Server-Side Rendering (SSR)

Many React web platforms use Server-Side Rendering (SSR) for fast initial page loads.

UDS provide utilities that must be integrated into the SSR config of web apps, to ensure that the server-rendered version of each page has the right styles.

Note that UDS is "mobile-first" and components rendered on the server will display assuming they are on the smallest viewport: see Limitations section.

What is SSR?

In SSR, page is rendered on the server and sent to the browser as HTML, so users don't have to wait for React's JavaScript render to complete in their browser before they can see a page.

React then "hydrates" the page by running a client-side render (CSR) over this static HTML to enable JavaScript-based interactivity.

NextJS and some other web platforms enable SSR by default. However, a little configuration is needed to ensure styles are rendered correctly.

Using `ssrStyles`

"CSS-in-JS" libraries like Styled Components and React Native Web generate style tags containing style rules for their rendered components, and a step must be added to the SSR build to "collect" these styles and generate a plain-text version of these `<style>` tags.

Both `@telus-uds/ds-allium` and `@telus-uds/components-base` export a utility, `ssrStyles`, to support this.

Use with NextJS

In NextJS, `ssrStyles` should be applied in `_document.js`. This file does not exist by default so may need to be created - it should sit next to `_app.js`, usually in the `pages/` directory.

Here is a complete example, tested in NextJS 12.0.10:

```
// In pages/_document.js
import Document from 'next/document'

// !!! Choose one of these based on your app's needs, and delete
the other !!!
import { ssrStyles } from '@telus-uds/components-web'
// !!! ...or... !!!
import { ssrStyles } from '@telus-uds/components-base'

export default class MyDocument extends Document {
  static async getInitialProps(context) {
    // Initialise UDS's SSR style listener utilities
    const { renderApp, getStyles } = ssrStyles()

    const initialProps = await Document.getInitialProps({
      ...context,
      renderPage: () =>
        context.renderPage({
          enhanceApp: (App) => (props) => {
            // Render the app to a React element, using UDS's
style listeners
            return renderApp(App, props)
          }
        })
    })

    // Generate React <style> elements, using data obtained
during the render
    const styles = getStyles(initialProps.styles)

    // Return UDS-generated styles alongside default props and
styles
    return { ...initialProps, styles }
  }
}
```

Other SSR platforms

UDS's `ssrStyles` utility should be compatible with any React SSR framework that is customisable enough to allow a custom render and insertion of style tags. Use the utility and its functions in the following order:

1. Import `ssrStyles` into the file that manages SSR builds. Be sure that this file is only used during SSR.

```
import { ssrStyles } from '@telus-uds/components-web'
// ...or...
import { ssrStyles } from '@telus-uds/components-base'
```

2. At the start of the SSR process, call `ssrStyles`:

```
const { renderApp, getStyles } = ssrStyles()
```

3. Where the SSR process renders the root component to a React element, call `renderApp`:

```
const element = renderApp(
  AppRoot, // The app's outermost component
  props, // Props object for `AppRoot` e.g. current page
navigation state
  {
    // Optional options object
    renderedByRNW, // pass as true if you use React Native Web's
`AppRegistry.runApplication`
    WrapperComponent // if a component is passed here it'll wrap
AppRoot with no props
  }
)
```

4. After the main content render, when the SSR process collates `<style>` tags into the page `<head>`, call `getStyles`, which returns an array of style tags as React elements.

```
const styles = getStyles(
  existingStyles // if your SSR config has already generated any
style tags, include them here
)
```

Use with Styled Components

If you use the `ssrStyles` utility exported from `@telus-uds/ds-allium`, it has built-in support for Styled Components. Styled Components is used internally within ds-allium for some web-only components, and its version of `ssrStyles` adds Styled Components style collection. This will pick up styled from any components in your app that are built on Styled Components, including external ones.

The `ssrStyles` utility exported from `@telus-uds/components-base` does not include this feature, because it only contains CSS-in-JS from React Native Web. If your app imports from `@telus-uds/components-base` directly but also uses Styled Components elsewhere, you'll need to integrate with Styled Components' `ServerStyleSheet`. Look at the implementation of `ssrStyles` in `packages/ds-allium` in the UDS repo for an example of how to extend `ssrStyles` to add styled-components support.

Limitations

NextJS dev builds

NextJS enables SSR in development builds by default, but has a feature that hides the SSR content until the JavaScript has hydrated. This feature is intended to avoid "flashes of unstyled content" during hot refreshes, but as a side effect, it makes it much harder to see SSR during development.

NextJS SSR builds can be seen during development by either:

- Disabling JavaScript in the browser, then reloading the page
- Generating a local production build

Mobile-first layout

A lot of logic around responsive layout and viewports in UDS works via JavaScript. SSR is not aware of the size of the user's screen, so in SSR any utilities that are based on the screen viewport will behave as if the viewport is `xs`.

For small mobile devices, which tend to be slower and most sensitive to layout shifts, if SSR is set up correctly the page should load extremely quickly and correctly first time.

However, for larger viewports, there may be one noticeable shift early in the "hydration" process, as the page becomes aware that the user's screen is wider than the `xs` default.

For real users, if navigation routers are correctly set up, this layout shift should happen only once, the first time any page is loaded, and will likely happen around the same time as other unavoidable page-load flickers, such as loading fonts and images. The layout shift is often more noticable on development machines, where it may be the only noticable loading flicker, as fonts and images are usually already cached. It will also be more impactful on pages that have not correctly set up navigation routers, causing each page navigation to completely reload the page.

API

`ssrStyles`

Initialises SSR style observers for an app render, and returns initialised `renderApp` and `getStyles` functions.

Parameters

In most cases, no parameters need to be passed when calling `ssrStyles`.

- `appName`: string (optional). If multiple 'apps' are rendered in one process, call `ssrStyles` for each one, with a unique app name string each call so the underlying utilities associate the right styles with the right app.
- `options`: object (optional). These allow `ssrStyles` to be extended to support additional CSS-in-JS libraries. For example, `ssrStyles` in @telus-uds/ds-allium uses this to add Styled Components support.
  - `options.styleGetters`: array (optional). Array of functions to be called by `getStyles`, merged with those created by ssrStyles itself.
  - `options.collectStyles`: function `(ReactElement) => ReactElement` (optional). Called towards the end of `renderApp`, with the rendered app element. Whatever element this function returns will be used to generate SSR content, so either return the original argument, or, if the collection process creates a new, modified element, return that.

Returns an object containing two functions: `{ renderApp, getStyles }`:

`renderApp`

A function returned from `ssrStyles()`.

Server-side renders an app, passing data on styles used to the CSS-in-JS listeners initiated when `ssrStyles` was called.

Parameters

- `AppRoot`: React component (required). The root component for the application being rendered. Depending on the framework this could be your `App` or `Main` component, or the framework's own wrapper around this root component with framework providers and utilities added (e.g. routing)
- `props`: object (optional). The props this root component should be rendered with. For example, NextJS's SSR process may render multiple times passing different routing props to support chunking in the SSR build.
- `options`: object (optional):
  - `options.renderedByRNW`: boolean (optional). Pass this as `true` if your main app render uses React Native Web's `AppRegistry.runApplication`. Doing so ensures the SSR content is wrapped in React Native Web's `AppContainer`, matching the behaviour of React Native Web's renderer. This is false by default, because most web platforms use their own renderers, and React Native Web's `AppContainer` is usually not needed.
  - `options.WrapperComponent`: boolean (optional). Pass a component that takes no props if you need a wrapper around `AppRoot`. Make sure that the same wrapper is used in your main app render, in the same way, else there may be SSR mismatch errors.

Returns a React element.

`getStyles`

A return property of `ssrStyles()`.

Uses the data collected during `renderApp` to generate an array of style tags.

DANGER

`getStyles` must be called after `renderApp` has completed. An error will be thrown if it is called before.

If you see an error that `getStyles` was called before `renderApp`, but your code does call it after, consider these possibilities while debugging:

- It is possible you are calling `renderApp` inside an async function that is not being awaited. For example, NextJS's `Document.getInitialProps` is async and must be awaited before calling `getStyles` if it contains the call to `renderApp`.
- It is possible that your web platform encountered an error rendering your app, but caught the error and continued, despite having failed to complete `renderApp`. NextJS's `Document.getInitialProps` does this for some rendering errors.

Parameters

Any number of React elements containing style objects may be passed, which will be included in the returned array.

Returns an array of React elements rendering `<style>` tags.

Edit this page