Next.js

SSR(Server Side Rendering)을 구현하기 위한 React.js 기반의 프레임워크

#01. 프로젝트 구성

1) 프로젝트 생성하기

```
yarn create next-app 프로젝트이름
```

2) 프로젝트 가동하기

```
yarn dev
```

react.js는 웹 브라우저가 자동으로 실행되지만 next.js는 브라우저가 실행되지 않기 때문에 직접 브라우저에 URL을 입력하여 접속해야 한다.

```
http://localhost:3000
http://127.0.0.1:3000
http://자신의IP주소:3000
```

3) 패키지 설치

기본적으로 react와 패키지를 공유하지만 styledComponent의 경우 추가 패키지가 필요하다.

```
yarn add styled-components babel-plugin-styled-components
```

#03. 기본 구조

1) URL 구성

`/pages` 폴더가 react 프로젝트의 `/src` 폴더에 대응된다.

별도의 Router 처리 없이 `/pages` 폴더 내의 폴더,파일 구조가 URL로 구성된다.

2) Link 적용하기

`<a>` 태그를 통한 링크

서버로부터 새로운 HTML코드를 받아온다.

새로운 트래픽이 발생한다.

<Link> 컴포넌트를 통한 링크

최초 1회는 서버에 의해 렌더링된 HTML코드를 받아오고 그 이후에는 React의 SPA 방식으로 구동되어 네트워크 트래픽 사용을 최소화 한다.

```
import Link from 'next/link';

<Link href="/">홈</Link>
```

#04. 예약된 파일

특정한 목적으로 구현되는 파일로 파일 이름이 고정되어 있다.

1) /pages/_error.js

404 에러가 발생한 경우 노출될 페이지.

함수형 컴포넌트로 작성되며 함수의 이름은 페이지 구성에 영향을 주지 않는다.

2) /pages/_document.js

모든 next.js 페이지가 동작하기 전에 우선 동작하는 클래스로 styledComponent를 사용하기 위해 필수적으로 정의되어야 한다.

클래스 형태만 지원하며, 이 파일에서 처리하는 구현은 모든 페이지에 적용되는 기본 구성이 된다.

<html><head>...</head><body>...</body></html> 구조를 정의하고, <head>안에 공통적으로 포함될 내용을 명시한다.

```
// HTML 구조를 재정의하기 위한 참조
import Document, { Html, Head, Main, NextScript } from 'next/document';

// styledComponent를 사용하기 위한 참조
import { ServerStyleSheet } from 'styled-components';

class MyDocument extends Document {
    /**
     * 초기화 함수 (고정코드)
     * 이 함수에서 리턴하는 객체를 렌더링 함수 안에서 this.props로 접근한다.
     * 이 함수 안에서 ServerStyleSheet 객체를 사용해서 style객체를 반환해야
     * 컴포넌트 전역에서 styledComponent를 사용할 수 있다.
     */
    static getInitialProps({ renderPage }) {
        // 1) ServerStyleSheet 객체 생성
        const sheet = new ServerStyleSheet();

        // 2) 각 페이지의 컴포넌트에 style가 적용된 결과를 렌더링한 결과 생성
        const page = renderPage(App => props => sheet.collectStyles(<App
```

```
{...props} />));

        // 3) 컴포넌트에 적용된 style을 styleTags라는 이름으로 객체로 반환함
        const styleTags = sheet.getStyleElement();

        // 4) 반환받은 styleTags를 props로 리턴
        return { ...page, styleTags };
    }

    /**
     * 화면 렌더링 함수 -> Html, Head, Main 첫 글자가 대문자임에 주의
     */
    render() {
        return (
            <Html>
                {/*
                    <head>는 순수 HTML태그. <Head>는 next.js의 컴포넌트.
                    이 안에서 charset과 viewport 지정은 자동으로 이루어진다.
                    그 외에 개발자가 적용하고자 하는 외부 CSS나 JS리소스 참조, SEO 구현등
을 처리할 수 있다.
                */}
                <Head>
                    <title>Hello Next.js</title>

                    {/* SEO 메타태그 */}
                    <meta name="description" content="검색결과에 표시될 내용"/>
                    <meta name="robots" content="index,follow" />
                    <meta name="keywords" content="SEO,검색엔진 최적화,메타 태그"
/>
                    <meta name="author" content="leekh" />

                    {/* SNS 메타태그 */}
                    <meta property="og:type" content="website" />
                    <meta property="og:title" content="페이지 제목" />
                    <meta property="og:description" content="페이지 설명" />
                    <meta property="og:image"
content="http://www.mysite.com/myimage.jpg" />
                    <meta property="og:url"
content="http://www.mysite.com" />

                    {/*  getInitialProps에서 리턴한 styleTags를 출력한다. */}
                    {this.props.styleTags}
                </Head>
                <body>
                    {/* 이 구조를 기본으로 적용한 상태에서 일반 페이지용 js들이 이 위치에
출력된다.
                        만약 _app.js가 정의되어 있다면 _app.js의 구조를 먼저 적용한 후
에 페이지가 표시된다. */}
                    <Main />
                    <NextScript />
                </body>
            </Html>
        );
    }
```

```
}

export default MyDocument;
```

3) /pages/_app.js

프로젝트 생성시 기본 제공됨 (내용 추가가 필요함)

모든 페이지들에게 적용되는 공통 컴포넌트.

헤더, 푸터를 구현하는 용도로 사용된다.

클래스 형태로 작성해야 함.

이 페이지에서 return하는 JSX 내용이 _document.js의 <body>안에 표시된다고 이해할 수 있다.

개발자가 작성하는 일반 페이지에서 return 하는 내용은 이 파일에 포함된다.

```
[개발자코드(ex: hello.js)] -----> [_app.js] -----> [_document.js]
```

- 개발자코드 : 실제 페이지 내용을 구현
- _app.js : 개발자 코드의 주변부에 header, footer를 덧붙임
- _document.js : _app.js가 구성한 내용을 <html><head>...</head><body>...</body></html> 구조에 포함시킴

```
import App from 'next/app';

class MyApp extends App {
    /** 초기화함수(고정코드) */
    static async getInitialProps(appContext) {
        // 브라우저가 URL로 접근했을 때,
        // index.js, hello.js, world.js와 같은 일반 페이지 스크립트들을 appContext
로 받는다.
        // 이를 리턴하여 렌더링 함수로 전달해야 한다.
        const appProps = await App.getInitialProps(appContext);
        return { ...appProps };
    }

    render() {
        const { Component, pageProps } = this.props;

        return (
            <div>
                {/* ... 원한다면 Header 구성 */}
                <h1>여기는 Header 입니다.</h1>

                {/* 일반 페이지 컴포넌트를 출력한다. --> index.js, hello.js,
world.js 등 */}
                <Component {...pageProps} />
```

```
                {/* ... 원한다면 Footer 구성 */}
                <h1>여기는 Footer 입니다.</h1>
            </div>
        );
    }
}

export default MyApp;
```

#05. StyledComponent를 적용한 링크

1) 패키지 참조

```
import styled from 'styled-components';
import Link from 'next/link';
```

2) <a> 태그에 대한 StyledComponent 만들기

React.js의 경우 Link나 NavLink 컴포넌트에 대한 StyledComponent 확장을 사용했지만 Next.js는 <a>태그에 대한 StyledComponent를 정의해야 한다.

```
const MenuLink = styled.a`
    ... SCSS 구문 적용 ...
`;
```

3) 렌더링 함수에서 StyledComponent 사용하기

최초 1회는 SSR, 그 이후는 SPA로 동작하는 링크

<Link> 컴포넌트 안에 정의한 StyledComponent를 포함시킨다.

```
<Link href="/world" passHref>
    <MenuLink>World</MenuLink>
</Link>
```

SSR로만 동작하는 링크

<Link> 컴포넌트 없이 순수한 <a>를 사용하거나 StyledComponent만 독립적으로 사용한다.

```
<MenuLink href="/world">World</MenuLink>
```