Integrating with client-side routing libraries

UDS offers a variety of components that can link to a web page via a `href` prop, including many navigation components that, on Web, will usually link to another page on the same site.

Most modern React websites use a routing library that links render states to URLs allowing links to navigate very quickly by selectively re-rendering only those parts of a page that have changed, avoiding clunky whole-page reloads, while still supporting browser features that depend on URLs such as refresh, new tab and bookmarking page.

What to do

There are two steps to integrating link-like components with a router:

1. Add a `LinkRouter` integration component to your project. There are complete code examples below for several popular routing libraries:

- NextJS for web apps built on NextJS and using its own router
- React Navigation for cross-platform React Native apps
- React Router 6 for modern web apps not built on NextJS
- React Router 5 for legacy web apps

2. When using a link or link-like component, pass this to its `LinkRouter` prop; and if necessary, pass any router-specific props (like `to`) in an object to its `linkRouterProps` prop.

How it works

UDS is designed to work with most modern router systems rather than being tied to a specific router:

- Every component that renders a link to a page takes the system props `LinkRouter` and `linkRouterProps`, to connect a special integration component that will wrap this component.

- Apps create a "`LinkRouter`" that takes the props the router needs, like `to` and `replace`, and passes them to the router's tools to generate props like `href` and `onPress` (or `onClick`)
- The app's "`LinkRouter`" also takes a special prop `Component` that can be any link-like component, which the `LinkRouter` forwards its props to.

This way, a `LinkRouter` can be created once then every link-like component can be connected to the app's router system by passing in the `LinkRouter` and, if necessary, any special props it needs.

Below are examples of `LinkRouter`s for popular router libraries, and here is an example of how these `LinkRouter` components may be used:

```
<StackView space={2}>
  <Link variant={{ size: 'small' }} LinkRouter={LinkRouter}
href="/some/route">
    Some route, using `href` prop.
  </Link>
  <ChevronLink
    LinkRouter={LinkRouter}
    linkRouterProps={{ to: ['another', 'route'], replace: true }}
  >
    Another route, using router's own `to` prop.
  </ChevronLink>
  <Pagination LinkRouter={LinkRouter}>
    {/* All page buttons get the LinkRouter from the parent */}
    <Pagination.PageButton linkRouterProps={{ to: ['this',
'route'] }} isActive />
    <Pagination.PageButton linkRouterProps={{ to: ['next',
'route'] }} />
  </Pagination>
</StackView>
```

Integrating your own components

If your project creates new shared components and you want these to be compatible with many routers in the same way UDS components are, you can wrap your component in the `withLinkRouter` higher-order component from @telus-uds/components-base.

For example:

```
import React, { forwardRef } from 'react'
import { withLinkRouter } from '@telus-uds/components-base'

// No need to handle `LinkRouter` and `linkRouterProps` here.
It's all added by withLinkRouter below.
// Component should however take similar props to UDS links: at
minimum, `href` and `onPress`.
const SomeNewLink = forwardRef(({ href, onPress, ...props }, ref)
=> {
  const content = useSomeHook(props)
  return <SomeLink ref={ref} href={href} onPress={onPress}>
{content}</SomeLink>
}
// Properties like displayName are preserved by withLinkRouter
SomeNewLink.displayName = 'SomeNewLink'

// If there are propTypes, withLinkRouter adds `LinkRouter` and
`linkRouterProps` automatically
SomeNewLink.propTypes = {
  href: PropTypes.string
}

// Just pass your component to withLinkRouter before you export
it.
// It adds handling for the `LinkRouter` and `linkRouterProps`
props.
export default withLinkRouter(SomeNewLink)
```

NextJS routing

NextJS (web only) has its own routing system, with many advanced features including pre-loading of link content on hover.

Integrations with NextJS router can be done using Next/Link, which generates `href` and `onClick` props then injects them into its child using `React.cloneElement`.

Below is an example `LinkRouter` that integrates with NextJS, and the UDS examples repo includes an example NextJS app using this integration that can be run locally.

```
import React, { forwardRef } from 'react'

import NextLink from 'next/link'

export default forwardRef(function LinkRouter({ href, Component,
linkRouterProps, ...rest }, ref) {
  return (
    <NextLink href={href} passHref {...linkRouterProps}>
      <Component ref={ref} {...rest} />
    </NextLink>
  )
})
```

React Navigation

React Navigation (cross-platform) is the most popular package for React Native navigation and routing that works across web, Android apps and iOS apps.

Integrations with React Navigation can be done using its `useLinkProps` hook, which returns a cross-platform `onPress` handler, an appropriate `href` if on web, and appropriate React Native accessibility props.

Below is an example `LinkRouter` that integrates with React Navigation, and the UDS examples repo includes an example expo app using this integration that can be run locally.

```
import React, { forwardRef } from 'react'
import { Platform } from 'react-native'
import { useLinkProps } from '@react-navigation/native'

export default forwardRef(function RouterLink(
  { href, linkRouterProps: { to = href || '', ...options } = {},
onPress, Component, ...rest },
  ref
) {
  const {
    onPress: handleNavigation,
    href: resolvedHref,
    ...linkProps
  } = useLinkProps({ to, ...options })

  const handlePress = (...args) => {
    if (typeof onPress === 'function') onPress(...args)
    handleNavigation(...args)
  }
  // React Navigation's useLinkProps returns `href` regardless of
platform.
  // We only want href on Web, we don't want apps to open the
device browser.
  if (Platform.OS === 'web') linkProps.href = resolvedHref

  return <Component ref={ref} {...rest} {...linkProps} onPress=
{handlePress} />
})
```

React Router

React Router (web only) has a different API and different usage since version 6, and many applications have not yet taken the steps to migrate from React Router 5 to 6. Integration therefore needs to be handled differently for modern applications using React Router >= 6 and legacy applications using React Router 5.

React Router 6

React Router 6 has a new API after the project merged with the popular Reach Router project, and comes with improved support for integrating with custom links.

Integrations with React Router 6 can be done using its hook `useLinkClickHandler` to generate the handler function and `useHref` to generate the href.

Below is an example `LinkRouter` that integrates with React Router 6, and the UDS examples repo includes an example CRA app using this integration that can be run locally.

```
import React, { forwardRef } from 'react'

import { useHref, useLinkClickHandler } from 'react-router-dom'

export default forwardRef(function RouterLink(
  {
    href,
    hrefAttrs,
    linkRouterProps: { to = href || '', target =
hrefAttrs?.target, ...options } = {},
    onPress,
    Component,
    ...rest
  },
  ref
) {
  const resolvedHref = useHref(to)
  const handleNavigation = useLinkClickHandler(to, { target,
...options })
  const handlePress = (...args) => {
    if (typeof onPress === 'function') onPress(...args)
    handleNavigation(...args)
  }
  return (
    <Component
      ref={ref}
      href={resolvedHref}
      hrefAttrs={hrefAttrs}
      onPress={handlePress}
      {...rest}
    />
  )
})
```

React Router 5

React Router 5 is still in use by some applications that haven't yet updated their routes to the new approach used in version 6.

Version 5 had basic support for creating custom links via passing a custom `component`. However, to use this in a satisfactory way which matched the features of the router's own links required copying a lot of internal boilerplate logic that other routers exposed in hooks (see for example this React Router issue)

Below is an example `LinkRouter` that integrates with React Router 5, and the UDS examples repo includes an example CRA app using this integration that can be run locally.

```
import React, { forwardRef, useCallback } from 'react'

import { Link as RouterLink } from 'react-router-dom'

/**
 * React Router 5's Link's `component` prop only works on a
component that expects a `navigate` prop,
 * applies some specific boilerplate event handling, then turns
it into a click/press handler.
 */
const withNavigate = (Component) => {
  function ComponentWithNavigate({ navigate, onClick, target,
...props }) {
    // Mostly copied from React Router 5's default link inner
component LinkAnchor
    // to match the behaviour of using React Router 5's `<Link>`
out of the box.
    // https://github.com/remix-run/react-
router/blob/v5/packages/react-router-dom/modules/Link.js#L36
    const onPress = (event) => {
      try {
        if (onClick) onClick(event)
      } catch (error) {
        event.preventDefault()
        throw error
      }

      if (
        !event.defaultPrevented && // onClick prevented default
        event.button === 0 && // ignore everything but left
clicks
        (!target || target === '_self') && // let browser handle
"target=_blank" etc.
        !(event.metaKey || event.altKey || event.ctrlKey ||
event.shiftKey) // ignore clicks with modifier keys
      ) {
        event.preventDefault()
        navigate()
      }
    }
    return <Component onPress={onPress} {...props} />
  }
  ComponentWithNavigate.displayName = `${Component.displayName ||
Component.name}WithNavigate`
  return ComponentWithNavigate
}

export default forwardRef(function LinkRouter(
  { Component, href, onPress, linkRouterProps: { to = href, replace } = {}, hrefAttrs, ...rest },
  ref
) {
  // Can't use HoC the conventional way outside of the render cycle without knowing and rigidly
  // setting a static set of all components that could be passed in.
  // `Component` arg is stable, so the returned memoized function can === itself between renders.
  const renderWithNavigate = useCallback(withNavigate(Component), [Component])

  return (
    <RouterLink
      to={to}
      replace={replace}
      onClick={onPress}
      target={hrefAttrs?.target}
      component={renderWithNavigate}
      ref={ref}
      href={href}
      hrefAttrs={hrefAttrs}
      {...rest}
    />
  )
})
```

This example app can also be compared with the example React Router 6 app above to help understand the differences between versions 5 or 6 when judging whether to update or whether to persist with version 5.

Edit this page