper function called old.
  - https://laravel.com/docs/helpers
  - The old function retrieves an old input values flashed into the session.
- Time to go update our create page ☺

Modifying the Create View to Use Old Helper

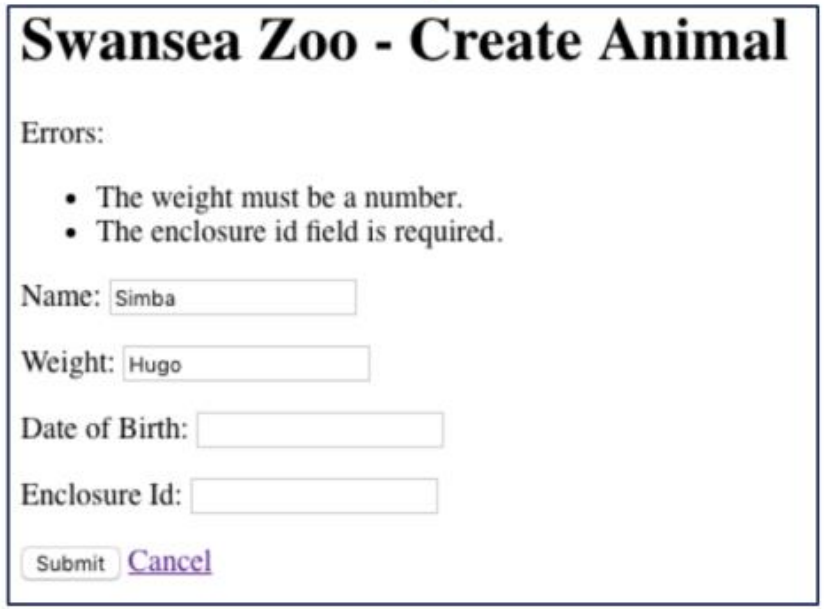
We set the default value of the fields to be the old input values.

When first displayed they will be blank (as the old function will return empty string).

```
@section('content')
    <form method="POST" action="{{ route('animals.store') }}">
        @csrf
        <p>Name: <input type="text" name="name"
            value="{{ old('name') }}"></p>
        <p>Weight: <input type="text" name="weight"
            value="{{ old('weight') }}"></p>
        <p>Date of Birth: <input type="text" name="date_of_birth"
            value="{{ old('date_of_birth') }}"></p>
        <p>Enclosure Id: <input type="text" name="enclosure_id"
            value="{{ old('enclosure_id') }}"></p>
        <input type="submit" value="Submit">
        <a href="{{ route('animals.index') }}">Cancel</a>
    </form>
@endsection
```

Testing the Validation – Bad Data – Attempt 3

```php
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'weight' => 'required|numeric',
        'date_of_birth' => 'nullable|date',
        'enclosure_id' => 'required|integer',
    ]);

    return "Passed Validation";
}
```



- This time we get error messages and old values.
- Great let's move on to actually saving the animal.

Saving the Animal in the Database

- This works exactly the same way as with seeding
- We create the animal model, put the data in it, then save to the database
- At the end we ‘flash’ a message and redirect the user to the index page.

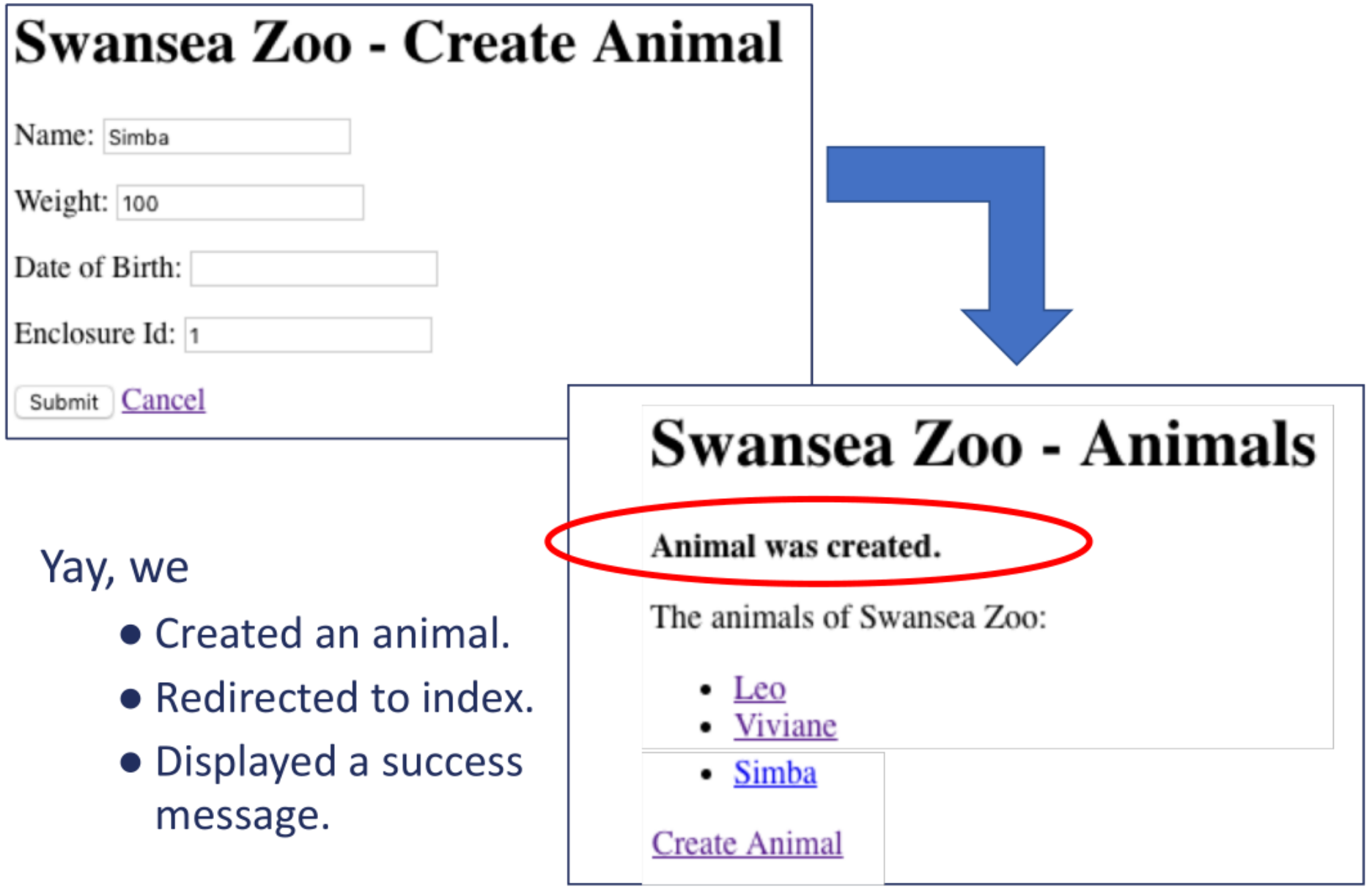
```php
$a = new Animal;

$a->name = $validatedData['name'];

$a->weight = $validatedData['weight'];

$a->date_of_birth = $validatedData['date_of_birth'];

$a->enclosure_id = $validatedData['enclosure_id'];

$a->save();



session()->flash('message', 'Animal was created.');

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

Displaying flashed messages

- Much like with errors it makes sense to put the flashed message code in the layout view, so we can just use it without much thought in our controllers

```
@if (session('message'))
    <p><b>{{ session('message') }}</b></p>
@endif
```

Final Result



Yay, we

- Created an animal.
- Redirected to index.
- Displayed a success message.

Tasks

1. Continuing with the application you have been creating. Pick one of your models and create the controllers and views required to allow the user to create these models.
2. Add data validation to your web application