Navigators Stack

Version: 6.x

Stack Navigator

Stack Navigator provides a way for your app to transition between screens where each new screen is placed on top of a stack.

By default the stack navigator is configured to have the familiar iOS and Android look & feel: new screens slide in from the right on iOS, use OS default animation on Android. But the animations can be customized to match your needs.

One thing to keep in mind is that while `@react-navigation/stack` is extremely customizable, it's implemented in JavaScript. While it runs animations and gestures using natively, the performance may not be as fast as a native implementation. This may not be an issue for a lot of apps, but if you're experiencing performance issues during navigation, consider using `@react-navigation/native-stack` instead - which uses native navigation primitives.

Installation

To use this navigator, ensure that you have `@react-navigation/native` and its dependencies (follow this guide), then install `@react-navigation/stack`:

npm Yarn

```
npm install @react-navigation/stack
```

Then, you need to install and configure the libraries that are required by the stack navigator:

1. First, install `react-native-gesture-handler`.

   If you have a Expo managed project, in your project directory, run:

   ```
   npx expo install react-native-gesture-handler
   ```

   If you have a bare React Native project, in your project directory, run:

   npm Yarn

   ```
   npm install react-native-gesture-handler
   ```

2. To finalize installation of `react-native-gesture-handler`, add the following at the **top** (make sure it's at the top and there's nothing else before it) of your entry file, such as `index.js` or `App.js`:

   ```
   import 'react-native-gesture-handler';
   ```

> Note: If you are building for Android or iOS, do not skip this step, or your app may crash in production even if it works fine in development. This is not applicable to other platforms.

3. Optionally, you can also install `@react-native-masked-view/masked-view`. This is needed if you want to use UIKit style animations for the header (`HeaderStyleInterpolators.forUIKit`).

   If you have a Expo managed project, in your project directory, run:

   ```
   npx expo install @react-native-masked-view/masked-view
   ```

   If you have a bare React Native project, in your project directory, run:

   npm Yarn

   ```
   npm install @react-native-masked-view/masked-view
   ```

4. If you're on a Mac and developing for iOS, you also need to install the pods (via Cocoapods) to complete the linking.

   ```
   npx pod-install ios
   ```

API Definition

To use this navigator, import it from `@react-navigation/stack`:

```
import { createStackNavigator } from '@react-navigation/stack';

const Stack = createStackNavigator();

function MyStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Home" component={Home} />
      <Stack.Screen name="Notifications" component={Notifications} />
```

```
      <Stack.Screen name="Profile" component={Profile} />
      <Stack.Screen name="Settings" component={Settings} />
    </Stack.Navigator>
  );
}
```

Try this example on Snack

Props

The `Stack.Navigator` component accepts following props:

`id`

Optional unique ID for the navigator. This can be used with `navigation.getParent` to refer to this navigator in a child navigator.

`initialRouteName`

The name of the route to render on first load of the navigator.

`screenOptions`

Default options to use for the screens in the navigator.

`detachInactiveScreens`

Boolean used to indicate whether inactive screens should be detached from the view hierarchy to save memory. This enables integration with react-native-screens. Defaults to `true`.

If you need to disable this optimization for specific screens (e.g. you want to screen to stay in view even when unfocused) `detachPreviousScreen` option.

`keyboardHandlingEnabled`

If `false`, the keyboard will NOT automatically dismiss when navigating to a new screen from this screen. Defaults to `true`.

Options

The following options can be used to configure the screens in the navigator. These can be specified under `screenOptions` prop of `Stack.navigator` or `options` prop of `Stack.Screen`.

`title`

String that can be used as a fallback for `headerTitle`.

`cardShadowEnabled`

Use this prop to have visible shadows during transitions. Defaults to `true`.

`cardOverlayEnabled`

Use this prop to have a semi-transparent dark overlay visible under the card during transitions. Defaults to `true` on Android and `false` on iOS.

`cardOverlay`

Function which returns a React Element to display as the overlay for the card. Make sure to set `cardOverlayEnabled` to `true` when using this.

`cardStyle`

Style object for the card in stack. You can provide a custom background color to use instead of the default background here.

You can also specify `{ backgroundColor: 'transparent' }` to make the previous screen visible underneath (for transparent modals). This is useful to implement things like modal dialogs. You should also specify `presentation: 'modal'` in the options when using a transparent background so previous screens aren't detached and stay visible underneath.

On Web, the height of the screen isn't limited to the height of the viewport. This is by design to allow the browser's address bar to hide when scrolling. If this isn't desirable behavior, you can set `cardStyle` to `{ flex: 1 }` to force the screen to fill the viewport.

`presentation`

This is shortcut option which configures several options to configure the style for rendering and transitions:

- `card`: Use the default OS animations for iOS and Android screen transitions.
- `modal`: Use Modal animations. This changes a few things:
  - Sets `headerMode` to `screen` for the screen unless specified otherwise.
  - Changes the screen animation to match the platform behavior for modals.
- `transparentModal`: Similar to `modal`. This changes following things:
  - Sets `headerMode` to `screen` for the screen unless specified otherwise.
  - Sets background color of the screen to transparent, so previous screen is visible
  - Adjusts the `detachPreviousScreen` option so that the previous screen stays rendered.
  - Prevents the previous screen from animating from its last position.
  - Changes the screen animation to a vertical slide animation.

See Transparent modals for more details on how to customize `transparentModal`.

`animationEnabled`

Whether transition animation should be enabled on the screen. If you set it to `false`, the screen won't animate when pushing or popping. Defaults to `true` on iOS and Android, `false` on Web.

`animationTypeForReplace`

The type of animation to use when this screen replaces another screen. It takes the following values:

- `push` - The animation of a new screen being pushed will be used
- `pop` - The animation of a screen being popped will be used

Defaults to `push`.

When `pop` is used, the `pop` animation is applied to the screen being replaced.

`gestureEnabled`

Whether you can use gestures to dismiss this screen. Defaults to `true` on iOS, `false` on Android.

Gestures are not supported on Web.

`gestureResponseDistance`

Number to override the distance of touch start from the edge of the screen to recognize gestures.

It'll configure either the horizontal or vertical distance based on the `gestureDirection` value.

The default values are:

- `50` - when `gestureDirection` is `horizontal` or `horizontal-inverted`
- `135` - when `gestureDirection` is `vertical` or `vertical-inverted`

This is not supported on Web.

`gestureVelocityImpact`

Number which determines the relevance of velocity for the gesture. Defaults to 0.3.

This is not supported on Web.

`gestureDirection`

Direction of the gestures. Refer the Animations section for details.

This is not supported on Web.

`transitionSpec`

Configuration object for the screen transition. Refer the Animations section for details.

`cardStyleInterpolator`

Interpolated styles for various parts of the card. Refer the Animations section for details.

`headerStyleInterpolator`

Interpolated styles for various parts of the header. Refer the Animations section for details.

`detachPreviousScreen`

Boolean used to indicate whether to detach the previous screen from the view hierarchy to save memory. Set it to `false` if you need the previous screen to be seen through the active screen. Only applicable if `detachInactiveScreens` isn't set to `false`.

This is automatically adjusted when using `presentation` as `transparentModal` or `modal` to keep the required screens visible. Defaults to `true` in other cases.

`freezeOnBlur`

Boolean indicating whether to prevent inactive screens from re-rendering. Defaults to `false`. Defaults to `true` when `enableFreeze()` from `react-native-screens` package is run at the top of the application.

Requires `react-native-screens` version >=3.16.0.

Only supported on iOS and Android.

Header related options

You can find the list of header related options here. These options can be specified under `screenOptions` prop of `Stack.navigator` or `options` prop of `Stack.Screen`. You don't have to be using `@react-navigation/elements` directly to use these options, they are just documented in that page.

In addition to those, the following options are also supported in stack:

`header`

Custom header to use instead of the default header.

This accepts a function that returns a React Element to display as a header. The function receives an object containing the following properties as the argument:

- `navigation` - The navigation object for the current screen.
- `route` - The route object for the current screen.
- `options` - The options for the current screen
- `layout` - Dimensions of the screen, contains `height` and `width` properties.
- `progress` Animated nodes representing the progress of the animation.
- `back` - Options for the back button, contains an object with a `title` property to use for back button label.
- `styleInterpolator` - Function which returns interpolated styles for various elements in the header.

Make sure to set `headerMode` to `screen` as well when using a custom header (see below for more details).

Example:

```
import { getHeaderTitle } from '@react-navigation/elements';

// ..

header: ({ navigation, route, options, back }) => {
  const title = getHeaderTitle(options, route.name);

  return (
    <MyHeader
      title={title}
      leftButton={
        back ? <MyBackButton onPress={navigation.goBack} /> : undefined
      }
      style={options.headerStyle}
    />
  );
};
```

To set a custom header for all the screens in the navigator, you can specify this option in the `screenOptions` prop of the navigator.

When using a custom header, there are 2 things to keep in mind:

Specify a `height` in `headerStyle` to avoid glitches

If your header's height differs from the default header height, then you might notice glitches due to measurement being async. Explicitly specifying the height will avoid such glitches.

Example:

```
headerStyle: {
  height: 80, // Specify the height of your custom header
};
```

Note that this style is not applied to the header by default since you control the styling of your custom header. If you also want to apply this style to your header, use `headerStyle` from the props.

Set `headerMode` to `float` for custom header animations

By default, there is one floating header which renders headers for multiple screens on iOS for non-modals. These headers include animations to smoothly switch to one another.

If you specify a custom header, React Navigation will change it to `screen` automatically so that the header animated along with the screen instead. This means that you don't have to implement animations to animate it separately.

But you might want to keep the floating header to have a different transition animation between headers. To do that, you'll need to specify `headerMode: 'float'` in the options, and then interpolate on the `progress.current` and `progress.next` props in your custom header. For example, following will cross-fade the header:

```
const opacity = Animated.add(progress.current, progress.next || 0).interpolate({
  inputRange: [0, 1, 2],
  outputRange: [0, 1, 0],
});

return (
  <Animated.View style={{ opacity }}>{/* Header content */}</Animated.View>
);
```

`headerMode`

Specifies how the header should be rendered:

- `float` - Render a single header that stays at the top and animates as screens are changed. This is default on iOS.
- `screen` - Each screen has a header attached to it and the header fades in and out together with the screen. This is default on other platforms.

`headerShown`

Whether to show or hide the header for the screen. The header is shown by default. Setting this to `false` hides the header.

`headerBackAllowFontScaling`

Whether back button title font should scale to respect Text Size accessibility settings. Defaults to false.

`headerBackAccessibilityLabel`

Accessibility label for the header back button.

`headerBackImage`

Function which returns a React Element to display custom image in header's back button. When a function is used, it receives the `tintColor` in it's argument object. Defaults to Image component with back image source, which is the default back icon image for the platform (a chevron on iOS and an arrow on Android).

`headerBackTitle`

Title string used by the back button on iOS. Defaults to the previous scene's `headerTitle`.

`headerBackTitleVisible`

A reasonable default is supplied for whether the back button title should be visible or not, but if you want to override that you can use `true` or `false` in this option.

`headerTruncatedBackTitle`

Title string used by the back button when `headerBackTitle` doesn't fit on the screen. `"Back"` by default.

`headerBackTitleStyle`

Style object for the back title.

Events

The navigator can emit events on certain actions. Supported events are:

`transitionStart`

This event is fired when the transition animation starts for the current screen.

Event data:

- `e.data.closing` - Boolean indicating whether the screen is being opened or closed.

Example:

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('transitionStart', (e) => {
    // Do something
  });

  return unsubscribe;
}, [navigation]);
```

`transitionEnd`

This event is fired when the transition animation ends for the current screen.

Event data:

- `e.data.closing` - Boolean indicating whether the screen was opened or closed.

Example:

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('transitionEnd', (e) => {
    // Do something
  });

  return unsubscribe;
}, [navigation]);
```

`gestureStart`

This event is fired when the swipe gesture starts for the current screen.

Example:

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('gestureStart', (e) => {
    // Do something
  });

  return unsubscribe;
}, [navigation]);
```

`gestureEnd`

This event is fired when the swipe gesture ends for the current screen. e.g. a screen was successfully dismissed.

Example:

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('gestureEnd', (e) => {
    // Do something
  });

  return unsubscribe;
}, [navigation]);
```

`gestureCancel`

This event is fired when the swipe gesture is cancelled for the current screen. e.g. a screen wasn't dismissed by the gesture.

Example:

```
React.useEffect(() => {
  const unsubscribe = navigation.addListener('gestureCancel', (e) => {
    // Do something
  });

  return unsubscribe;
}, [navigation]);
```

Helpers

The stack navigator adds the following methods to the navigation prop:

`replace`

Replaces the current screen with a new screen in the stack. The method accepts following arguments:

- `name` - *string* - Name of the route to push onto the stack.
- `params` - *object* - Screen params to pass to the destination route.

```
navigation.replace('Profile', { owner: 'Michaś' });
```

`push`

Pushes a new screen to top of the stack and navigate to it. The method accepts following arguments:

- `name` - *string* - Name of the route to push onto the stack.
- `params` - *object* - Screen params to pass to the destination route.

```
navigation.push('Profile', { owner: 'Michaś' });
```

`pop`

Pops the current screen from the stack and navigates back to the previous screen. It takes one optional argument (`count`), which allows you to specify how many screens to pop back by.

```
navigation.pop();
```

`popToTop`

Pops all of the screens in the stack except the first one and navigates to it.

```
navigation.popToTop();
```

Example

```
import { createStackNavigator } from '@react-navigation/stack';

const Stack = createStackNavigator();

function MyStack() {
  return (
    <Stack.Navigator
      initialRouteName="Home"
      screenOptions={{
        headerMode: 'screen',
        headerTintColor: 'white',
        headerStyle: { backgroundColor: 'tomato' },
      }}
    >
      <Stack.Screen
        name="Home"
        component={Home}
        options={{
          title: 'Awesome app',
        }}
      />
      <Stack.Screen
        name="Profile"
        component={Profile}
        options={{
          title: 'My profile',
        }}
      />
      <Stack.Screen
        name="Settings"
        component={Settings}
        options={{
          gestureEnabled: false,
        }}
      />
    </Stack.Navigator>
  );
}
```

Try this example on Snack

Animations

Animation related options

Stack Navigator exposes various options to configure the transition animation when a screen is added or removed. These transition animations can be customized on a per-screen basis by specifying the options in the `options` prop for each screen.

- `gestureDirection` - The direction of swipe gestures:
  - `horizontal` - The gesture to close the screen will start from the left, and from the right in RTL. For animations, screen will slide from the right with `SlideFromRightIOS`, and from the left in RTL.
  - `horizontal-inverted` - The gesture to close the screen will start from the right, and from the left in RTL. For animations, screen will slide from the left with `SlideFromRightIOS`, and from the right in RTL as the direction is inverted.
  - `vertical` - The gesture to close the screen will start from the top. For animations, screen will slide from the bottom.
  - `vertical-inverted` - The gesture to close the screen will start from the bottom. For animations, screen will slide from the top.

  You may want to specify a matching horizontal/vertical animation along with `gestureDirection` as well. For the animations included in the library, if you set `gestureDirection` to one of the inverted ones, it'll also flip the animation direction.

- `transitionSpec` - An object which specifies the animation type (`timing` or `spring`) and their options (such as `duration` for `timing`). It takes 2 properties:
  - `open` - Configuration for the transition when adding a screen
  - `close` - Configuration for the transition when removing a screen.

  Each of the object should specify 2 properties:

  - `animation` - The animation function to use for the animation. Supported values are `timing` and `spring`.
  - `config` - The configuration object for the timing function. For `timing`, it can be `duration` and `easing`. For `spring`, it can be `stiffness`, `damping`, `mass`, `overshootClamping`,

`restDisplacementThreshold` and `restSpeedThreshold`.

A config which uses spring animation looks like this:

```
const config = {
  animation: 'spring',
  config: {
    stiffness: 1000,
    damping: 500,
    mass: 3,
    overshootClamping: true,
    restDisplacementThreshold: 0.01,
    restSpeedThreshold: 0.01,
  },
};
```

We can pass this config in the `transitionSpec` option:

```
<Stack.Screen
  name="Profile"
  component={Profile}
  options={{
    transitionSpec: {
      open: config,
      close: config,
    },
  }}
/>
```

Try this example on Snack

- `cardStyleInterpolator` - This is a function which specifies interpolated styles for various parts of the card. This allows you to customize the transitions when navigating from screen to screen. It is expected to return at least empty object, possibly containing interpolated styles for container, the card itself, overlay and shadow. Supported properties are:
  - `containerStyle` - Style for the container view wrapping the card.
  - `cardStyle` - Style for the view representing the card.
  - `overlayStyle` - Style for the view representing the semi-transparent overlay below

- `shadowStyle` - Style for the view representing the card shadow.

The function receives the following properties in its argument:

- `current` - Values for the current screen:
  - `progress` - Animated node representing the progress value of the current screen.
- `next` - Values for the screen after this one in the stack. This can be `undefined` in case the screen animating is the last one.
  - `progress` - Animated node representing the progress value of the next screen.
- `index` - The index of the card in the stack.
- `closing` - Animated node representing whether the card is closing. `1` when closing, `0` if not.
- `layouts` - Layout measurements for various items we use for animation.
  - `screen` - Layout of the whole screen. Contains `height` and `width` properties.

> **Note that when a screen is not the last, it will use the next screen's transition config.** This is because many transitions involve an animation of the previous screen, and so these two transitions need to be kept together to prevent running two different kinds of transitions on the two screens (for example a slide and a modal). You can check the `next` parameter to find out if you want to animate out the previous screen. For more information about this parameter, see Animation section.

A config which just fades the screen looks like this:

```
const forFade = ({ current }) => ({
  cardStyle: {
    opacity: current.progress,
  },
});
```

We can pass this function in `cardStyleInterpolator` option:

```
<Stack.Screen
  name="Profile"
  component={Profile}
  options={{ cardStyleInterpolator: forFade }}
/>
```

Try this example on Snack

The interpolator will be called for each screen. For example, say you have a 2 screens in the stack, A & B. B is the new screen coming into focus and A is the previous screen. The interpolator will be called for each screen:

- The interpolator is called for `B`: Here, the `current.progress` value represents the progress of the transition, which will start at `0` and end at `1`. There won't be a `next.progress` since `B` is the last screen.
- The interpolator is called for `A`: Here, the `current.progress` will stay at the value of `1` and won't change, since the current transition is running for `B`, not `A`. The `next.progress` value represents the progress of `B` and will start at `0` and end at `1`.

Say we want to animate both screens during the transition. The easiest way to do it would be to combine the progress value of current and next screens:

```
const progress = Animated.add(
  current.progress.interpolate({
    inputRange: [0, 1],
    outputRange: [0, 1],
    extrapolate: 'clamp',
  }),
  next
    ? next.progress.interpolate({
        inputRange: [0, 1],
        outputRange: [0, 1],
        extrapolate: 'clamp',
      })
    : 0
);
```

Here, the screen `A` will have both `current.progress` and `next.progress`, and since `current.progress` stays at `1` and `next.progress` is changing, combined, the progress will change from `1` to `2`. The screen `B` will only have `current.progress` which will change from `0` to `1`. So, we can apply different interpolations for `0-1` and `1-2` to animate focused screen and unfocused screen respectively.

A config which translates the previous screen slightly to the left, and translates the current screen from the right edge would look like this:

```
const forSlide = ({ current, next, inverted, layouts: { screen } }) => {
  const progress = Animated.add(
    current.progress.interpolate({
      inputRange: [0, 1],
      outputRange: [0, 1],
      extrapolate: 'clamp',
    }),
    next
      ? next.progress.interpolate({
          inputRange: [0, 1],
          outputRange: [0, 1],
          extrapolate: 'clamp',
        })
      : 0
  );

  return {
    cardStyle: {
      transform: [
        {
          translateX: Animated.multiply(
            progress.interpolate({
              inputRange: [0, 1, 2],
              outputRange: [
                screen.width, // Focused, but offscreen in the beginning
                0, // Fully focused
                screen.width * -0.3, // Fully unfocused
              ],
              extrapolate: 'clamp',
            }),
            inverted
          ),
        },
      ],
    },
  };
};
```

- `headerStyleInterpolator` - This is a function which specifies interpolated styles for various parts of the header. It is expected to return at least empty object, possibly containing interpolated styles for left label and button, right button, title and background. Supported properties are:

- `leftLabelStyle` - Style for the label of the left button (back button label).
- `leftButtonStyle` - Style for the left button (usually the back button).
- `rightButtonStyle` - Style for the right button.
- `titleStyle` - Style for the header title text.
- `backgroundStyle` - Style for the header background.

The function receives the following properties in it's argument:

- `current` - Values for the current screen (the screen which owns this header).
  - `progress` - Animated node representing the progress value of the current screen. `0` when screen should start coming into view, `0.5` when it's mid-way, `1` when it should be fully in view.
- `next` - Values for the screen after this one in the stack. This can be `undefined` in case the screen animating is the last one.
  - `progress` - Animated node representing the progress value of the next screen.
- `layouts` - Layout measurements for various items we use for animation. Each layout object contain `height` and `width` properties.
  - `screen` - Layout of the whole screen.
  - `title` - Layout of the title element. Might be `undefined` when not rendering a title.
  - `leftLabel` - Layout of the back button label. Might be `undefined` when not rendering a back button label.

A config which just fades the elements looks like this:

```
const forFade = ({ current, next }) => {
  const opacity = Animated.add(
    current.progress,
    next ? next.progress : 0
  ).interpolate({
    inputRange: [0, 1, 2],
    outputRange: [0, 1, 0],
  });

  return {
    leftButtonStyle: { opacity },
    rightButtonStyle: { opacity },
    titleStyle: { opacity },
    backgroundStyle: { opacity },
  };
};
```

We can pass this function in `headerStyleInterpolator` option:

```
<Stack.Screen
  name="Profile"
  component={Profile}
  options={{ headerStyleInterpolator: forFade }}
/>
```

Try this example on Snack

Pre-made configs

With these options, it's possible to build custom transition animations for screens. We also export various configs from the library with ready-made animations which you can use:

`TransitionSpecs`

- `TransitionIOSSpec` - Exact values from UINavigationController's animation configuration.
- `FadeInFromBottomAndroidSpec` - Configuration for activity open animation from Android Nougat.
- `FadeOutToBottomAndroidSpec` - Configuration for activity close animation from Android Nougat.
- `RevealFromBottomAndroidSpec` - Approximate configuration for activity open animation from Android Pie.

Example:

```
import { TransitionSpecs } from '@react-navigation/stack';

// ...

<Stack.Screen
  name="Profile"
  component={Profile}
  options={{
    transitionSpec: {
      open: TransitionSpecs.TransitionIOSSpec,
      close: TransitionSpecs.TransitionIOSSpec,
    },
  }}
/>;
```

`CardStyleInterpolators`

- `forHorizontalIOS` - Standard iOS-style slide in from the right.
- `forVerticalIOS` - Standard iOS-style slide in from the bottom (used for modals).
- `forModalPresentationIOS` - Standard iOS-style modal animation in iOS 13.
- `forFadeFromBottomAndroid` - Standard Android-style fade in from the bottom for Android Oreo.
- `forRevealFromBottomAndroid` - Standard Android-style reveal from the bottom for Android Pie.

Example configuration for Android Oreo style vertical screen fade animation:

```
import { CardStyleInterpolators } from '@react-navigation/stack';

// ...

<Stack.Screen
  name="Profile"
  component={Profile}
  options={{
    title: 'Profile',
    cardStyleInterpolator: CardStyleInterpolators.forFadeFromBottomAndroid,
  }}
/>;
```

Try this example on Snack

`HeaderStyleInterpolators`

- `forUIKit` - Standard UIKit style animation for the header where the title fades into the back button label.
- `forFade` - Simple fade animation for the header elements.

- `forStatic` - Simple translate animation to translate the header along with the sliding screen.

Example configuration for default iOS animation for header elements where the title fades into the back button:

```
import { HeaderStyleInterpolators } from '@react-navigation/stack';

// ...

<Stack.Screen
  name="Profile"
  component={Profile}
  options={{
    title: 'Profile',
    headerStyleInterpolator: HeaderStyleInterpolators.forUIKit,
  }}
/>;
```

Try this example on Snack

> Note: Always define your animation configuration at the top-level of the file to ensure that the references don't change across re-renders. This is important for smooth and reliable transition animations.

`TransitionPresets`

We export various transition presets which bundle various set of these options together to match certain native animations. A transition preset is an object containing few animation related screen options exported under `TransitionPresets`. Currently the following presets are available:

- `SlideFromRightIOS` - Standard iOS navigation transition.
- `ModalSlideFromBottomIOS` - Standard iOS navigation transition for modals.
- `ModalPresentationIOS` - Standard iOS modal presentation style (introduced in iOS 13).
- `FadeFromBottomAndroid` - Standard Android navigation transition when opening or closing an Activity on Android < 9 (Oreo).
- `RevealFromBottomAndroid` - Standard Android navigation transition when opening or closing an Activity on Android 9 (Pie).

- `ScaleFromCenterAndroid` - Standard Android navigation transition when opening or closing an Activity on Android >= 10.
- `DefaultTransition` - Default navigation transition for the current platform.
- `ModalTransition` - Default modal transition for the current platform.

You can spread these presets in `options` to customize the animation for a screen:

```
import { TransitionPresets } from '@react-navigation/stack';

// ...

<Stack.Screen
  name="Profile"
  component={Profile}
  options={{
    title: 'Profile',
    ...TransitionPresets.ModalSlideFromBottomIOS,
  }}
/>;
```

Try this example on Snack

If you want to customize the transition animations for all of the screens in the navigator, you can specify it in `screenOptions` prop for the navigator.

Example configuration for iOS modal presentation style:

```
import { TransitionPresets } from '@react-navigation/stack';

// ...

<Stack.Navigator
  initialRouteName="Home"
  screenOptions={({ route, navigation }) => ({
    headerShown: false,
    gestureEnabled: true,
    ...TransitionPresets.ModalPresentationIOS,
  })}
>
  <Stack.Screen name="Home" component={Home} />
  <Stack.Screen name="Profile" component={Profile} />
</Stack.Navigator>;
```

Try this example on Snack

Transparent modals

A transparent modal is like a modal dialog which overlays the screen. The previous screen still stays visible underneath. To get a transparent modal screen, you can specify `presentation: 'transparentModal'` in the screen's options.

Example:

```
<Stack.Navigator>
  <Stack.Screen name="Home" component={HomeStack} />
  <Stack.Screen
    name="Modal"
    component={ModalScreen}
    options={{ presentation: 'transparentModal' }}
  />
</Stack.Navigator>
```

Now, when you navigate to the `Modal` screen, it'll have a transparent background and the `Home` screen will be visible underneath.

In addition to `presentation`, you might want to optionally specify few more things to get a modal dialog like behavior:

- Disable the header with `headerShown: false`
- Enable the overlay with `cardOverlayEnabled: true` (you can't tap the overlay to close the screen this way, see below for alternatives)

If you want to further customize how the dialog animates, or want to close the screen when tapping the overlay etc., you can use the `useCardAnimation` hook to customize elements inside your screen.

Example:

```
import {
  Animated,
  View,
  Text,
  Pressable,
  Button,
  StyleSheet,
} from 'react-native';
import { useTheme } from '@react-navigation/native';
import { useCardAnimation } from '@react-navigation/stack';

function ModalScreen({ navigation }) {
  const { colors } = useTheme();
  const { current } = useCardAnimation();

  return (
    <View
      style={{
        flex: 1,
        alignItems: 'center',
        justifyContent: 'center',
      }}
    >
      <Pressable
        style={[
          StyleSheet.absoluteFill,
          { backgroundColor: 'rgba(0, 0, 0, 0.5)' },
        ]}
        onPress={navigation.goBack}
      />
      <Animated.View
        style={{
          padding: 16,
          width: '90%',
          maxWidth: 400,
          borderRadius: 3,
          backgroundColor: colors.card,
          transform: [
            {
              scale: current.progress.interpolate({
                inputRange: [0, 1],
                outputRange: [0.9, 1],
                extrapolate: 'clamp',
              }),
            },
          ],
        }}
      >
        <Text>
          Mise en place is a French term that literally means “put in place.” It
          also refers to a way cooks in professional kitchens and restaurants
          set up their work stations—first by gathering all ingredients for a
          recipes, partially preparing them (like measuring out and chopping),
          and setting them all near each other. Setting up mise en place before
          cooking is another top tip for home cooks, as it seriously helps with
          organization. It’ll pretty much guarantee you never forget to add an
          ingredient and save you time from running back and forth from the
          pantry ten times.
        </Text>
        <Button
          title="Okay"
          color={colors.primary}
          style={{ alignSelf: 'flex-end' }}
          onPress={navigation.goBack}
        />
      </Animated.View>
    </View>
  );
}
```

Here we animate the scale of the dialog, and also add an overlay to close the dialog.

Edit this page