HTML Selects, Deleting Models, and Route Model Binding

…all the stuff which doesn’t really fit anywhere else!

Swansea Zoo - Create Animal

Name: Simba

Weight: 100

Date of Birth:

Enclosure Id: 1

Submit Cancel

Swansea Zoo - Animals

Animal was created.

The animals of Swansea Zoo:

- Leo
- Viviane
- Simba

Create Animal

Yay, we

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

A recap and plan of things to come…

- At this stage you have seen the overview of how the framework works
- Moving forward there are just bits and pieces to help you do ‘things’
- In reality these are the kinds of things you would just google, the idea being now you understand the basics you can just google stuff – for example “How to setup authentication in Laravel”
- I’m probably not going to do any more live coding because at this stage you need to try and swim yourself!

Bits and Pieces of this Lecture

Setting up HTML select inputs, for when we want the user to pick from a number of valid options

The D of CRUD

Route Model Binding

HTML Select Inputs

Create Form - Problems

- Our current create form is okay.
- But you have to magically know the enclosure ID.
- Not very user friendly.
- Lets turn it into this.

How do we achieve this?

- The create page needs to know about all the enclosures to display them as options
- We can’t hardcode this because the number of enclosures or names might change
- We could write some code to find a list of enclosures in the view, because the blade templating system allows us to, but we should because the view shouldn’t be doing any business logic
- So we do it in the controller and pass the data to the view

Adapting the Controller

Controller: app/Http/Controllers/AnimalController.php

```
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    $enclosures = Enclosure::all();
    return view('animals.create', ['enclosures' => $enclosures]);
}
```

All we need to do is to grab the enclosures and pass them to the view.

Adapting the View

View: resources/views/animals/create.blade.php

```
<form method="POST" action="{{ route('animals.store') }}">
    @csrf
    <p>Name: <input type="text" name="name"
        value="{{ old('name') }}"></p>
..
        <select name="enclosure_id">
            @foreach ($enclosures as $enclosure)
                <option value="{{ $enclosure->id }}">
                    {{ $enclosure->name }}
                </option>
            @endforeach
        </select>
    </p>

    <input type="submit" value="Submit">
    <a href="{{ route('animals.index') }}">Cancel</a>
</form>
```

We replace the text input with a select input.

Its name still matches the model's field.

We then loop over each enclosure and add it as an option to the select input.

Each option uses the enclosures id as its value and its name as the option's text.

The text is shown on the webpage, but the value is used when the form is submitted,

- So now we have a nice select drop down.
- Not too hard at all.
- But the list of enclosures is not in order.
  - We could change the database to add the enclosures in alphabetical order. However, the enclosures will get out of order as the database is manipulated.
  - We need to sort the data.
    - The view could do this …
    - But it is better placed in the controller.
    - Views should only present data
      - and possibly use a little presentation logic.

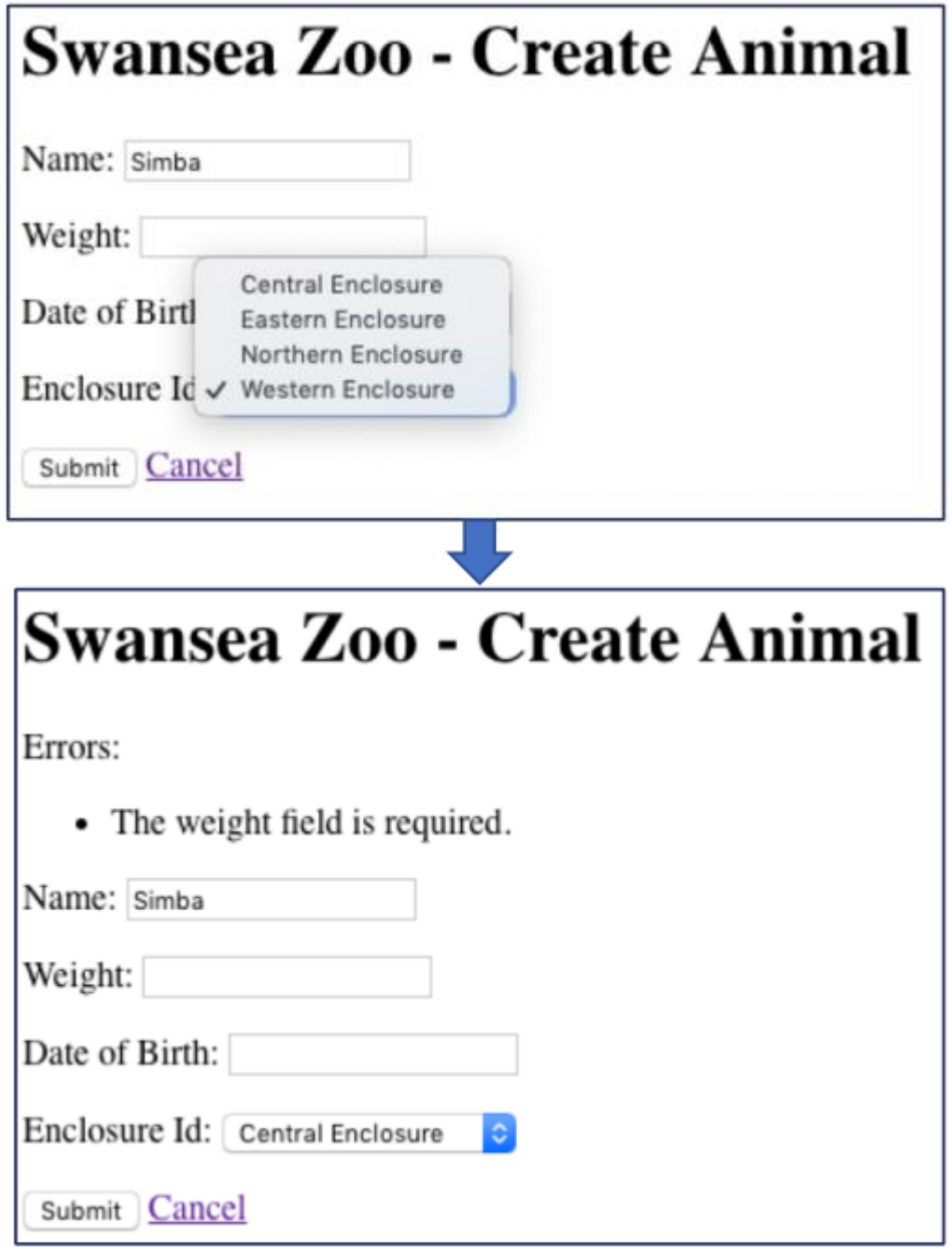
Sorting the Enclosures

We want the controller to pass a sorted list of enclosures to the view.

```
public function create()
{
    // $enclosures = Enclosure::all();
    $enclosures = Enclosure::orderBy('name', 'asc')->get();
    return view('animals.create', ['enclosures' => $enclosures]);
}
```

- The method all() returns a Laravel Collection.
  - We could sort the collection, but that could be slow.
  - Better to make the database do it.
- We use Laravel's Query Builder to get all the enclosures but already sorted by the database engine.

- Now we have an sorted dropdown menu.
- Have we missed anything?
- We forgot to preserve the selected enclosure if the validation fails – we didn't use the old() helper…



View: resources/views/animals/create.blade.php

Select uses a selected attribute to indicate which option should be pre-selected.

```
<select name="enclosure_id">
    @foreach ($enclosures as $enclosure)
        <option value="{{ $enclosure->id }}"
            @if ($enclosure->id == old('enclosure_id'))
                selected="selected"
            @endif
        >{{ $enclosure->name }}</option>
    @endforeach
</select>
```

We only want to add the selected attribute to the option which has the same enclosure id as was entered previously (and flashed to the session data).

Thus, we use a Blade Conditional Statement to check.

Notice the closing tag of the option element is down here.

Swansea Zoo - Create Animal

Name: Simba

Weight:

Date of Birt

Enclosure Id

- Central Enclosure
- Eastern Enclosure
- Northern Enclosure
- ✓ Western Enclosure

Submit Cancel

Swansea Zoo - Create Animal

Errors:

- The weight field is required.

Name: Simba

Weight:

Date of Birth:

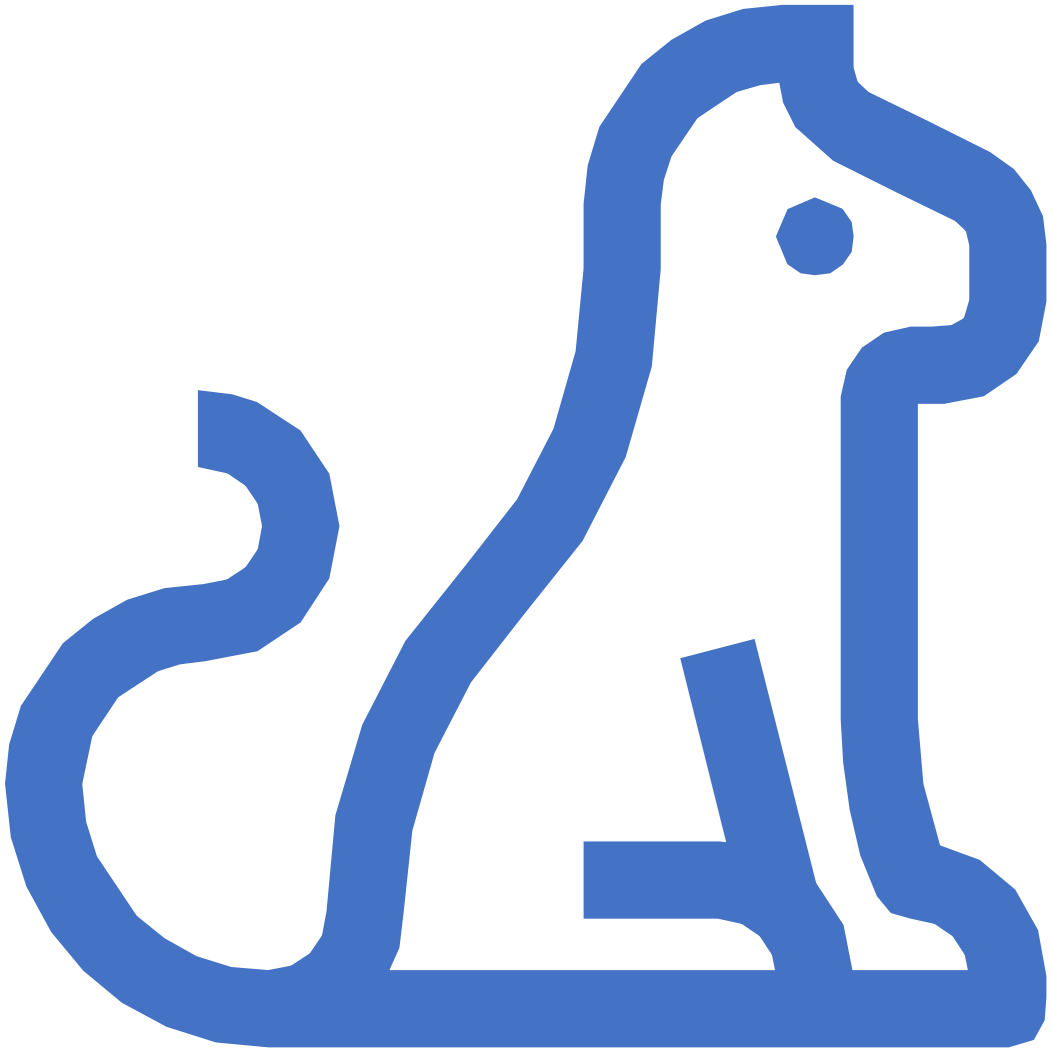
Enclosure Id: Western Enclosure

Submit Cancel

Reflection

- None of these steps are particularly difficult
- But it does take time to implement and test
- Just build things up gradually, split the problem into small testable chunks.
- Essentially I'm recommending that you adopt a rapid prototyping approach

Deleting Animals



The Routes

Verb	URI	Action	Route Name	Description
DELETE	/animals/{id}	destroy	animals.destroy	Delete an animal.

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');
Route::delete('animals/{id}', 'AnimalController@destroy')->name('animals.destroy');
```

Note that the routes with parameters must come last.

- If they came before animals.create then the /create would be captured by the parameters.

Controller

- The controller method is dead simple:
  - Get the model.
  - Delete it.
  - Redirect with a success message.

```
public function destroy($id)
{
    $animal = Animal::findOrFail($id);
    $animal->delete();

    // session()->flash('message', 'Animal was deleted.');
    return redirect()->route('animals.index')->with('message', 'Animal was
        deleted.');
}
```

Flashing data is so common that there is a short hand using the with method on a redirect.

The with “addon” can replace the session->flash line.

HTTP Methods / Verbs and Forms

- HTTP has several methods (also known as verbs):
  - GET
  - POST
  - PUT/PATCH
  - DELETE
  - etc.
- So how do we use DELETE:
  - HTML hyperlinks use only GET.
  - HTML forms only support GET and POST.
    - They seem to have been considered for HTML5, but were dropped.
  - So if HTML forms don't support them.
    - We spoof them using forms – it's a little ugly!

HTML Forms and HTTP DELETE

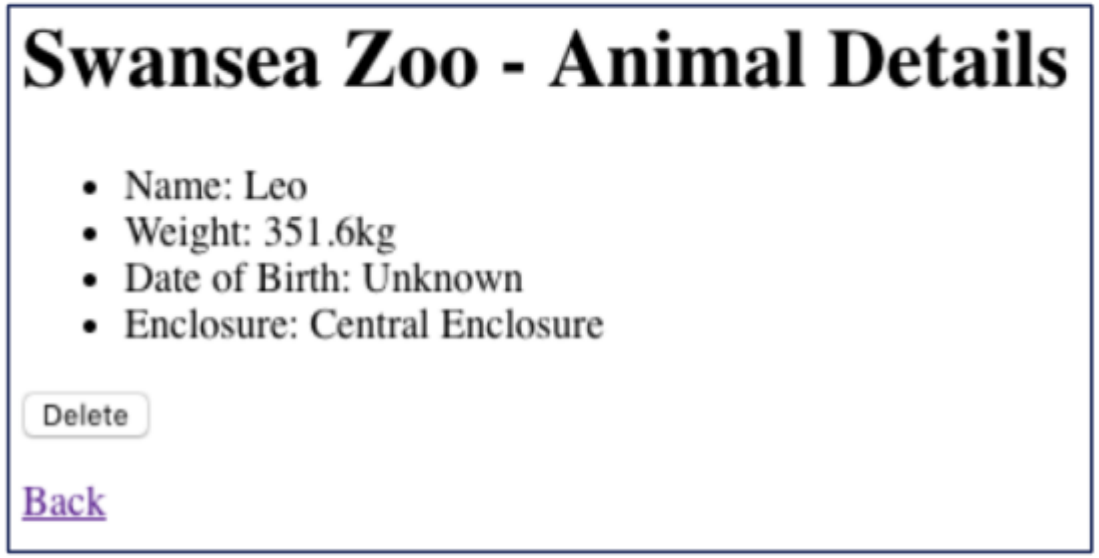
- This is dependent on the framework you are using, but they are all similar.
- In Laravel you can use a hidden form field named _method to specify the HTTP method that you want Laravel to use for routing.
- The form will actually use POST.
- But when Laravel receives it and notices the hidden field it will internally treat it as if the HTTP DELETE method was used.
- This is known as spoofing. You pretend the request is something else.

Back to animals.show View

We will add the delete button to the show page that displays details about an animal.

We will use a form and change the submit button's text to Delete.

You can style this to look consistent with CSS.



Back to animals.show View

Make a form that POSTS to the destroy route.

Remember we need to provide the ID for the route.

The @method Blade construct allows us to easily provide the hidden field to spoof the HTTP method.

```
@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>

    <form method="POST"
        action="{{ route('animals.destroy', ['id' => $animal->id]) }}">
        @csrf
        @method('DELETE')
        <button type="submit">Delete</button>
    </form>

    <p><a href="{{ route('animals.index') }}">Back</a></p>
@endsection
```

The Produced Form

```
<form method="POST" action="http://swanseazoo.test/animals/2">
  <input type="hidden" name="_token" value="yghXMD1MRS77fsQ2NPB6yxvJnfiFLqGag9yUO1QS">
  <input type="hidden" name="_method" value="DELETE">
  <button type="submit">Delete</button>
</form>
```

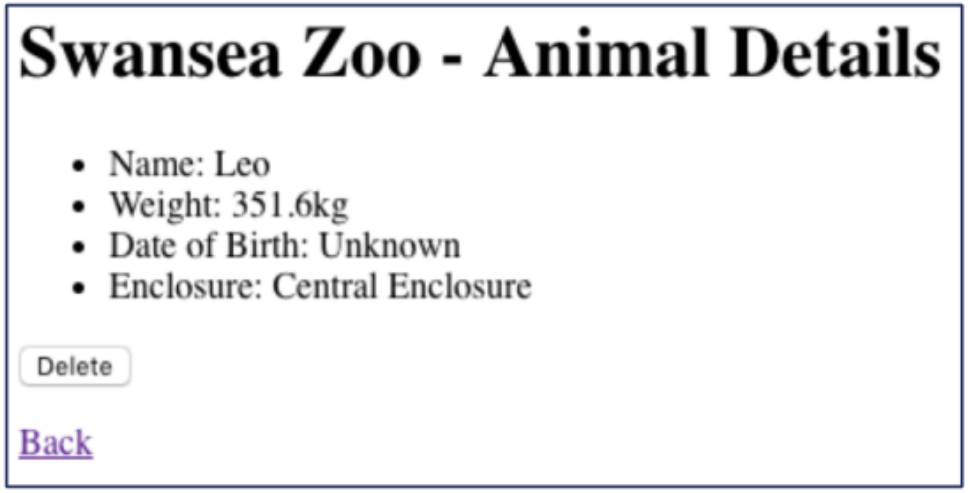
Here you can see the hidden form fields that was produced to spoof the HTTP DELETE method.

- You can also see the result of the @csrf blade construct too. More on this in a later lecture.

Checking Delete Works

So now when we click the delete button:

- We POST to animals.destroy route.
- But the _method hidden field allows Laravel to treat the HTTP request as a DELETE.
- It matches the destroy route.
- The controller deletes the model and redirects us with a nice message.
  - We didn't change the show page as we already display the flashed message data.



Route Model Binding

Let's Take a Look Back at Our Controller

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

Repeated Line

```php
public function destroy($id)
{
    $animal = Animal::findOrFail($id);
    $animal->delete();

    // session()->flash('message', 'Animal was deleted.');
    return redirect()->route('animals.index')->with('message', 'Animal was
        deleted.');
}
```

Repeated Line

Let's Take a Look Back at Our Controller (Cont.)

```
public function update(Request $request, $id)
{
    //
}
```

```
public function edit($id)
{
    //
}
```

It even looks like the update and edit methods might end up having the same line.

- We haven't looked at these yet.
- But, the id of a model is passed in so we will probably have to go and find it.

Route Model Binding

- Route Model Binding is a technique in web frameworks whereby a route can automatically be bound to a model
- Specifically when a route has a parameter, Laravel can automatically accept and id, find the corresponding model, then pass that model the controller
  - This is called injecting the model
  - The controller will receive the model as a parameter and not the id – so we don’t have to write code to get the model from the database

Implicit Route Model Binding

Step 1: Update your route file:

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

```
Route::get('animals/{animal}', 'AnimalController@show')->name('animals.show');
```

We change the “variable” name from “id” to “animal” to match the lowercased (and snake cased) version of the class name for the model.

- This is not strictly necessary, but it is good practice.
- But it must match the parameter name in the controller in the next step.

Implicit Route Model Binding (Cont).

Step 2: Update your controller:

This is called Type Hinting. We specify the type we expect and Laravel recognises this and injects the corresponding model with the matching id.

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

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

You still work with ids when building routes with the Blade files. Ids are still used in the actual URIs.

But now the controllers are passed models directly instead of just the id.

Implicit Route Model Binding – Can’t Find Model

If Laravel can’t find a corresponding model with a matching id then a 404 HTTP response is automatically generated.

Tasks

If you haven’t already – start and do your coursework!