Route transition animations

Prerequisites A basic understanding of the following concepts:

- Introduction to Angular animations
- Transition and triggers
- Reusable animations

Routing enables users to navigate between different routes in an application. When a user navigates from one route to another, the Angular router maps the URL path to a relevant component and displays its view. Animating this route transition can greatly enhance the user experience.

The Angular router comes with high-level animation functions that let you animate the transitions between views when a route changes. To produce an animation sequence when switching between routes, you need to define nested animation sequences. Start with the top-level component that hosts the view, and nest additional animations in the components that host the embedded views.

To enable routing transition animation, do the following:

1. Import the routing module into the application and create a routing configuration that defines the possible routes.
2. Add a router outlet to tell the Angular router where to place the activated components in the DOM.
3. Define the animation.

Illustrate a router transition animation by navigating between two routes, *Home* and *About* associated with the `HomeComponent` and `AboutComponent` views respectively. Both of these component views are children of the top-most view, hosted by `AppComponent`. We'll implement a router transition animation that slides in the new view to the right and slides out the old view when the user navigates between the two routes.

Route configuration

To begin, configure a set of routes using methods available in the `RouterModule` class. This route configuration tells the router how to navigate.

Use the `RouterModule.forRoot` method to define a set of routes. Also, add `RouterModule` to the `imports` array of the main module, `AppModule`.

Note: Use the `RouterModule.forRoot` method in the root module, `AppModule`, to register top-level application routes and providers. For feature modules, call the `RouterModule.forChild` method instead.

The following configuration defines the possible routes for the application.

The `home` and `about` paths are associated with the `HomeComponent` and `AboutComponent` views. The route configuration tells the Angular router to

instantiate the `HomeComponent` and `AboutComponent` views when the navigation matches the corresponding path.

In addition to `path` and `component`, the `data` property of each route defines the key animation-specific configuration associated with a route. The `data` property value is passed into `AppComponent` when the route changes. You can also pass additional data in route configuration that is consumed within the animation. The data property value has to match the transitions defined in the `routeAnimation` trigger, which we'll define shortly.

Note: The `data` property names that you use can be arbitrary. For example, the name *animation* used in the preceding example is an arbitrary choice.

Router outlet

After configuring the routes, add a `<router-outlet>` inside the root `AppComponent` template. The `<router-outlet>` directive tells the Angular router where to render the views when matched with a route.

The `ChildrenOutletContexts` holds information about outlets and activated routes. We can use the `data` property of each `Route` to animate our routing transitions.

`AppComponent` defines a method that can detect when a view changes. The method assigns an animation state value to the animation trigger (`@routeAnimation`) based on the route configuration `data` property value. Here's an example of an `AppComponent` method that detects when a route change happens.

Here, the `getRouteAnimationData()` method takes the value of the outlet and returns a string which represents the state of the animation based on the custom data of the current active route. Use this data to control which transition to execute for each route.

Animation definition

Animations can be defined directly inside your components. For this example you are defining the animations in a separate file, which lets us re-use the animations.

The following code snippet defines a reusable animation named `slideInAnimation`.

The animation definition performs the following tasks:

- Defines two transitions (a single `trigger` can define multiple states and transitions).
- Adjusts the styles of the host and child views to control their relative positions during the transition.
- Uses `query()` to determine which child view is entering and which is leaving the host view.

A route change activates the animation trigger, and a transition matching the state change is applied.

Note: The transition states must match the `data` property value defined in the route configuration.

Make the animation definition available in your application by adding the reusable animation (`slideInAnimation`) to the `animations` metadata of the `AppComponent`.

So, let's break down the animation definition and see more closely what it does...

Styling the host and child components

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 behavior, update the host view to use relative positioning. Then, update the removed and inserted child views to use absolute positioning. Adding these styles to the views animates the containers in place and prevents one view from affecting the position of the other on the page.

Querying the view containers

Use the `query()` method to find and animate elements within the current host component. The `query(":enter")` statement returns the view that is being inserted, and `query(":leave")` returns the view that is being removed.

Assume that you are routing from the *Home => About*.

The animation code does the following after styling the views:

- `query(':enter', style({ left: '-100%' }))` matches the view that is added and hides the newly added view by positioning it to the far left.
- Calls `animateChild()` on the view that is leaving, to run its child animations.
- Uses group() function to make the inner animations run in parallel.
- Within the group() function:
  - Queries the view that is removed and animates it to slide far to the right.
  - Slides in the new view by animating the view with an easing function and duration. This animation results in the `about` view sliding in from the left.
- Calls the `animateChild()` method on the new view to run its child animations after the main animation completes.

You now have a basic routable animation that animates routing from one view to another.

More on Angular animations

You might also be interested in the following:

- Introduction to Angular animations
- Transition and triggers
- Complex animation sequences
- Reusable animations