Version: 6.x

Server rendering

This guide will cover how to server render your React Native app using React Native for Web and React Navigation. We'll cover the following cases:

1. Rendering the correct layout depending on the request URL
2. Setting appropriate page metadata based on the focused screen

Pre-requisites

Before you follow the guide, make sure that your app already renders fine on server. To do that, you will need to ensure the following:

- All of the dependencies that you use are compiled before publishing to npm, so that you don't get syntax errors on Node.
- Node is configured to be able to `require` asset files such as images and fonts. You can try webpack-isomorphic-tools to do that.
- `react-native` is aliased to `react-native-web`. You can do it with babel-plugin-module-resolver.

> Note: Some of the libraries in React Navigation don't work well on Web, such as `@react-navigation/material-top-tabs`. SSR also doesn't work if you're using Expo libraries.

Rendering the app

First, let's take a look at an example of how you'd do server rendering with React Native Web without involving React Navigation:

```
import { AppRegistry } from 'react-native-web';
import ReactDOMServer from 'react-dom/server';
import App from './src/App';

const { element, getStyleElement } = AppRegistry.getApplication('App');
```

```
const html = ReactDOMServer.renderToString(element);
const css = ReactDOMServer.renderToStaticMarkup(getStyleElement());

const document = `
  <!DOCTYPE html>
  <html style="height: 100%">
  <meta charset="utf-8">
  <meta httpEquiv="X-UA-Compatible" content="IE=edge">
  <meta
    name="viewport"
    content="width=device-width, initial-scale=1, minimum-scale=1, maximum-
scale=1.00001, viewport-fit=cover"
  >
  ${css}
  <body style="min-height: 100%">
  <div id="root" style="display: flex; min-height: 100vh">
  ${html}
  </div>
`;
```

Here, `./src/App` is the file where you have `AppRegistry.registerComponent('App', () => App)`.

If you're using React Navigation in your app, this will render the screens rendered by your home page. However, if you have configured links in your app, you'd want to render the correct screens for the request URL on server so that it matches what'll be rendered on the client.

We can use the `ServerContainer` to do that by passing this info in the `location` prop. For example, with Koa, you can use the `path` and `search` properties from the context argument:

```
app.use(async (ctx) => {
  const location = new URL(ctx.url, 'https://example.org/');

  const { element, getStyleElement } = AppRegistry.getApplication('App');

  const html = ReactDOMServer.renderToString(
    <ServerContainer location={location}>{element}</ServerContainer>
  );

  const css = ReactDOMServer.renderToStaticMarkup(getStyleElement());
```

```
  const document = `
    <!DOCTYPE html>
    <html style="height: 100%">
    <meta charset="utf-8">
    <meta httpEquiv="X-UA-Compatible" content="IE=edge">
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, minimum-scale=1, maximum-
scale=1.00001, viewport-fit=cover"
    >
    ${css}
    <body style="min-height: 100%">
    <div id="root" style="display: flex; min-height: 100vh">
    ${html}
    </div>
`;

  ctx.body = document;
});
```

You may also want to set the correct document title and descriptions for search engines, open graph etc. To do that, you can pass a `ref` to the container which will give you the current screen's options.

```
app.use(async (ctx) => {
  const location = new URL(ctx.url, 'https://example.org/');

  const { element, getStyleElement } = AppRegistry.getApplication('App');

  const ref = React.createRef<ServerContainerRef>();

  const html = ReactDOMServer.renderToString(
    <ServerContainer
      ref={ref}
      location={location}
    >
      {element}
    </ServerContainer>
  );

  const css = ReactDOMServer.renderToStaticMarkup(getStyleElement());
```

```
  const options = ref.current?.getCurrentOptions();

  const document = `
    <!DOCTYPE html>
    <html style="height: 100%">
    <meta charset="utf-8">
    <meta httpEquiv="X-UA-Compatible" content="IE=edge">
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, minimum-scale=1, maximum-
scale=1.00001, viewport-fit=cover"
    >
    ${css}
    <title>${options.title}</title>
    <body style="min-height: 100%">
    <div id="root" style="display: flex; min-height: 100vh">
    ${html}
    </div>
`;

  ctx.body = document;
});
```

Make sure that you have specified a `title` option in your screens:

```
<Stack.Screen
  name="Profile"
  component={ProfileScreen}
  options={{ title: 'My profile' }}
/>
```

Handling 404 or other status codes

When rendering a screen for an invalid URL, we should also return a `404` status code from the server.

First, we need to create a context where we'll attach the status code. To do this, place the following code in a separate file that we will be importing on both the server and client:

```
import * as React from 'react';

const StatusCodeContext = React.createContext();

export default StatusCodeContext;
```

Then, we need to use the context in our `NotFound` screen. Here, we add a `code` property with the value of `404` to signal that the screen was not found:

```
function NotFound() {
  const status = React.useContext(StatusCodeContext);

  if (status) {
    staus.code = 404;
  }

  return (
    <View>
      <Text>Oops! This URL doesn't exist.</Text>
    </View>
  );
}
```

You could also attach additional information in this object if you need to.

Next, we need to create a status object to pass in the context on our server. By default, we'll set the `code` to `200`. Then pass the object in `StatusCodeContext.Provider` which should wrap the element with `ServerContainer`:

```
// Create a status object
const status = { code: 200 };

const html = ReactDOMServer.renderToString(
  // Pass the status object via context
  <StatusCodeContext.Provider value={status}>
    <ServerContainer ref={ref} location={location}>
      {element}
    </ServerContainer>
  </StatusCodeContext.Provider>
```

```
);

// After rendering, get the status code and use it for server's response
ctx.status = status.code;
```

After we render the app with `ReactDOMServer.renderToString`, the `code` property of the `status` object will be updated to be `404` if the `NotFound` screen was rendered.

You can follow a similar approach for other status codes too, for example, `401` for unauthorized etc.

Summary

- Use the `location` prop on `ServerContainer` to render correct screens based on the incoming request.
- Attach a `ref` to the `ServerContainer` get options for the current screen.
- Use context to attach more information such as status code.

Edit this page