Server rendering

The most common use case for server-side rendering is to handle the initial render when a user (or search engine crawler) first requests your app.

When the server receives the request, it renders the required component(s) into an HTML string, and then sends it as a response to the client. From that point on, the client takes over rendering duties.

MUI on the server

MUI was designed from the ground-up with the constraint of rendering on the server, but it's up to you to make sure it's correctly integrated. It's important to provide the page with the required CSS, otherwise the page will render with just the HTML then wait for the CSS to be injected by the client, causing it to flicker (FOUC). To inject the style down to the client, we need to:

1. Create a fresh, new `emotion cache` instance on every request.
2. Render the React tree with the server-side collector.
3. Pull the CSS out.
4. Pass the CSS along to the client.

On the client-side, the CSS will be injected a second time before removing the server-side injected CSS.

Setting up

In the following recipe, we are going to look at how to set up server-side rendering.

The theme

Create a theme that will be shared between the client and the server:

`theme.js`

```
import { createTheme } from '@mui/material/styles';
import { red } from '@mui/material/colors';

// Create a theme instance.
const theme = createTheme({
  palette: {
    primary: {
      main: '#556cd6',
    },
    secondary: {
      main: '#19857b',
    },
    error: {
      main: red.A400,
```

```
    },
  },
});

export default theme;
```

The server-side

The following is the outline for what the server-side is going to look like. We are going to set up an Express middleware using app.use to handle all requests that come into the server. If you're unfamiliar with Express or middleware, know that the `handleRender` function will be called every time the server receives a request.

`server.js`

```
import express from 'express';

// We are going to fill these out in the sections to follow.
function renderFullPage(html, css) {
  /* ... */
}

function handleRender(req, res) {
  /* ... */
}

const app = express();

// This is fired every time the server-side receives a request.
app.use(handleRender);

const port = 3000;
app.listen(port);
```

Handling the request

The first thing that we need to do on every request is to create a new `emotion cache`.

When rendering, we will wrap `App`, the root component, inside a `CacheProvider` and `ThemeProvider` to make the style configuration and the `theme` available to all components in the component tree.

The key step in server-side rendering is to render the initial HTML of the component **before** we send it to the client-side. To do this, we use ReactDOMServer.renderToString().

MUI is using emotion as its default styled engine. We need to extract the styles from the emotion instance. For this, we need to share the same cache configuration for both the client and server:

createEmotionCache.js

```
import createCache from '@emotion/cache';

export default function createEmotionCache() {
  return createCache({ key: 'css' });
}
```

With this we are creating new emotion cache instance and using this to extract the critical styles for the html as well.

We will see how this is passed along in the `renderFullPage` function.

```
import express from 'express';
import * as React from 'react';
import ReactDOMServer from 'react-dom/server';
import CssBaseline from '@mui/material/CssBaseline';
import { ThemeProvider } from '@mui/material/styles';
import { CacheProvider } from '@emotion/react';
import createEmotionServer from '@emotion/server/create-instance';
import App from './App';
import theme from './theme';
import createEmotionCache from './createEmotionCache';

function handleRender(req, res) {
  const cache = createEmotionCache();
  const { extractCriticalToChunks, constructStyleTagsFromChunks } =
    createEmotionServer(cache);

  // Render the component to a string.
  const html = ReactDOMServer.renderToString(
    <CacheProvider value={cache}>
      <ThemeProvider theme={theme}>
        {/* CssBaseline kickstart an elegant, consistent, and simple baseline to build upon.
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>,
  );

  // Grab the CSS from emotion
  const emotionChunks = extractCriticalToChunks(html);
  const emotionCss = constructStyleTagsFromChunks(emotionChunks);
```

```
  // Send the rendered page back to the client.
  res.send(renderFullPage(html, emotionCss));
}

const app = express();

app.use('/build', express.static('build'));

// This is fired every time the server-side receives a request.
app.use(handleRender);

const port = 3000;
app.listen(port);
```

Inject initial component HTML and CSS

The final step on the server-side is to inject the initial component HTML and CSS into a template to be rendered on the client-side.

```
function renderFullPage(html, css) {
  return `
    <!DOCTYPE html>
    <html>
      <head>
        <title>My page</title>
        ${css}
        <meta name="viewport" content="initial-scale=1, width=device-width" />
        <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,
      </head>
      <body>
        <div id="root">${html}</div>
      </body>
    </html>
  `;
}
```

The client-side

The client-side is straightforward. All we need to do is use the same cache configuration as the server-side. Let's take a look at the client file:

client.js

```
import * as React from 'react';
import ReactDOM from 'react-dom';
import CssBaseline from '@mui/material/CssBaseline';
import { ThemeProvider } from '@mui/material/styles';
import { CacheProvider } from '@emotion/react';
```

```
import App from './App';
import theme from './theme';
import createEmotionCache from './createEmotionCache';

const cache = createEmotionCache();

function Main() {
  return (
    <CacheProvider value={cache}>
      <ThemeProvider theme={theme}>
        {/* CssBaseline kickstart an elegant, consistent, and simple baseline to build upon.
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>
  );
}

ReactDOM.hydrate(<Main />, document.querySelector('#root'));
```

Reference implementations

We host different reference implementations which you can find in the GitHub repository under the `/examples` folder:

- The reference implementation of this tutorial
- Gatsby
- Next.js (TypeScript version)

Troubleshooting

Check out the FAQ answer: My App doesn't render correctly on the server.