Version: 6.x

Custom routers

The router object provides various helper methods to deal with the state and actions, a reducer to update the state as well as some action creators.

The router is responsible for handling actions dispatched by calling methods on the navigation object. If the router cannot handle an action, it can return `null`, which would propagate the action to other routers until it's handled.

You can make your own router by building an object with the following functions:

- `type` - String representing the type of the router, e.g. `'stack'`, `'tab'`, `'drawer'` etc.
- `getInitialState` - Function which returns the initial state for the navigator. Receives an options object with `routeNames` and `routeParamList` properties.
- `getRehydratedState` - Function which rehydrates the full navigation state from a given partial state. Receives a partial state object and an options object with `routeNames` and `routeParamList` properties.
- `getStateForRouteNamesChange` - Function which takes the current state and updated list of route names, and returns a new state. Receives the state object and an options object with `routeNames` and `routeParamList` properties.
- `getStateForAction` - function which takes the current state and action along with an options object with `routeNames` and `routeParamList` properties, and returns a new state. If the action cannot be handled, it should return `null`.
- `getStateForRouteFocus` - Function which takes the current state and key of a route, and returns a new state with that route focused.
- `shouldActionChangeFocus` - Function which determines whether the action should also change focus in parent navigator. Some actions such as `NAVIGATE` can change focus in the parent.
- `actionCreators` - Optional object containing a list of action creators, such as `push`, `pop` etc. These will be used to add helper methods to the `navigation` object to dispatch those actions.

Example:

```
const router = {
  type: 'tab',

  getInitialState({ routeNames, routeParamList }) {
    const index =
      options.initialRouteName === undefined
        ? 0
        : routeNames.indexOf(options.initialRouteName);

    return {
      stale: false,
      type: 'tab',
      key: shortid(),
      index,
      routeNames,
      routes: routeNames.map(name => ({
        name,
        key: name,
        params: routeParamList[name],
      })),
    };
  },

  getRehydratedState(partialState, { routeNames, routeParamList }) {
    const state = partialState;

    if (state.stale === false) {
      return state as NavigationState;
    }

    const routes = state.routes
      .filter(route => routeNames.includes(route.name))
      .map(
        route =>
          ({
            ...route,
            key: route.key || `${route.name}-${shortid()}`,
            params:
              routeParamList[route.name] !== undefined
                ? {
                    ...routeParamList[route.name],
                    ...route.params,
                  }
```

```
              : route.params,
          } as Route<string>)
      );

    return {
      stale: false,
      type: 'tab',
      key: shortid(),
      index:
        typeof state.index === 'number' && state.index < routes.length
          ? state.index
          : 0,
      routeNames,
      routes,
    };
  },

  getStateForRouteNamesChange(state, { routeNames }) {
    const routes = state.routes.filter(route =>
      routeNames.includes(route.name)
    );

    return {
      ...state,
      routeNames,
      routes,
      index: Math.min(state.index, routes.length - 1),
    };
  },

  getStateForRouteFocus(state, key) {
    const index = state.routes.findIndex(r => r.key === key);

    if (index === -1 || index === state.index) {
      return state;
    }

    return { ...state, index };
  },

  getStateForAction(state, action) {
    switch (action.type) {
      case 'NAVIGATE': {
        const index = state.routes.findIndex(
```

```
            route => route.name === action.payload.name
          );

          if (index === -1) {
            return null;
          }

          return { ...state, index };
        }

        default:
          return BaseRouter.getStateForAction(state, action);
      }
    },

    shouldActionChangeFocus() {
      return false;
    },
  };

const SimpleRouter = () => router;

export default SimpleRouter;
```

Built-In Routers

The library ships with a few standard routers:

- `StackRouter`
- `TabRouter`
- `DrawerRouter`

Customizing Routers

You can reuse a router and override the router functions as per your needs, such as customizing how existing actions are handled, adding additional actions etc.

See custom navigators for details on how to override the router with a custom router in an existing navigator.

Custom Navigation Actions

Let's say you want to add a custom action to clear the history:

```
import { TabRouter } from '@react-navigation/native';

const MyTabRouter = options => {
  const router = TabRouter(options);

  return {
    ...router,
    getStateForAction(state, action, options) {
      switch (action.type) {
        case 'CLEAR_HISTORY':
          return {
            ...state,
            routeKeyHistory: [],
          };
        default:
          return router.getStateForAction(state, action, options);
      }
    },

    actionCreators: {
      ...router.actionCreators,
      clearHistory() {
        return { type: 'CLEAR_HISTORY' };
      },
    },
  };
};
```

Instead of writing a custom router to handle custom actions, you can pass a function to `dispatch` instead. It's cleaner and recommended instead of overriding routers.

Blocking Navigation Actions

Sometimes you may want to prevent some navigation activity, depending on your route. Let's say, you want to prevent pushing a new screen if `isEditing` is `true`:

```
import { StackRouter } from '@react-navigation/native';

const MyStackRouter = options => {
  const router = StackRouter(options);

  return {
    ...router,
    getStateForAction(state, action, options) {
      const result = router.getStateForAction(state, action, options);

      if (
        result != null &&
        result.index > state.index &&
        state.routes[state.index].params?.isEditing
      ) {
        // Returning the current state means that the action has been handled,
but we don't have a new state
        return state;
      }

      return result;
    },
  };
};
```

If you want to prevent going back, the recommended approach is to use the `beforeRemove` event.

Edit this page