

20. Server Side Rendering

20.1 SSR의 이해

- 클라이언트 사이드 렌더링은 UI 렌더링을 모두 브라우저에서 처리한다.

20.1.1 SSR의 장점

- 웹 서비스 검색엔진 최적화가 가능
- 초기 렌더링 성능을 개선
- JS 파일 다운로드 전에도 html을 사용자가 볼 수 있기 때문에 대기시간 최소화, 사용자 경험 향상이 된다.

20.1.2 SSR의 단점

- 서버 리소스가 사용된다. 사용자가 많은 서비스라면 캐싱과 로드 밸런싱을 통해 성능을 최적화 해 주어야 한다.
- 프로젝트구조가 좀 더 복잡해 질 수 있다.
- 데이터 미리 불러오기, 코드스플리팅과의 호환등 고려사항이 많아져서 개발이 어려워 질 수 있다.

20.1.3 SSR과 코드스플리팅 충돌

- SSR과 코드스플리팅을 함께 적용하면 작업이 까다롭다. 별도의 호환작업없이 함께 적용하면 페이지에 깜박임이 발생한다.
- 이러한 이슈를 해결하려면 라우트경로마다 코드 스플리팅된 파일 중에서 필요한 모든 파일을 렌더링 전에 불러와야 한다.
- 여기서는 Loadable Components에서 제공하는 기능을 써서 SSR후 필요한 파일의 경로를 추출해서 렌더링결과에 스크립트/스타일태그를 삽입해 주는 방법
- 상기 방법은 2019년 시점, 2022년 시점에서는 Remix 또는 Next.js로 훨씬 쉽게 처리할 수 있다.
- 2022년이후에는 Remix프레임워크를 사용하는 것을 권장 (사용법 : <https://velog.io/@velopert/learn-remix>)

20.2 프로젝트 준비하기

- 설치 : `yarn add react-router-dom`

20.2.1 컴퍼넌트 만들기

components/Red.js

```
import './Red.css'

const Red = () => {
  return <div className="Red">Red</div>
}
```

```
export default Red;
```

components/Red.css

```
.Red {  
  background: red;  
  font-size: 1.5rem;  
  color: white;  
  width: 128px;  
  height: 128px;  
  display: flex;  
  align-items: center;  
  justify-content: center;  
}
```

components/Blue.js

```
import './Blue.css'  
  
const Blue = () => {  
  return <div className="Red">Red</div>  
}  
  
export default Red;
```

components/Blue.css

```
.Blue {  
  background: Blue;  
  font-size: 1.5rem;  
  color: white;  
  width: 128px;  
  height: 128px;  
  display: flex;  
  align-items: center;  
  justify-content: center;  
}
```

components/Menu.js

```
import {Link} from 'react-router-dom';  
  
const Menu = () => {  
  return(  
    <ul>  
      <li>  
        <Link to="/red">Red</Link>  
      </li>  
      <li>  
        <Link to="/blue">Blue</Link>  
      </li>  
    </ul>  
  );  
}  
  
export default Menu;
```

20.2.2 페이지 컴퍼넌트 만들기

pages/RedPage.js

```
import Red from '../components/Red';

const RedPage = () => {
  return <Red />
}

export default RedPage;
```

pages/BluePage.js

```
import Blue from '../components/Red';

const BluePage = () => {
  return <Blue />
}

export default BluePage;
```

src/App.js

```
import { Route, Routes } from 'react-router-dom';
import Menu from '../components/Menu';
import RedPage from '../pages/RedPage';
import BluePage from '../pages/BluePage';

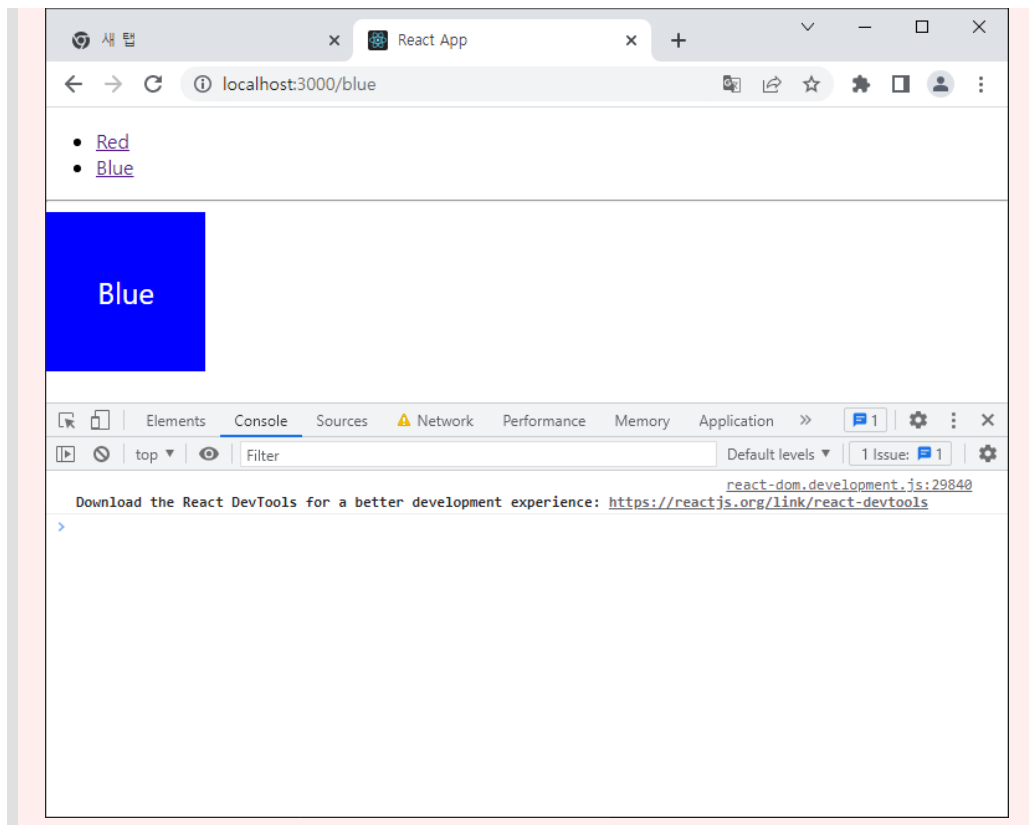
function App() {
  return (
    <div>
      <Menu />
      <hr />
      <Routes>
        <Route path="/red" element={ <RedPage /> } />
        <Route path="/blue" element={ <BluePage /> } />
      </Routes>
    </div>
  );
}

export default App;
```

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```



20.3 SSR 구현하기

- SSR을 구현하려면 웹팩설정을 커스터마이징해야 한다. CRA로 만든 프로젝트는 웹팩 설정이 기본적으로 숨겨져 있다.
- `yarn eject` 명령어를 실행하여 밖으로 꺼내야 한다.

```

• git add .
• git commit -m'Commit before eject'
• yarn eject

```

- 실행후에 `import`구문에서 `NODE_DEV` 환경변수가 설정되어있지 않다는 오류가 발생할 것이다.
- 해당 오류가 발생한다면 `package.json` 을 아래와 같이 수정 - VS코드에 오류가 보이지 않는다면 이 작업은 생략가능

```

"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ],
  "parserOptions": {
    "babelOptions": {
      "presets": [
        [
          "babel-parset-react-app",
          false
        ],
        "babel-preset-react-app/prod"
      ]
    }
  }
}

```

```
  }
},
```

20.3.1 서버사이드 랜더링용 엔트리 만들기

- 엔트리는 웹팩에서 프로젝트를 불러올 때 가장 먼저 불러오는 파일이다.
 - 일반적인 리액트프로젝트에서는 index.js를 엔트리파일로 사용한다.
- SSR의 엔트리파일은 별도로 생성해야 한다. src/index.server.js 파일을 생성
 - 서버에서 랜더링할 때는 ReactDOMServer.renderToString() 을 사용한다.
 - 이 함수에 jsx를 넣어서 호출한다.

src/index.server.js

```
import ReactDOMServer from 'react-dom/server';

const html = ReactDOMServer.renderToString(
  <div>Hello Server Side Rendering!!</div>
)

console.log(html);
```

20.3.2 SSR 전용 웹팩환경 설정 작성하기

- 엔트리파일을 웹팩으로 불러와서 빌드하려면 서버 전용 환경 설정을 만들어 야 한다.
- config\paths.js 파일에 module.exports부분을 아래부분 추가
 - ssrIndexJs 는 불러올 파일의 경로이고 ssrBuild는 웹팩처리 결과물을 저장할 경로

20.ssr\config\paths.js

```
module.exports = {
  dotenv: resolveApp('.env'),
  appPath: resolveApp('.'),
  (... 중간생략 ...)
  publicUrlOrPath,
  ssrIndexJs: resolveApp('src/index.server.js'), // SSR 엔트리
  ssrBuild: resolveApp('dist'), // 웹팩처리후 저장 경로
};
```

```
module.exports.moduleFileExtensions = moduleFileExtensions;
```

20.ssr\config\webpack.config.server.js

- 빌드할 때 웹팩 기본설정을 작성. 빌드할 때 어떤 파일에서 시작하는 파일을 불러 올지, 결과물을 어디에 저장할지 를 설정
- 로더를 설정. 웹팩의 로더는 파일을 불러올 확장자에 맞는 처리를 한다.
 - 자바스크립트는 babel을 사용하여 트랜스파일링을 하고
 - CSS는 모준 CSS코드를 결합해 주고
 - 이미지파일은 다른경로에 별도로 저장하고
 - 그 파일에 대한 경로를 자바스크립트에서 참조할 수 있게 한다.
- 코드에서 node_modules내부의 라이브러리를 불러올 수 있도록 설정

```
( ... )
resolve: {
```

```
modules: ['node_modules'],
},
```

- 상기와 같이 했을 경우 react, react-dom/server같은 라이브러리를 import구문으로 불러오면 node_modules에서 찾아 사용한다.
- 라이브러리를 불러오면 빌드할 때 결과물 파일안에 해당 라이브러리 관련코드가 함께 번들링된다.
- 서버에서는 결과물파일안에 리액트 라이브러리가 없어도 된다. node_modules를 통해 바로 사용할 수 있기 때문이다.
- 서버를 번들링할 때는 node_modules를 제외하고 번들링하는 것이 좋다.
- 이를 위해서 webpack-node-externals 라이브러리를 사용해야 한다.

```
yarn add webpack-node-externals
```

```
const paths = require('./paths');
const getCSSModuleLocalIdent = require('react-dev-
utils/getCSSModuleLocalIdent');
const nodeExternals = require('webpack-node-externals');
const webpack = require('webpack');
const getClientEnvironment = require('./env');

const cssRegex = /\.css$/;
const cssModuleRegex = /\.module\.css$/;
const sassRegex = /\.scss$/;
const sassModuleRegex = /\.module\.scss$/;

const env = getClientEnvironment(paths.publicUrlOrPath.slice(0, -1));

module.exports = {
  mode: 'production', // 프로덕션모드로 설정하여 최적화 옵션들을 활성화
  entry: paths.ssrIndexJs, // 엔트리경로
  target: 'node', // node환경에서 실행할 것이라는 것을 명시
  output: {
    path: paths.ssrBuild, // 빌드경로
    filename: 'server.js', // 파일이름
    chunkFilename: 'js/[name].chunk.js', // 청크파일이름
    publicPath: paths.publicUrlOrPath, // 정적파일이 제공될 경로
  },
  module: {
    rules: [
      {
        oneOf: [
          // 자바스크립트를 위한 처리
          // 기존 webpack.config.js 를 참고하여 작성
          {
            test: /\.js$/,
            include: paths.appSrc,
            loader: require.resolve('babel-loader'),
            options: {
              customize: require.resolve(
                'babel-preset-react-app/webpack-overrides'
              ),
              presets: [
                require.resolve('babel-preset-react-app'),
              ],
            },
          },
        ],
      },
    ],
  },
};
```

```

        runtime: 'automatic',
      },
    ],
  ],
  plugins: [
    [
      require.resolve('babel-plugin-named-asset-import'),
      {
        loaderMap: {
          svg: {
            ReactComponent:
              '@svgr/webpack?-svggo,+titleProp,+ref![path]',
          },
        },
      },
    ],
  ],
  cacheDirectory: true,
  cacheCompression: false,
  compact: false,
},
},
// CSS 를 위한 처리
{
  test: cssRegex,
  exclude: cssModuleRegex,
  // exportOnlyLocals: true 옵션을 설정해야 실제 css 파일을 생성하
지 않습니다.
  loader: require.resolve('css-loader'),
  options: {
    importLoaders: 1,
    modules: {
      exportOnlyLocals: true,
    },
  },
},
},
// CSS Module 을 위한 처리
{
  test: cssModuleRegex,
  loader: require.resolve('css-loader'),
  options: {
    importLoaders: 1,
    modules: {
      exportOnlyLocals: true,
      getLocalIdent: getCSSModuleLocalIdent,
    },
  },
},
},
// Sass 를 위한 처리
{
  test: sassRegex,
  exclude: sassModuleRegex,
  use: [
    {
      loader: require.resolve('css-loader'),
      options: {
        importLoaders: 3,

```

```

        modules: {
          exportOnlyLocals: true,
        },
      },
    },
    require.resolve('sass-loader'),
  ],
},
// Sass + CSS Module 을 위한 처리
{
  test: sassRegex,
  exclude: sassModuleRegex,
  use: [
    {
      loader: require.resolve('css-loader'),
      options: {
        importLoaders: 3,
        modules: {
          exportOnlyLocals: true,
          getLocalIdent: getCSSModuleLocalIdent,
        },
      },
    },
    require.resolve('sass-loader'),
  ],
},
// url-loader 를 위한 설정
{
  test: [/\.bmp$/, /\.gif$/, /\.jpe?g$/, /\.png$/],
  loader: require.resolve('resolve-url-loader'),
  options: {
    emitFile: false, // 파일을 따로 저장하지 않는 옵션
    limit: 10000, // 원래는 9.76KB가 넘어가면 파일로 저장하는데
    // emitFile 값이 false 일때 경로만 준비하고 파일은 저장하지 않습
니다.
    name: 'static/media/[name].[hash:8].[ext]',
  },
},
// 위에서 설정된 확장자를 제외한 파일들은
// file-loader 를 사용합니다.
{
  loader: require.resolve('file-loader'),
  exclude: [/\.js$/, /\.mjs$/, /\.ts$/, /\.tsx$/, /\.html$/, /\.json$/],
  options: {
    emitFile: false, // 파일을 따로 저장하지 않는 옵션
    name: 'static/media/[name].[hash:8].[ext]',
  },
},
],
},
],
},
resolve: {
  modules: ['node_modules'],
},
externals: [nodeExternals({
  allowlist: [/@babel/]
})]

```



```

    ]],
    plugins: [
      new webpack.DefinePlugin(env.stringified), // 환경변수를 주입해줍니다.
    ],
  };

```

20.3.3 빌드스크립트 작성하기

- 상기에서 설정한 환경설정정보를 이용하여 웹팩으로 프로젝트를 빌드하는 스크립트를 작성

20.ssr/scripts/build.server.js

```

process.env.BABEL_ENV = 'production';
process.env.NODE_ENV = 'production';

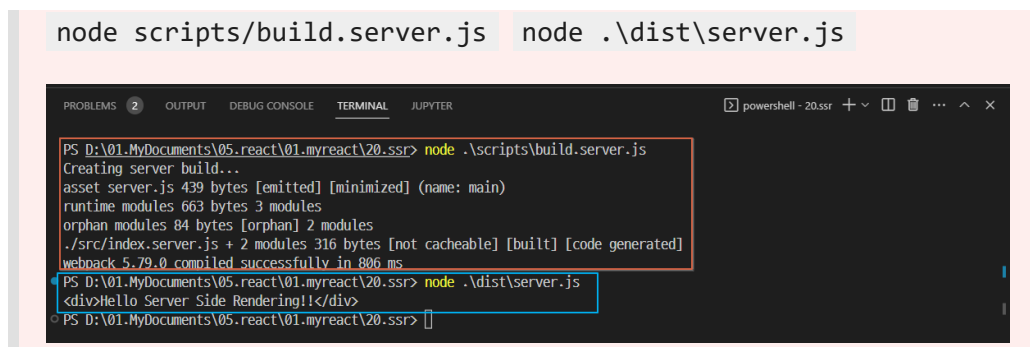
process.on('unhandledRejection', (err) => {
  throw err;
});

require('../config/env');
const fs = require('fs-extra');
const webpack = require('webpack');
const config = require('../config/webpack.config.server');
const paths = require('../config/paths');

function build() {
  console.log('Creating server build...');
  fs.emptyDirSync(paths.ssrBuild);
  let compiler = webpack(config);
  return new Promise((resolve, reject) => {
    compiler.run((err, stats) => {
      if (err) {
        console.log(err);
        return;
      }
      console.log(stats.toString());
    });
  });
}

build();

```



- 테스트시에 만들었던 `<div>Hello Server Side Rendering!!</div>` jsx문자열 형태 랜더링 확인

- 매번 빌드하고 실행할 때 마다 편리하게 `package.json` 에서 스크립트생성하여 편하게 명령어 사용하기

`package.json`

```
( ... )
"scripts": {
  "start": "node scripts/start.js",
  "build": "node scripts/build.js",
  "test": "node scripts/test.js",
  "start:server": "node dist/server.js",
  "build:server": "node scripts/build.server.js"
},
( ... )
```

- 스크립트를 만들면 다음 명령어로 서버를 빌드하고 시작할 수 있다.

```
▪ yarn build:server
▪ yarn start:server
```

20.3.4 서버코드 작성하기

- 서버 사이드 렌더링을 처리할 서버 작성
- Express를 이용하여 웹서버를 만드는데 Koa, Hapi, connect라는 라이브러리를 사용해도 구현할 수 있다.
- express설치 : `yarn add express`

`src/index.server.js`

```
import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';

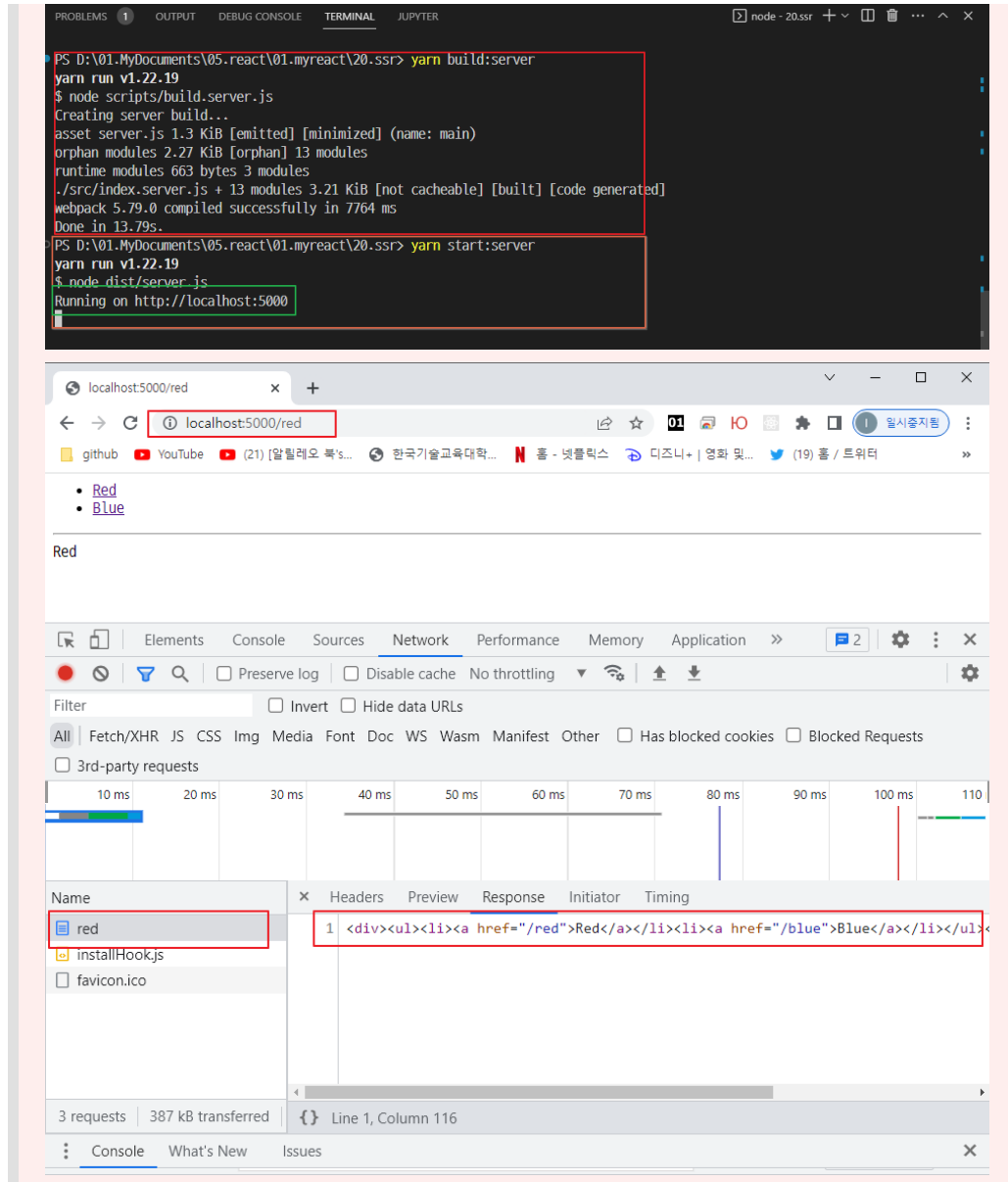
const app = express();

// 서버 사이드 렌더링을 처리할 핸들러 함수
const serverRender = (req, res, next) => {
  // 이 함수는 404가 떠야하는 상황에 404를 띄우지 않고 서버사이드렌더링을 해준다.
  const context = {};
  const jsx = (
    <StaticRouter location={req.url} context={context}>
      <App />
    </StaticRouter>
  );
  const root = ReactDOMServer.renderToString(jsx); // 렌더링을 하고
  res.send(root); // 클라이언트에게 결과를 리턴
};

app.use(serverRender);

// 5000번 포트로 서버를 가동
app.listen(5000, () => {
  console.log('Running on http://localhost:5000');
});
```

- StaticRouter는 주로 서버 사이드 렌더링용으로 사용하는 라우터
- props에 location에 따라 라우팅을 해준다.
- 또한, context props값을 이용하여 나중에 렌더링한 컴포넌트에 따라 http status 코드를 설정해 줄 수 있다.
- 다시 서버를 시작하고 빌드
 - yarn build:server
 - yarn start:server



20.3.5 정적파일 제공하기

src/index.server.js

```
import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';

const app = express();

// 서버 사이드 렌더링을 처리할 핸들러 함수
const serverRender = (req, res, next) => {
```

// 이 함수는 404가 떠야하는 상황에 404를 띄우지 않고 서버사이드랜더링을 해준다.

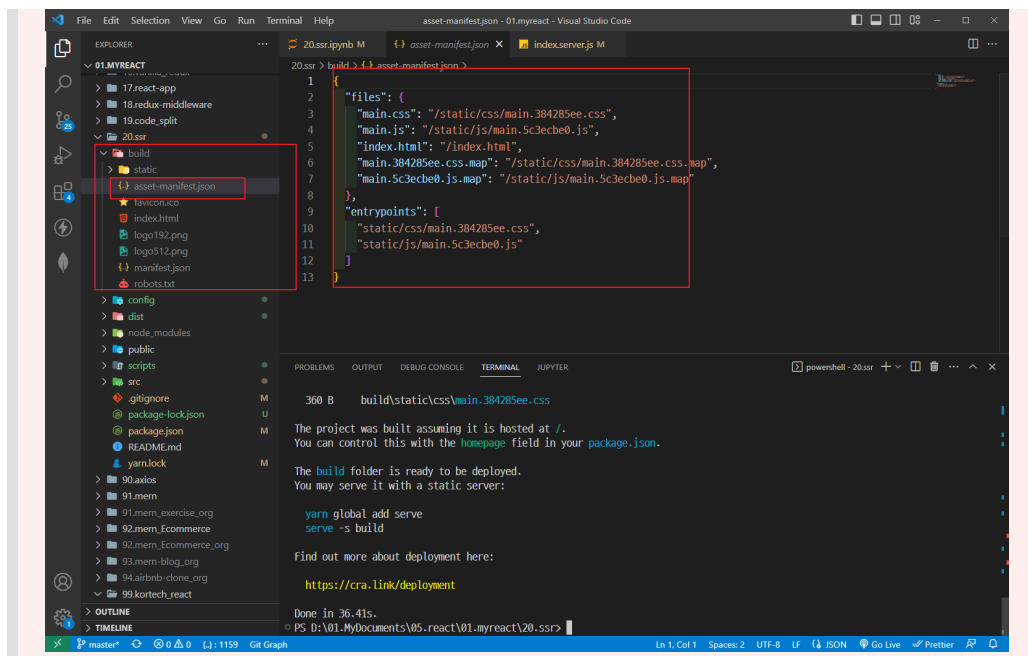
```
const context = {};
const jsx = (
  <StaticRouter location={req.url} context={context}>
    <App />
  </StaticRouter>
);
const root = ReactDOMServer.renderToString(jsx); // 랜더링을 하고
res.send(root); // 클라이언트에게 결과를 리턴
};
```

```
const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});
```

```
app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);
```

```
// 5000번 포트로 서버를 가동
app.listen(5000, () => {
  console.log('Running on http://localhost:5000');
});
```

- JS와 css파일을 불러오도록 html에 코드를 삽입해야 한다.
- 불러와야할 파일이름은 매번빌드할 때마다 변경되기 때문에 asset-manifest.json파일을 참고
- `yarn build` 명령어 실행후 build/asset-manifest.json 수정



build/asset-manifest.json

```
{
  "files": {
    "main.css": "/static/css/main.384285ee.css",
    "main.js": "/static/js/main.5c3ecbe0.js",
    "index.html": "/index.html",
    "main.384285ee.css.map": "/static/css/main.384285ee.css.map",
    "main.5c3ecbe0.js.map": "/static/js/main.5c3ecbe0.js.map"
  }
}
```

```

    },
    "entrypoints": [
      "static/css/main.384285ee.css",
      "static/js/main.5c3ecbe0.js"
    ]
  }
}

```

- css와 js파일을 html에 삽입해 주어야 한다.

```

  ▪ "main.css": "/static/css/main.384285ee.css",
  ▪ "main.js": "/static/js/main.5c3ecbe0.js",

```

src/index.server.js

```

import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';
import fs from 'fs';

// asset-manifest.json에서 파일경로들을 조회
const manifest = JSON.parse(
  fs.readFileSync(path.resolve('./build/asset-manifest.json'), 'utf-8')
);

function createPage(root, tags) {
  return `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width,initial-scale=1,shrink-to-fit=no"
  />
  <meta name="theme-color" content="#000000" />
  <title>React App</title>
  <link href="${manifest.files['main.css']}" rel="stylesheet" />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root">${root}</div>
  <script src="${manifest.files['main.js']}"></script>
</body>
</html>
`;
}

const app = express();

// 서버 사이드 랜더링을 처리할 핸들러 함수
const serverRender = (req, res, next) => {
  // 이 함수는 404가 떠야하는 상황에 404를 띄우지 않고 서버사이드랜더링을 해준다.
  const context = {};
  const jsx = (

```

```

<StaticRouter location={req.url} context={context}>
  <App />
</StaticRouter>
);
const root = ReactDOMServer.renderToStringjsx); // 랜더링을 하고
// res.send(root); // 클라이언트에게 결과를 리턴
res.send(createPage(root)); // 클라이언트에게 결과를 리턴
};

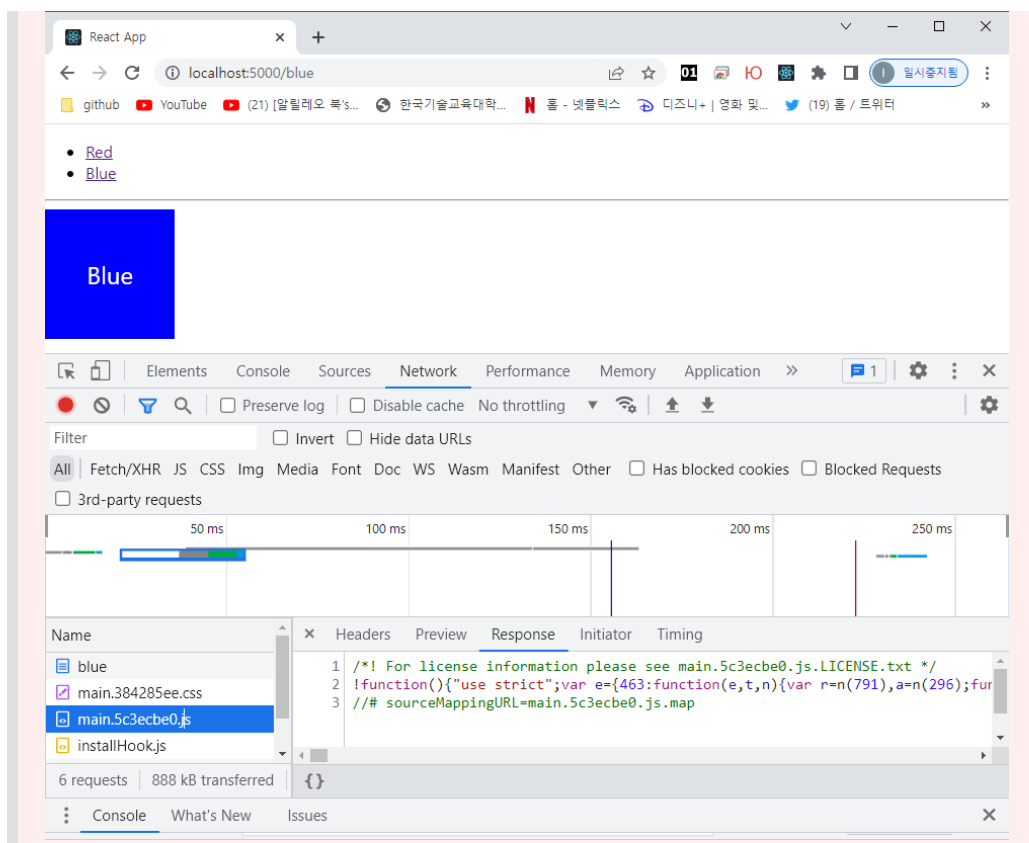
const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});

app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);

// 5000번 포트로 서버를 가동
app.listen(5000, () => {
  console.log('Running on http://localhost:5000');
});

```

- 다시 서버를 시작하고 빌드
 - yarn build:server
 - yarn start:server



20.4 데이터로딩

- 일반 브라우저환경에서는 API요청, 응답을 받은 후 state, redux store에 넣으면 자동으로 랜더링한다

- 하지만, SSR의 경우 문자열형태로 랜더링 하기 때문에 자동으로 랜더링되지 않는다.
- 그대신 renderToString함수를 한번 더 호출해 줘야 한다. 또한 componentDidMount 같은 API도 사용할 수 없다.
- 여기서는 redux-thunk, redux-sga 미들웨어를 사용하여 API를 호출환경에서 SSR방법을 알아 보자
- 라이브러리설치
 - yarn add webpack-node-externals
 - yarn add express
 - yarn add redux react-redux redux-thunk axios

20.4.1 redux-thunk 코드 준비하기

- 20.1.ssr 폴더 전체 복사
- 라이브러리설치 :
 - npm i
 - yarn redux react-redux redux-thunk axios

src/modules/users.js

- thunk함수(getUsers)를 만들고 액션 GET_USERS_PENDING/GET_USERS_SUCCESS/GET_USERS_FAILURE로 상태관리
- 모듈의 상태에선 loading과 error객체가 들어 있다.

```
import axios from 'axios';

const GET_USERS_PENDING = 'users/GET_USERS_PENDING';
const GET_USERS_SUCCESS = 'users/GET_USERS_SUCCESS';
const GET_USERS_FAILURE = 'users/GET_USERS_FAILURE';

const getUsersPending = () => ({ type: GET_USERS_PENDING });
const getUsersSuccess = payload => ({ type: GET_USERS_SUCCESS, payload });
const getUsersFailure = payload => ({
  type: GET_USERS_FAILURE,
  error: true,
  payload
});

export const getUsers = () => async dispatch => {
  try {
    dispatch(getUsersPending());
    const response = await axios.get(
      'https://jsonplaceholder.typicode.com/users'
    );
    dispatch(getUsersSuccess(response));
  } catch (e) {
    dispatch(getUsersFailure(e));
    throw e;
  }
};

const initialState = {
  users: null,
```

```

    user: null,
    loading: {
      users: false,
      user: false
    },
    error: {
      users: null,
      user: null
    }
  }
};

function users(state = initialState, action) {
  switch (action.type) {
    case GET_USERS_PENDING:
      return {
        ...state,
        loading: { ...state.loading, users: true }
      };
    case GET_USERS_SUCCESS:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        users: action.payload.data
      };
    case GET_USERS_FAILURE:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        error: { ...state.error, users: action.payload }
      };
    default:
      return state;
  }
}

export default users;

```

src/modules/index.js

- 모듈을 작성한 후, 루트리듀서를 만들과 Provider 컴퍼넌트를 사용하여 프로젝트에 리덕스를 적용

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { BrowserRouter } from 'react-router-dom';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer from './modules';

const store = createStore( rootReducer, applyMiddleware(thunk));

const root = ReactDOM.createRoot(document.getElementById('root'));

```



```

root.render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
);

```

20.4.2 Users, UsersContainer 컴퍼넌트 준비하기

- Users.js, Blue.js, Red.js, Menu.js, Blue.css, Red.css는 20.1.ssr의 소소 복사

components/Users.js

```

import { useEffect } from 'react';
import Users from '../components/Users';
import { useDispatch, useSelector } from 'react-redux';
import { getUsers } from '../modules/users';

const UsersContainer = () => {
  const users = useSelector((state) => state.users.users);
  const dispatch = useDispatch();

  // 컴퍼넌트가 마운트된 후 호출
  useEffect(() => {
    if(users) return; // users가 이미 유효하다면 요청안함
    dispatch(getUsers());
  }, [dispatch, users]);

  return <Users users={users} />
};

export default UsersContainer;

```

- SSR을 할 때는 이미 있는 정보를 재요청하지 않도록 처리하는게 중요하다.
- 이 작업을 하지 않으면 SSR후 브라우저에서 확인할 때 불필요한 API를 호출하게 된다.'
- pages폴더에 RedPage.js BluePage.js를 20.1.ssr의 소소 복사

pages/UsersPage.js

```

import React from 'react';
import UsersContainer from '../containers/UsersContainer';

const UsersPage = () => {
  return <UsersContainer />;
};

export default UsersPage;

```

App.js

- 20.1.ssr의 소소 복사후 수정

```

import { Route, Routes } from 'react-router-dom';
import Menu from './components/Menu';
import RedPage from './pages/RedPage';

```

```

import BluePage from './pages/BluePage';
import UsersPage from './pages/UsersPage';

function App() {
  return (
    <div>
      <Menu />
      <hr />
      <Routes>
        <Route path="/red" element={<RedPage />} />
        <Route path="/blue" element={<BluePage />} />
        <Route path="/users/*" element={<UsersPage />} />
      </Routes>
    </div>
  );
}

export default App;

```

components/Menu.js

- 20.1.ssr의 소소 복사후 수정

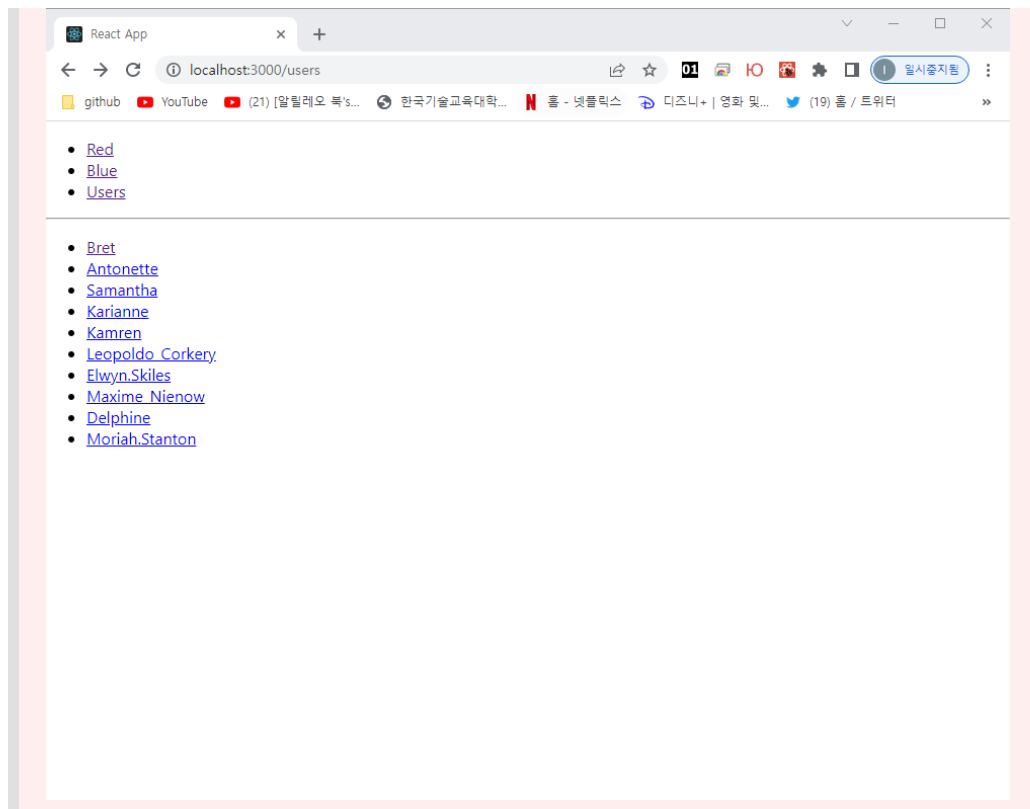
```

import { Link } from 'react-router-dom';
const Menu = () => {
  return (
    <ul>
      <li>
        <Link to="/red">Red</Link>
      </li>
      <li>
        <Link to="/blue">Blue</Link>
      </li>
      <li>
        <Link to="/users">Users</Link>
      </li>
    </ul>
  );
};

export default Menu;

```

중간테스트 : yarn start



20.4.3 PreloadContext만들기

- SSR에서는 useEffect나 componentDidMount에서 설정한 작업이 호출되지 않는다.
- 랜더링하기 전에 API를 요청한 후에 스토어에 데이터를 담아야 한다.
- 서버환경 이런 작업은 클래스형 컴퍼넌트의 constructor에서든 사용하거나 render함수에서 해결 해야 한다.
- 그리고 요청이 끝날 때까지 대기했다가 다시 랜더링을 해주어야 한다.
- 이 작업을 하기 위해 PreloadContext를 만들고 이를 사용하는 Preloader컴퍼넌트를 만들어 처리 한다.

src/lib/PreloadContext.js

```
import { createContext, useContext } from 'react';

// 클라이언트 환경: null
// 서버 환경:{ done: false, promises: [] }
const PreloadContext = createContext(null);
export default PreloadContext;

// resolve는 함수 타입입니다.
export const Preloader = ({ resolve }) => {
  const preloadContext = useContext(PreloadContext);
  if (!preloadContext) return null; // context 값이 유효하지 않다면 아무것도 하지 않음
  if (preloadContext.done) return null; // 이미 작업이 끝났다면 아무것도 하지 않음

  // promises 배열에 프로미스 등록
  // 설정 resolve 함수가 프로미스를 반환하지 않더라도, 프로미스 취급을 하기 위하여
  // Promise.resolve 함수 사용
  preloadContext.promises.push(Promise.resolve(resolve()));
}
```

```
    return null;
  };
}
```

- PreloadContext는 SSR과정에서 처리해야할 작업을 실행하고 기다리는 promiss가 있다면 수집한다.
- 모든 프로미스를 수집한 뒤, 수집된 프로미스들의 종료까지 대기후 재랜더링을 하면 데이터포함한 컴퍼넌트가 나타나게 된다.
- Preload컴퍼넌트는 resolve함수를 props로 받아 오며, 컴퍼넌트가 랜더링될 때 서버환경에서만 resolve를 호출
- 이 컴퍼넌트를 사용하기 위해 UsersContainer에서 정의

containers/UsersContainer.js

```
import { useEffect } from 'react';
import Users from '../components/Users';
import { useDispatch, useSelector } from 'react-redux';
import { getUsers } from '../modules/users';
import { Preloader } from '../lib/PreloadContext'

const UsersContainer = () => {
  const users = useSelector((state) => state.users.users);
  const dispatch = useDispatch();

  // 컴퍼넌트가 마운트된 후 호출
  useEffect(() => {
    if(users) return; // users가 이미 유효하다면 요청안함
    dispatch(getUsers());
  }, [dispatch, users]);

  return (
    <>
    <Users users={users} />
    <Preloader resolve={() => dispatch(getUsers)} />
    </>
  );
};

export default UsersContainer;
```

20.4.4 서버에서 redux설정 및 PreloadContext사용하기

src/index.server.js

- 브라우저에서 할 때와 동일하지만 주의할 점은 서버가 실행될 때 요청이 들어올 때 마다 새로운 store를 만든다.

```
import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';
import fs from 'fs';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
```

```

import rootReducer from './modules';

// asset-manifest.json에서 파일경로들을 조회
const manifest = JSON.parse(
  fs.readFileSync(path.resolve('./build/asset-manifest.json'), 'utf-8')
);

function createPage(root, tags) {
  return `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width,initial-scale=1,shrink-to-fit=no"
  />
  <meta name="theme-color" content="#000000" />
  <title>React App</title>
  <link href="${manifest.files['main.css']}" rel="stylesheet" />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root">${root}</div>
  <script src="${manifest.files['main.js']}"></script>
</body>
</html>
`;
}

const app = express();

// 서버 사이드 랜더링을 처리할 핸들러 함수
const serverRender = (req, res, next) => {
  const context = {};
  const store = createStore(rootReducer, applyMiddleware(thunk));
  const jsx = (
    <Provider store={store}>
      <StaticRouter location={req.url} context={context}>
        <App />
      </StaticRouter>
    </Provider>
  );
  const root = ReactDOMServer.renderToString(jsx); // 랜더링을 하고
  // res.send(root); // 클라이언트에게 결과를 리턴
  res.send(createPage(root)); // 클라이언트에게 결과를 리턴
};

const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});

app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);

// 5000번 포트로 서버를 가동
app.listen(5000, () => {

```

```
console.log('Running on http://localhost:5000');
});
```

src/index.server.js - PreloadContext를 사용하여 promiss수집 및 재 랜더링작업

- 첫 번째 랜더링할 때는 renderToString 대신에 renderToStaticMarkup함수를 사용
- renderToStaticMarkup은 리액트를 사용하여 정적인 페이지를 만들 때 사용
- 현 단계에서 renderToStaticMarkup를 사용한 이유는 Preloader의 함수를 호출하기 위해서이다.
- 또 이 함수의 처리속도가 renderToString보다 좀 더 빠르기 때문이다.

```
import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';
import fs from 'fs';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer from './modules';
import PreloadContext from './lib/PreloadContext';

// asset-manifest.json에서 파일경로들을 조회
const manifest = JSON.parse(
  fs.readFileSync(path.resolve('./build/asset-manifest.json'), 'utf-8')
);

function createPage(root, tags) {
  return `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width,initial-scale=1,shrink-to-fit=no"
  />
  <meta name="theme-color" content="#000000" />
  <title>React App</title>
  <link href="${manifest.files['main.css']}" rel="stylesheet" />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root">${root}</div>
  <script src="${manifest.files['main.js']}"></script>
</body>
</html>
`;
}

const app = express();

// 서버 사이드 랜더링을 처리할 핸들러 함수
const serverRender = async (req, res, next) => {
```

```

const context = {};

const store = createStore(rootReducer, applyMiddleware(thunk));

const preloadContext = {
  done: false,
  promises: [],
};

const jsx = (
  <PreloadContext.Provider value={preloadContext}>
    <Provider store={store}>
      <StaticRouter location={req.url} context={context}>
        <App />
      </StaticRouter>
    </Provider>
  </PreloadContext.Provider>
);

ReactDOMServer.renderToStaticMarkup(jsx); // renderToStaticMarkup으로 한번 렌더링합니다.

try {
  await Promise.all(preloadContext.promises); // 모든 프로미스를 기다립니다.
} catch (e) {
  return res.status(500);
}
preloadContext.done = true;

const root = ReactDOMServer.renderToString(jsx);
res.send(createPage(root)); // 클라이언트에게 결과를 리턴
};

const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});

app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);

// 5000번 포트로 서버를 가동
app.listen(5000, () => {
  console.log('Running on http://localhost:5000');
});

```

20.4.5 스크립트로 store초기상태 주입하기

- 현재 까지는 API를 통해 받아온 데이터를 렌더링하지만 그 과정에서의 store상태를 브라우저에서 재사용하지 못하는 상황이다.
- 서버에서 만들어준 상태를 브라우저에서 재사용하려면 현재 스토어상태를 문자열로 반환한 뒤 스크립트로 주입해 주어야 한다.

src/index.server.js - 스크립트로 store초기상태 주입

```

import ReactDOMServer from 'react-dom/server';
import express from 'express';

```

```

import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';
import fs from 'fs'
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer from './modules';
import PreloadContext from './lib/PreloadContext';

// asset-manifest.json에서 파일경로들을 조회
const manifest = JSON.parse(
  fs.readFileSync(path.resolve('./build/asset-manifest.json'), 'utf-8')
);

function createPage(root, stateScript) {
  return `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width,initial-scale=1,shrink-to-fit=no"
  />
  <meta name="theme-color" content="#000000" />
  <title>React App</title>
  <link href="${manifest.files['main.css']}" rel="stylesheet" />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root">${root}</div>
  ${stateScript}
  <script src="${manifest.files['main.js']}"></script>
</body>
</html>
`;
}

const app = express();

// 서버 사이드 랜더링을 처리할 핸들러 함수
const serverRender = async (req, res, next) => {

  const context = {};

  const store = createStore(rootReducer, applyMiddleware(thunk));

  const preloadContext = {
    done: false,
    promises: [],
  };

  const jsx = (
    <PreloadContext.Provider value={preloadContext}>
      <Provider store={store}>
        <StaticRouter location={req.url} context={context}>

```



```

    <App />
  </StaticRouter>
</Provider>
</PreloadContext.Provider>
);

ReactDOMServer.renderToStaticMarkup(js); // renderToStaticMarkup으로 한번 렌
더링합니다.

try {
  await Promise.all(preloadContext.promises); // 모든 프로미스를 기다립니다.
} catch (e) {
  return res.status(500);
}
preloadContext.done = true;

const root = ReactDOMServer.renderToString(js); // 렌더링을 한다.

// JSON 을 문자열로 변환하고 악성스크립트가 실행되는것을 방지하기 위해서 < 를
치환처리
// https://redux.js.org/recipes/server-rendering#security-considerations
const stateString = JSON.stringify(store.getState()).replace(/</g,
'\\u003c');
const stateScript = `<script>__PRELOADED_STATE__ = ${stateString}</script>`;
// 리덕스 초기 상태를 스크립트로 주입합니다.

res.send(createPage(root, stateScript)); // 클라이언트에게 결과물을 응답합니
다.
};

const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});

app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);

// 5000번 포트로 서버를 가동
app.listen(5000, () => {
  console.log('Running on http://localhost:5000');
});

```

- 브라우저에서 상태를 재사용할 때는 스토어생성과정에서
window.__PRELOADED_STATE__ 를 초기값으로 사용

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { BrowserRouter } from 'react-router-dom';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer from './modules';

```

```
const store = createStore(
  rootReducer,
  window.__PRELOADED_STATE__, // 이 값을 초기상태로 사용
  applyMiddleware(thunk)
);

const root = ReactDOM.createRoot(document.getElementById('root'));

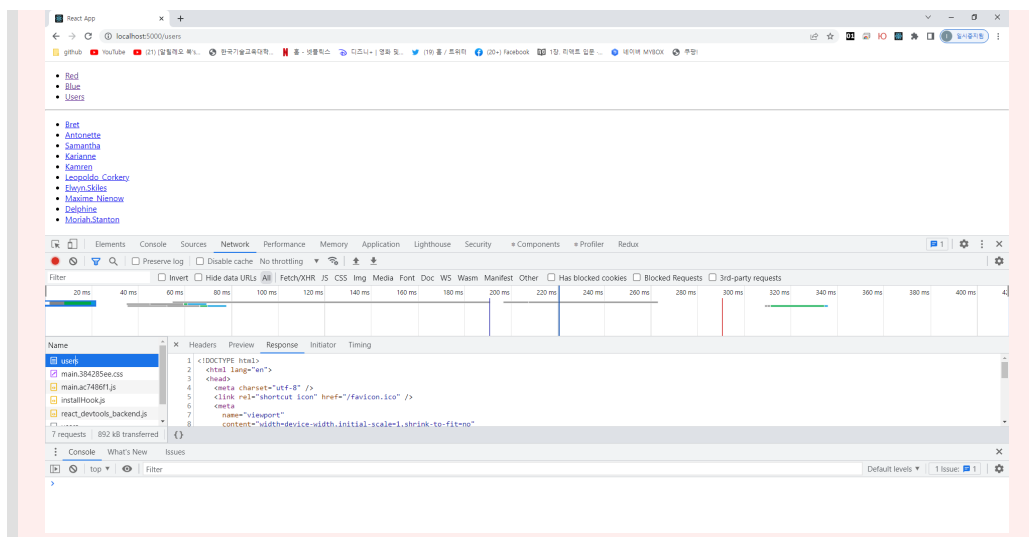
root.render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
);
```

서버재실행 후 확인하기

- dist, scripts, config를 20.1.ssr에서 복사
- package.json 확인

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "start:server": "node dist/server.js",
  "build:server": "node scripts/build.server.js"
},
```

- yarn build
- yarn build:server
- yarn strart:server
- <http://localhost:5000>



20.4.6 redux-saga 코드 준비하기

- <https://kyounghwan01.github.io/blog/React/redux/redux-saga/#사용하는-이유>
 - saga에서 사용하는 자바스크립트 문법 제너레이터함수 만드는 법
 - ES6의 제너레이트 함수문법은 제너레이터함수를 만들 때는

- 함수에 (``function``) 를 붙이고, `yield`라는 문법을 사용한다
- `next()`를 이용하여 다음 `yield`를 호출 한다
- 20.2.`redux_thunk`를 20.3.`redux_sega`로 전부 복사
- 라이브러리 설치
 - `yarn add redux-saga`
- `users`리덕스모듈에서 `redux-saga`를 이용해서 특정 사용자정보를 가져오기

modules/users.js

```
import axios from 'axios';
import { call, put, takeEvery } from 'redux-saga/effects';

const GET_USERS_PENDING = 'users/GET_USERS_PENDING';
const GET_USERS_SUCCESS = 'users/GET_USERS_SUCCESS';
const GET_USERS_FAILURE = 'users/GET_USERS_FAILURE';

const GET_USER = 'users/GET_USER';
const GET_USER_SUCCESS = 'users/GET_USER_SUCCESS';
const GET_USER_FAILURE = 'users/GET_USER_FAILURE';

const getUsersPending = () => ({ type: GET_USERS_PENDING });
const getUsersSuccess = payload => ({ type: GET_USERS_SUCCESS, payload });
const getUsersFailure = payload => ({
  type: GET_USERS_FAILURE,
  error: true,
  payload
});

export const getUser = id => ({ type: GET_USER, payload: id });
const getUserSuccess = data => ({ type: GET_USER_SUCCESS, payload: data });
const getUserFailure = error => ({
  type: GET_USER_FAILURE,
  payload: error,
  error: true
});

export const getUsers = () => async dispatch => {
  try {
    dispatch(getUsersPending());
    const response = await axios.get(
      'https://jsonplaceholder.typicode.com/users'
    );
    dispatch(getUsersSuccess(response));
  } catch (e) {
    dispatch(getUsersFailure(e));
    throw e;
  }
};

const initialState = {
  users: null,
  user: null,
  loading: {
    users: false,
    user: false
  }
};
```

```

    },
    error: {
      users: null,
      user: null
    }
  }
};

function users(state = initialState, action) {
  switch (action.type) {
    case GET_USERS_PENDING:
      return {
        ...state,
        loading: { ...state.loading, users: true }
      };
    case GET_USERS_SUCCESS:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        users: action.payload.data
      };
    case GET_USERS_FAILURE:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        error: { ...state.error, users: action.payload }
      };
    case GET_USER:
      return {
        ...state,
        loading: { ...state.loading, user: true },
        error: { ...state.error, user: null }
      };
    case GET_USER_SUCCESS:
      return {
        ...state,
        loading: { ...state.loading, user: false },
        user: action.payload
      };
    case GET_USER_FAILURE:
      return {
        ...state,
        loading: { ...state.loading, user: false },
        error: { ...state.error, user: action.payload }
      };
  }
}

import axios from 'axios';
import { call, put, takeEvery } from 'redux-saga/effects';

const GET_USERS_PENDING = 'users/GET_USERS_PENDING';
const GET_USERS_SUCCESS = 'users/GET_USERS_SUCCESS';
const GET_USERS_FAILURE = 'users/GET_USERS_FAILURE';

const GET_USER = 'users/GET_USER';
const GET_USER_SUCCESS = 'users/GET_USER_SUCCESS';
const GET_USER_FAILURE = 'users/GET_USER_FAILURE';

const getUsersPending = () => ({ type: GET_USERS_PENDING });
const getUsersSuccess = payload => ({ type: GET_USERS_SUCCESS, payload });
const getUsersFailure = payload => ({
  type: GET_USERS_FAILURE,
  error: true,
  payload
});

```

```

export const getUser = id => ({ type: GET_USER, payload: id });
const getUserSuccess = data => ({ type: GET_USER_SUCCESS, payload: data });
const getUserFailure = error => ({
  type: GET_USER_FAILURE,
  payload: error,
  error: true
});

export const getUsers = () => async dispatch => {
  try {
    dispatch(getUsersPending());
    const response = await axios.get(
      'https://jsonplaceholder.typicode.com/users'
    );
    dispatch(getUsersSuccess(response));
  } catch (e) {
    dispatch(getUsersFailure(e));
    throw e;
  }
};

const getUserById = id =>
  axios.get(`https://jsonplaceholder.typicode.com/users/${id}`);

function* getUserSaga(action) {
  try {
    const response = yield call(getUserById, action.payload);
    yield put(getUserSuccess(response.data));
  } catch (e) {
    yield put(getUserFailure(e));
  }
}

export function* usersSaga() {
  yield takeEvery(GET_USER, getUserSaga);
}

const initialState = {
  users: null,
  user: null,
  loading: {
    users: false,
    user: false
  },
  error: {
    users: null,
    user: null
  }
};

function users(state = initialState, action) {
  switch (action.type) {
    case GET_USERS_PENDING:
      return {
        ...state,
        loading: { ...state.loading, users: true }
      }
  }
}

```

```

    };
    case GET_USERS_SUCCESS:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        users: action.payload.data
      };
    case GET_USERS_FAILURE:
      return {
        ...state,
        loading: { ...state.loading, users: false },
        error: { ...state.error, users: action.payload }
      };
    case GET_USER:
      return {
        ...state,
        loading: { ...state.loading, user: true },
        error: { ...state.error, user: null }
      };
    case GET_USER_SUCCESS:
      return {
        ...state,
        loading: { ...state.loading, user: false },
        user: action.payload
      };
    case GET_USER_FAILURE:
      return {
        ...state,
        loading: { ...state.loading, user: false },
        error: { ...state.error, user: action.payload }
      };
    default:
      return state;
  }
}

```

```
export default users;
```

modules/index.js

- 리덕스 스토어에 redux-saga를 적용
- rootSaga를 작성

```

import { combineReducers } from 'redux';
import users, { usersSaga } from './users';
import { all } from 'redux-saga/effects';

export function* rootSaga() {
  yield all([usersSaga()]);
}

const rootReducer = combineReducers({ users });

export default rootReducer;

```

src/index.js

- store를 생성할 때 미들웨어를 적ㅇㅇ

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { BrowserRouter } from 'react-router-dom';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import rootReducer, { rootSaga } from './modules';
import createSagaMiddleware from 'redux-saga';

const sagaMiddleware = createSagaMiddleware();

const store = createStore(
  rootReducer,
  window.__PRELOADED_STATE__, // 이 값을 초기상태로 사용
  applyMiddleware(thunk, sagaMiddleware)
);

sagaMiddleware.run(rootSaga);

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
);
```

20.4.7 User, UserContainer 컴퍼넌트 준비하기

- 특정 사용자의 정보를 보여줄 User 컴퍼넌트 작성하기

components/User.js

```
import React from 'react';

const User = ({ user }) => {
  const { email, name, username } = user;
  return (
    <div>
      <h1>
        {username} ({name})
      </h1>
      <p>
        <b>e-mail:</b> {email}
      </p>
    </div>
  );
};

export default User;
```

- User컴퍼넌트에서는 user값이 null인지 객체인지 유효성검사를 해주지 않았다.
- 컨테이너 컴퍼넌트에서 유효성 검사하기
 - API요청시 id값을 props를 통해 전달 받기
 - connect함수를 사용하지 않고 `useSelector`와 `useDispatch` Hooks를 사용

containers/UserContainer.js

- 컨테이너에서 유효성검사를 할 때 정보가 없는 경우에 user값이 null이기 때문에 User컴퍼넌트가 렌더링되지 않도록 컨테이너 컴퍼넌트에 null을 반환해야 한다.
- 하지만 SSR을 해야 하기 때문에 null이 아닌 Preloader컴퍼넌트를 렌더링하여 리턴
- 이럴 경우 SSR과정에서 데이터가 없을 경우에 GET_USER액션을 발생시킨다.

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import User from '../components/User';
import { Preloader } from '../lib/PreloadContext';
import { getUser } from '../modules/users';

const UserContainer = ({ id }) => {
  const user = useSelector(state => state.users.user);
  const dispatch = useDispatch();

  useEffect(() => {
    if (user && user.id === parseInt(id, 10)) return; // 유저가 존재하고, id가
    일치한다면 요청하지 않음
    dispatch(getUser(id));
  }, [dispatch, id, user]); // id가 바뀔 때 새로 요청해야 함

  // 컨테이너 유효성 검사후 return null을 해야하는 경우에 null대신 Preloader반
  환
  if (!user) {
    return <Preloader resolve={() => dispatch(getUser(id))} />;
  }
  return <User user={user} />;
};

export default UserContainer;
```

pages/UserPage.js

- 컨테이너컴퍼넌트 완성후에 UserPage를 작성

```
import { useParams } from 'react-router-dom';
import UserContainer from '../containers/UserContainer';

const UserPage = () => {
  const { id } = useParams();
  return <UserContainer id={id} />;
};

export default UserPage;
```

pages/UsersPage.js

```
import React from 'react';
import { Route, Routes } from 'react-router-dom';
import UsersContainer from '../containers/UsersContainer';
```

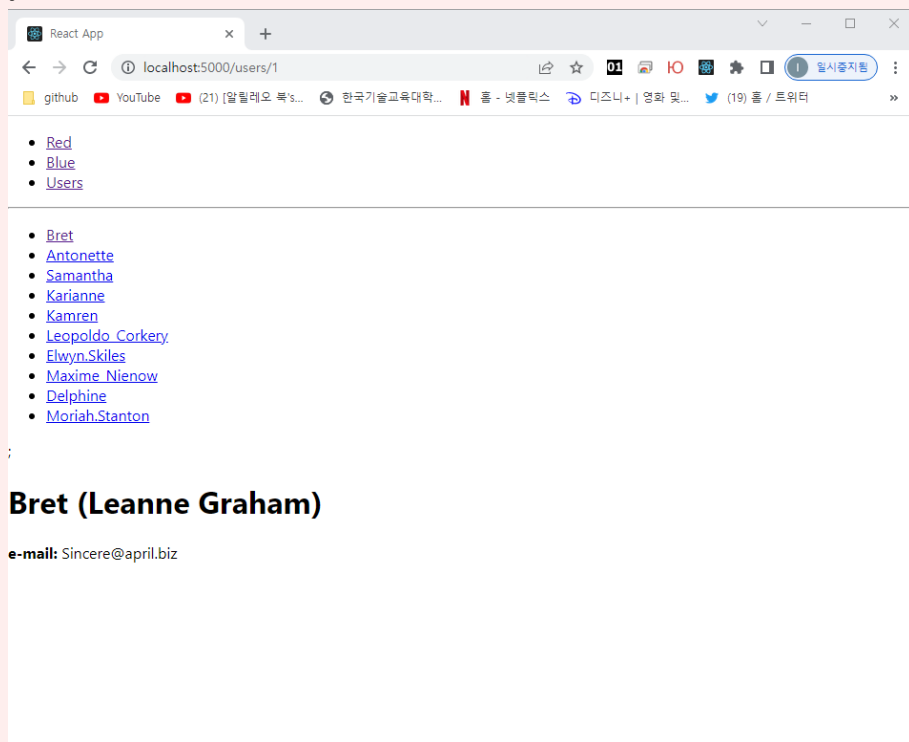


```
import UserPage from './UserPage';

const UsersPage = () => {
  return (
    <>
    <UsersContainer />
    <Routes>
    <Route path=':id' element={ <UserPage /> } />
    </Routes>
    </>
  );
};

export default UsersPage;
```

- yarn build
- yarn build:server
- yarn start:server



20.4.8 redux-saga를 위한 SSR 작업

- redux-thunk를 사용하면 Preloader를 통해 호출한 함수들이 Promise를 반환하지만, redux-saga를 사용하면 Promise를 반환하지 않기 때문에 추가작업이 필요하다.
- redux-saga 미들웨어 적용하기

src/index.server.jse

- toPromise는 sagaMiddleware.run을 통해 만든 taskfn Promise로 변환한다.
- 별도의 작업을 하지 않으면 이 Promise는 rootSaga에서 액션을 끊임없이 모니터링 하기 때문에 끝나지 않는다.
- 그러나 redux-saga의 END액션을 발생시키면 이 Promise를 끝낼 수 있다.
- END액션이 발생되면 액션모니터링작업이 모두 종료되고 모니터링되기 전에 시작된 getUserSaga와 같은 사가함수들이 있다면 해당 함수들이 완료된 후 Promise가 끝나게 된다.

- 이 Promise가 끝나는 시점에 리더스 스토어에는 우리가 원하는 데이터가 채워진다. 그 후에 다시 렌더링하면 원하는 결과물이 나타난다.

```
import ReactDOMServer from 'react-dom/server';
import express from 'express';
import { StaticRouter } from 'react-router-dom/server';
import App from './App';
import path from 'path';
import fs from 'fs';
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';
import PreloadContext from './lib/PreloadContext';
import createSagaMiddleware from 'redux-saga';
import rootReducer, { rootSaga } from './modules';
import { END } from 'redux-saga';

// asset-manifest.json에서 파일경로들을 조회
const manifest = JSON.parse(
  fs.readFileSync(path.resolve('./build/asset-manifest.json'), 'utf-8')
);

function createPage(root, stateScript) {
  return `<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="shortcut icon" href="/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width,initial-scale=1,shrink-to-fit=no"
  />
  <meta name="theme-color" content="#000000" />
  <title>React App</title>
  <link href="${manifest.files['main.css']}" rel="stylesheet" />
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root">${root}</div>
  ${stateScript}
  <script src="${manifest.files['main.js']}"></script>
</body>
</html>
`;
}

const app = express();

// 서버 사이드 렌더링을 처리할 핸들러 함수
const serverRender = async (req, res, next) => {

  const context = {};
  const sagaMiddleware = createSagaMiddleware();

  const store = createStore(
    rootReducer,
```

```

    applyMiddleware(thunk, sagaMiddleware)
  );

  const sagaPromise = sagaMiddleware.run(rootSaga).toPromise();

  const preloadContext = {
    done: false,
    promises: [],
  };

  const jsx = (
    <PreloadContext.Provider value={preloadContext}>
      <Provider store={store}>
        <StaticRouter location={req.url} context={context}>
          <App />
        </StaticRouter>
      </Provider>
    </PreloadContext.Provider>
  );

  ReactDOMServer.renderToStaticMarkup(jsx); // renderToStaticMarkup으로 한번 렌
  더링합니다.
  store.dispatch(END); // redux-saga 의 END 액션을 발생시키면 액션을 모니터링하
  는 saga 들이 모두 종료됩니다.

  try {
    await sagaPromise; // 기존에 진행중이던 saga 들이 모두 끝날때까지 기다립니
    다.
    await Promise.all(preloadContext.promises); // 모든 프로미스를 기다립니다.
  } catch (e) {
    return res.status(500);
  }
  preloadContext.done = true;

  const root = ReactDOMServer.renderToString(jsx); // 렌더링을 한다.

  // JSON 을 문자열로 변환하고 악성스크립트가 실행되는것을 방지하기 위해서 < 를
  치환처리
  // https://redux.js.org/recipes/server-rendering#security-considerations
  const stateString = JSON.stringify(store.getState()).replace(/</g,
  '\u003c');
  const stateScript = `<script>__PRELOADED_STATE__ = ${stateString}</script>`;
  // 리덕스 초기 상태를 스크립트로 주입합니다.

  res.send(createPage(root, stateScript)); // 클라이언트에게 결과물을 응답합니
  다.
};

const serve = express.static(path.resolve('./build'), {
  index: false // "/"에서 index.html을 보여주지 않도록 설정
});

app.use(serve); // 순서가 중요!!, serverRender전에 위치해야 한다.
app.use(serverRender);

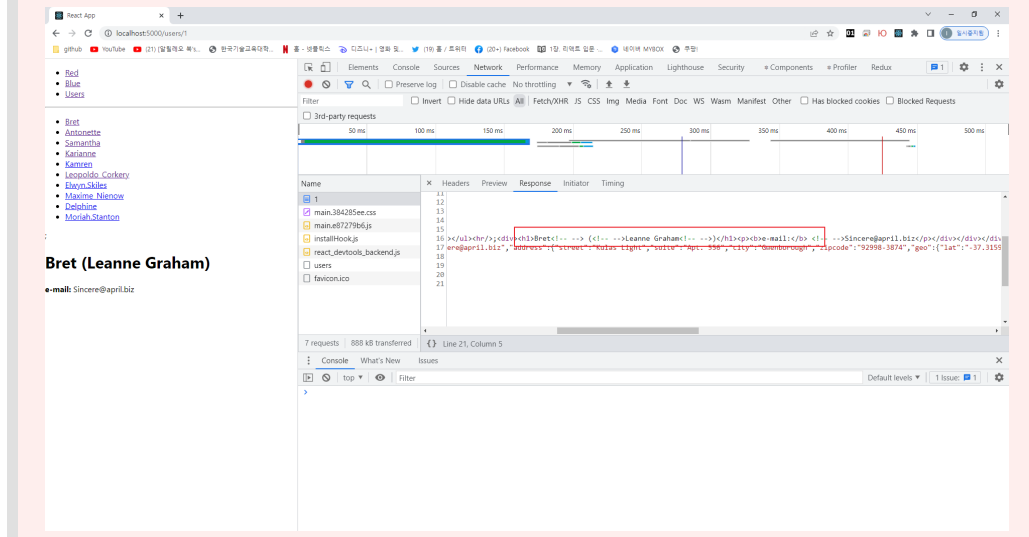
// 5000번 포트로 서버를 가동
app.listen(5000, () => {

```

```
console.log('Running on http://localhost:5000');
});
```

서버 rebuild후 재시작

- yarn build
- yarn build:server
- yarn start:server



20.4.9 usePreloader Hook 만들어 사용하기

- 지금까지는 Preloader컴퍼넌트를 사용해야 SSR을 하기전에 필요한 상황에 API를 요청
- 이번에는 usePreloader 커스텀 Hook을 만들어 더 편리하게 처리하는 작업을 진행

lib/usePreloadContext.js

```
import { createContext, useContext } from 'react';
```

```
// 클라이언트 환경: null
// 서버 환경:{ done: false, promises: [] }
const PreloadContext = createContext(null);
export default PreloadContext;
```

// resolve는 함수 타입입니다.

```
export const Preloader = ({ resolve }) => {
  const preloadContext = useContext(PreloadContext);
  if (!preloadContext) return null; // context 값이 유효하지 않다면 아무것도 하지 않음
  if (preloadContext.done) return null; // 이미 작업이 끝났다면 아무것도 하지 않음
```

```
  // promises 배열에 프로미스 등록
  // 설정 resolve 함수가 프로미스를 반환하지 않더라도, 프로미스 취급을 하기 위하여
  // Promise.resolve 함수 사용
  preloadContext.promises.push(Promise.resolve(resolve()));
  return null;
};
```

// Hook 형태로 사용 할 수 있는 함수

```
export const usePreloader = resolve => {
  const preloadContext = useContext(PreloadContext);
  if (!preloadContext) return null;
  if (preloadContext.done) return null;

  // promises 배열에 프로미스 등록
  // 설정 resolve 함수가 프로미스를 반환하지 않더라도, 프로미스 취급을 하기 위
  하여
  // Promise.resolve 함수 사용
  preloadContext.promises.push(Promise.resolve(resolve()));
};
```

containers/UserContainer.js

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import User from '../components/User';
import { usePreloader } from '../lib/PreloadContext';
import { getUser } from '../modules/users';

const UserContainer = ({ id }) => {
  const user = useSelector(state => state.users.user);
  const dispatch = useDispatch();

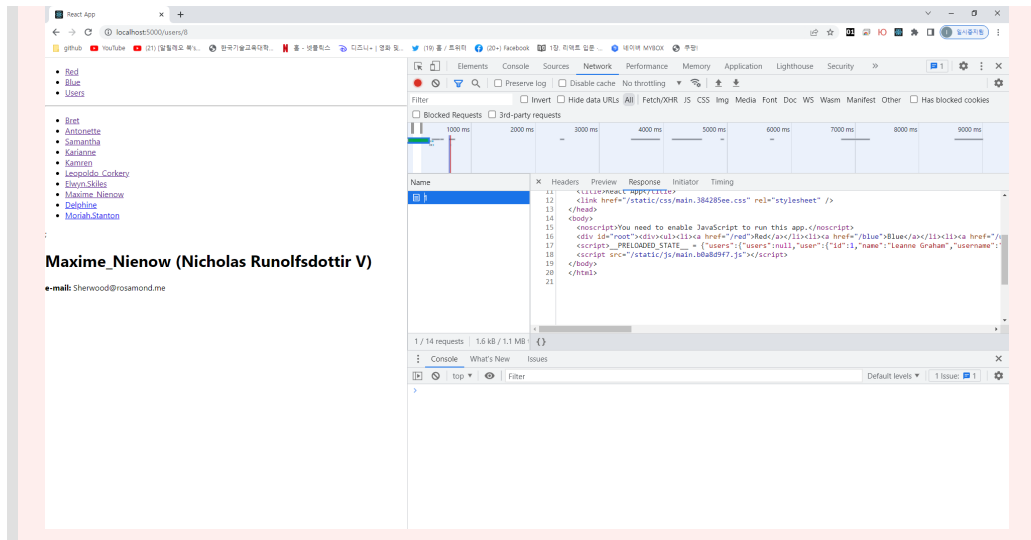
  usePreloader(() => dispatch(getUser(id))); // 서버 사이드 렌더링 할 때 API 호
  출하기
  useEffect(() => {
    if (user && user.id === parseInt(id, 10)) return; // 유저가 존재하고, id가
    일치한다면 요청하지 않음
    dispatch(getUser(id));
  }, [dispatch, id, user]); // id가 바뀔 때 새로 요청해야 함

  if (!user) return null;
  return <User user={user} />;
};

export default UserContainer;
```

- 코드가 훨씬 간결해 졌다.
- 함수 컴퍼넌트에서는 usePreloader Hook을 사용하고 클래스형 컴퍼넌트에서는 Preloader컴퍼넌트를 사용
- 서버 rebuild후 재시작

```
▪ yarn build
▪ yarn build:server
▪ yarn start:server
```



코드스플리팅은 복잡한 작업이기 때문에 강의는 스킵할 것

20.5 SSR과 코드 스플리팅

- 리액트 공식제공하는 React.lazy와 Suspense는 SSR을 아직 지원하지 않는다.
- 현재(2019.4월) SSR과 코드 스플리팅을 함께 사용할 때는 Loadable Components를 사용할 것을 권장 하고 있다.
- Loadable Components를 설치
 - `yarn add @loadable/component @loadable/server @loadable/webpack-plugin @loadable/babel-plugin`

20.5.1 라우트 컴퍼넌트 스플리팅하기

- BluePage, RedPage, UserPage 스플리팅

src/App.js


```
import { Route, Routes } from 'react-router-dom';
import Menu from './components/Menu';
import loadable from '@loadable/component'

const RedPage = loadable(() => import('./pages/RedPage'));
const BluePage = loadable(() => import('./pages/BluePage'));
const UsersPage = loadable(() => import('./pages/UsersPage'));

function App() {
  return (
    <div>
      <Menu />
      <hr />
      <Routes>
        <Route path="/red" element={<RedPage />} />
        <Route path="/blue" element={<BluePage />} />
        <Route path="/users/*" element={<UsersPage />} />
      </Routes>
    </div>
  );
}
```

```
export default App;
```

- 서버 rebuild후 재시작

- yarn build
- yarn build:server
- yarn start:server
- Running on <http://localhost:5000/users/1> 

20.5.2 웹팩과 babel plugin 적용

- 네트워크환경이 느릴 경우에는 깜박임현상이 나타나는데 이를 해결하기 위해 웹팩과 babel플러그인을 적용하면 해결
 - 깜박임을 확인하려면 크롬개발자도구 Network탭에서 인터넷속도를 Slow 3G로 설정 후 localhost:5000/user/1 실행
- babel plugin 적용
 - package.json수정

```
{
  "name": "20.ssr_dataloading",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@loadable/babel-plugin": "^5.15.3",
    "@loadable/component": "^5.15.3",
    "@loadable/server": "^5.15.3",
    "@loadable/webpack-plugin": "^5.15.2",
    "@testing-library/jest-dom": "^5.14.1",
    "@testing-library/react": "^13.0.0",
    "@testing-library/user-event": "^13.2.1",
    "axios": "^1.3.5",
    "express": "^4.18.2",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-redux": "^8.0.5",
    "react-router-dom": "^6.10.0",
    "react-scripts": "5.0.1",
    "redux": "^4.2.1",
    "redux-saga": "^1.2.3",
    "redux-thunk": "^2.4.2",
    "web-vitals": "^2.1.0",
    "webpack-node-externals": "^3.0.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "start:server": "node dist/server.js",
    "build:server": "node scripts/build.server.js"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
```

```

    "react-app/jest"
  ],
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "babel": {
    "presets": [
      "react-app"
    ],
    "plugins": [
      "@loadable/babel-plugin"
    ]
  }
}

```

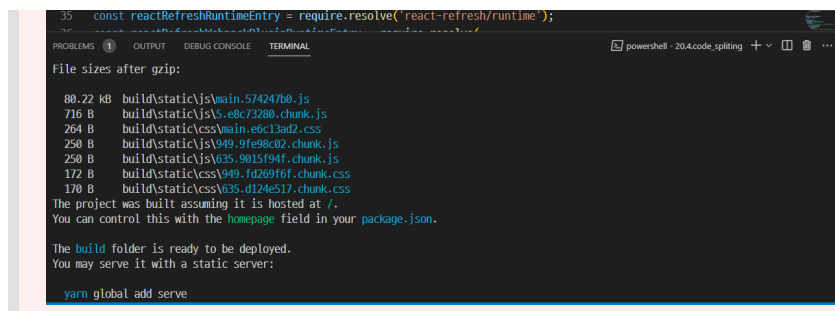
- config/webpack.config.js 수정 (new LoadablePlugin(), 추가)

```

const LoadablePlugin = require('@loadable/webpack-plugin');
( ... )
plugins: [
  new LoadablePlugin(),
  // Generates an `index.html` file with the <script> injected.
  new HtmlWebpackPlugin(
    Object.assign(
      {},
    ( ... )

```

- yarn build 실행후 build/loadable-stats.json 파일 생성확인
- 이 파일은 각 컴퍼넌트코드가 어떤 청크(chunk)파일에 들어가 있는지에 대한 정보를 가지고 있다.
- 파일생성은 안되고 yarn build후에 아래 메시지 표시
 - build/static 폴더와 asset-manifest.json 재생성이 된다.



loadable-stats.json 파일 생성이 되지 않아 이 후작업은 실행할 수 없음

20.5.3 필요한 chunk파일 경로 추출하기

- 해당 파일들의 경로를 추출하기 위해 Loadable Components에서 제공하는 `ChunkExtractor`와 `ChunkExtractorManager` 를 사용
- Loadable Components을 통해 파일경로를 조회하므로 기존에 `asset-manifest.json`을 확인하던 코드는 지워 준다.