**Route Animation**

1. In app.module.ts add BrowserAnimationsModule to the module imports.

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';@NgModule({

imports: [

BrowserAnimationsModule

],

})

export class AppModule {}

1. We will animate views left or right based on the direction the router is navigating. We’ll have three components named HomeComponent, ContactComponent, and AboutComponent. When navigating from Home to Contact, Home will slide out to the left while Contact will slide in from the right. Contact to About will do the same. When navigating from About to Contact, the animations will be reversed. The opacity of the views will be animated as they leave and enter the page.
2. State is a static style definition. A transition defines how a property in the style will change. A trigger defines what action will cause one state to transition to another state.
3. To connect animations to the router, add a data property to the route configuration. Here are our modified routes:

const routes: Routes = [

{

path: '',

children: [

{

path: 'home',

component: HomeComponent,

data: { animationState: 'Home' }

},

{

path: 'contact',

component: ContactComponent,

data: { animationState: 'Contact' }

},

{

path: 'about',

component: AboutComponent,

data: { animationState: 'About' }

},

{

path: '\*\*',

redirectTo: 'home'

}

]

},

{

path: '\*\*',

redirectTo: 'home'

}

];

1. Configure the AppComponent to set up the animations for the route changes. In app.component.ts add a method:

prepareRoute(outlet: RouterOutlet) {

return outlet &&

outlet.activatedRouteData &&

outlet.activatedRouteData['animationState'];

}

1. Check for a route with data for the state specified property, animationState. Then hook up the template. Add a template variable so that we can get a reference to the <router-outlet>.
2. <router-outlet #outlet="outlet"></router-outlet>
3. Next, add a synthetic property to the container element of the <router-outlet>. It needs to be on a container div, not on the <router-outlet> itself. This synthetic property’s name is arbitrary, but it’s good to understand that it will correspond to an animation trigger’s name. Let’s call it triggerName.

<div [@triggerName]="prepareRoute(outlet)">

<router-outlet #outlet="outlet"></router-outlet>

</div>

1. We pass the method prepareRoute with the argument of the template variable outlet to the synthetic property @triggerName.
2. At this point, if you run the application, you’ll find that there is an error in the console:
3. ERROR Error: Found the synthetic property @triggerName. Please include either "BrowserAnimationsModule" or "NoopAnimationsModule" in your application.
4. An animation is caused by a trigger that causes a transition from one state to another state. When we define an animation we start with the trigger and work inward on that definition.
5. Create a new file named route-transition-animations.ts next to app.component.ts. This will contain the trigger definition, triggerName, and the transitions from and to the states we wish to animate.

import { trigger } from '@angular/animations';export const routeTransitionAnimations = trigger('triggerName', []);

1. Here we finally define the trigger triggerName. The array argument is where we will define the transitions.
2. Before we define the transitions, let’s hook the app.component.ts to the trigger definition:

import { routeTransitionAnimations } from './route-transition-animations';@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.scss'],

animations: [routeTransitionAnimations]

})

export class AppComponent {...}

1. Now, let’s go back and flesh out the trigger’s transitions in the route-transition-animations.ts.
2. Angular uses simple arrow syntax to define the transition from one state to another. For example, if we want to handle the navigation from Home to Contact we use Home => Contact. If we want to handle both directions, we can use a bi-directional arrow, Home <=> Contact, and then the transition will be applied going from Home to Contact and from Contact to Home.
3. Angular has pre-defined concepts in addition to the named states.

void = an element is entering or leaving the view.

\* = any state

:enter and :leave are aliases for the void => \* and \* => void transitions.

1. Home => Contact and Contact => About should slide the previous view off to the left and bring the new view in from the right. Since they both have the same transition, both state changes can be defined in a single transition using comma separated values:

import { trigger, transition } from '@angular/animations';

export const routeTransitionAnimations = trigger('triggerName', [

transition('Home => Contact, Contact => About', [])

]);

1. According to official Angular documentation: during a transition, a new view is inserted directly after the old one and both elements appear on screen at the same time. To prevent this, apply additional styling to the host view, and to the removed and inserted child views. The host view must use relative positioning, and the child views must use absolute positioning. Adding styling to the views animates the containers in place, without the DOM moving things around.
2. Apply this to the style definition by adding the following:

import { trigger, transition, style, query } from '@angular/animations';export const routeTransitionAnimations = trigger('triggerName', [

transition('Home => Contact, Contact => About', [

style({ position: 'relative' }),

query(':enter, :leave', [

style({

position: 'absolute',

top: 0,

right: 0,

width: '100%'

})

])

])

]);

1. First, style({ position: ‘relative’ }) sets the style on the element that is the target of the trigger to be position: relative. The target element is the one with the synthetic property @triggerName, which is the div that contains the <router-outlet>. Now, the “host view” is using relative positioning per the official docs.
2. Next, query(':enter, :leave', [...]). This means “query for child elements that are entering or leaving the view.” Then it applies the following style definition to those elements. I won’t dive too much into the CSS solution for the positions, but the real key is that we are setting the child elements to absolute positioning, per the official docs. Your CSS will almost certainly differ at this point based on your chosen animation style and application DOM layout.
3. Let’s define the individual transitions, in order. These will follow the first query in the transition's array arguments.
4. This query defines what the start state is for the view that is entering, positioning it off screen to the far right:

query(':enter', [style({ right: '-100%', opacity: 0 })]),

1. The next query ensures that any child component animations that need to happen on the leaving component happen before the leaving view animates off screen:

query(':leave', animateChild()),

1. Next, we group the leave and enter together so that these transitions happen in unison (otherwise, the old would leave, leaving a blank space, and then the new would enter). We animate, meaning “transition existing styles to the specified styles over a period of time with an easing function.” The leaving view animates its right value to be 100% (the far left of the screen) and the entering animate’s its right value to be 0% (the far right of the screen):

group([

query(':leave', [animate('1s ease-out', style({ right: '100%', opacity: 0 }))]),

query(':enter', [animate('1s ease-out', style({ right: '0%', opacity: 1 }))])

]),

1. At this point, the old view has left, the new one has entered, and we want to trigger any child animations on the new view:

query(':enter', animateChild())

1. Add the transition for the reverse direction, About => Contact, and Contact => Home, after the first transition, and change the rights to lefts:

transition('About => Contact, Contact => Home', [

style({ position: 'relative' }),

query(':enter, :leave', [

style({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

query(':enter', [style({ left: '-100%', opacity: 0 })]),

query(':leave', animateChild()),

group([

query(':leave', [animate('1s ease-out', style({ left: '100%', opacity: 0 }))]),

query(':enter', [animate('1s ease-out', style({ left: '0%', opacity: 1 }))])

]),

query(':enter', animateChild())

])

1. Let’s add two transition definitions, Home => About, and About => Home. Rather than defining something different, we will add these to the existing ones. Add Home => About to the right definition, and the About => Home to the left. The transitions now look like this:

transition('Home => Contact, Contact => About, Home => About', [...]),

transition(' About => Contact, Contact => Home, About => Home', [...])