Contents

[Url’s 2](#_Toc8843515)

[Create project from cli 2](#_Toc8843516)

[Typescript support 2](#_Toc8843517)

[Sass support 3](#_Toc8843518)

[Static files 3](#_Toc8843519)

[Ingredients 3](#_Toc8843520)

[Meta tags 3](#_Toc8843521)

[Routing 3](#_Toc8843522)

[getInitialProps 4](#_Toc8843523)

[Deep learning 4](#_Toc8843524)

# Url’s

//Next.js routing

<https://zeit.co/guides/custom-next-js-server-to-routes>

## domain propagation result

<https://www.whatsmydns.net/#CNAME/www.sushantagarwal.com>

## Process for next.js project’s production deployment in cloudjiffy

* Create new environment
* Activate SSL in new environment
* Manually delete all files and folders in home / jelastic/ROOT folder
* Upload the test.zip. This is zipped file for compiled .next + server.js + static folder + any other folder. In server.js set port to 8080 and **not to port 80.**
* Go to command mode. Webssh. Cd /home/jelastic/ROOT. > unzip test.zip
* Restart environment
* This site will be https enabled

## SEO compliance

**robots.txt for seo**

* Create a text file with name robots.txt with following contents
  + User-agent: \*
  + Disallow:
* In server.js add these lines

const robotsOptions = {

    root: \_\_dirname + '/static/',

    headers: {

        'Content-Type': 'text/plain;charset=UTF-8',

    }

};

server.get('/robots.txt', (req, res) => (

        res.status(200).sendFile('robots.txt', robotsOptions)

     ));

**Include lang attribute in html tag. You need to modify \_document.js in pages folder**.

import Document, { Html, Head, Main, NextScript} from 'next/document'

render() {

return (

<Html lang="en-US">

<Head>

</Head>

<body>

<Main />

<NextScript />

</body>

</Html>

)

}

## CSS frameworks with next.js

I found styled-components library good working with next.js. You can activate VS plugins for intellisense etc.

# Server side rendering with styled-components

In **absence** of server-side rendering for stylesheet by styled-components library, for every server call for a page there **is too much flickering** and too much **re-adjustments** on the screen. It appears that the page first gets loaded without any stylesheet in action then the browser activates the stylesheet and rearranges the items on the screen.

This happens because the page is rendered at server without any stylesheet embedded in it. So, when the page arrives at client then the browser first shows the page without any stylesheet in action then does the rearrangement of controls using the stylesheet. This is too much visual annoyance.

**Remedy**: Just include following \_document.js file in the pages directory and everything will be through.

import Document from 'next/document'

import { ServerStyleSheet } from 'styled-components'

export default class MyDocument extends Document {

static async getInitialProps (ctx) {

const sheet = new ServerStyleSheet()

const originalRenderPage = ctx.renderPage

try {

ctx.renderPage = () =>

originalRenderPage({

enhanceApp: App => props => sheet.collectStyles(<App {...props} />)

})

const initialProps = await Document.getInitialProps(ctx)

return {

...initialProps,

styles: (

<>

{initialProps.styles}

{sheet.getStyleElement()}

</>

)

}

} finally {

sheet.seal()

}

}

}

I also had my .babelrc as follows while it worked, but I am not sure if that is needed.

{

"presets": [

"next/babel",

"@zeit/next-typescript/babel"

],

"plugins": [

// ["babel-plugin-root-import"],

[

"styled-components",

{

"ssr": true,

"displayName": true,

"preprocess": false

}

]

]

}

## Serverless deployment at Zeit Now

I was able to deploy simple functions and express server in Zeit now and it was working fine. I was unable to deploy Next.js website in Zeit now, several errors were coming when trying express server along with next module.

const next = require(‘next’)

const dev = true

const app = next(dev)

app.prepare().then…

The app.prepare did not work with ‘now’. Entire app in there in github as zeit-serverless. See the routes in now.json. In this project I called an express based function to pull contacts from cloudjiffy. It worked fine.

# More on getInitialProps on 17-05-2019

Be informed that when client-side navigation (Link tag) for a page say page1 is clicked and that page exists in pages folder then **no server trip** is made, and page is fed from the client side itself. In that case the getInitialProps code is run only at client side and req property is undefined. If(req) is good check to see whether on client or on server. When getInitialProps is run at client side then any console.log is rendered at browser.

If the page1 does not exist in pages folder then a server trip is made and checked if ‘/page1’ is defined as any express route. If route is found then code for that route is executed. Then req and res objects of getInitialProps have values in them. When you do F5 or forceful page refresh or ‘page1’ is in <a> tag then also server trip is made. You must distinguish code to run at server and at client in getInitialProps and return a valid object. This object returned by getInitialProps is then fed to render method.

## \_app and understanding getInitialProps

See this code snippet

**import** React **from** 'react';

**import** App, { Container } **from** 'next/app';

**class** MyApp **extends** App {

**static** **async** getInitialProps({ Component, ctx }) {

**let** pageProps = {};

**if** (Component.getInitialProps) {

pageProps = **await** Component.getInitialProps(ctx);

}

**return** { pageProps };

}

render() {

**const** { Component, pageProps } = **this**.props;

**return** (

<Container>

<div>Extra markup</div>

<Component {...pageProps} />

</Container>

);

}

}

**export** **default** MyApp;

The getInitialProps code as above is always executed at server side. You can inject any data value in pageProps. That data value will be available as props to all pages components at client. This can be used as global data for all the pages.

Life cycle of getInitialProps when \_app.js as above is put in pages folder. This is only valid at server trip.

* At any given page say page1, if there is getInitialProps() method exists then at first getInitialProps() is executed from \_app.js
* Then page1’s getInitialProps() method is executed
* Then render or main function of the component is executed. Any new properties added by \_app.js’s getInitialProps() is available to the component’s props.
* The getInitialProps() of page does not get any value from \_app.js’s getInitialProps(). But on other hand whatever you return from page’s getInitialProps is obtained in pageProps above.
* Any extra markup you put inside <Container> is set in all the pages.

# Compiled code deployment for next.js

I have taken project adam3 in next/adam3 at github as reference, but any other project will also work. The server.js makes use of “next” module. This module is used to create the app object. App.prepare().then will further create the server object which provides the routing. The next.js module is used for refresh of a url by pressing F5. At that point next.js module is used to create on the fly html from the js files in pages folder. The app object can be created in dev mode (dev=true) and not dev mode (dev=false). In dev mode the pages are compiled to create the **.next** build folder, where compiled code resides. Each time the ‘node server’ is run in dev mode the .next folder is recreated. The server.js is below

const express = **require**('express');

const compression = **require**('compression');

const next = **require**('next');

const fs = **require**('fs');

const port = **parseInt**(process.env.PORT, 10) || 3000;

const dev = false *// process.env.NODE\_ENV !== 'production';*

const app = **next**({ dev });

const handle = app.**getRequestHandler**();

const **files** = source => fs.**readdirSync**(source, {

    withFileTypes: true

}).**reduce**((a, c) => {

    !c.**isDirectory**() && a.**push**({ title: `this is ${c.name.**split**('.')[0]}`, slug: c.name.**split**('.')[0] })

    return a

}, [])

app.**prepare**().**then**(() => {

    const server = **express**()

    server.**use**(**compression**())

    server.**get**('/posts', (req, res) => {

        const posts = **files**(\_\_dirname.**concat**('/docs'))

        res.locals.posts = posts;

        return app.**render**(req, res, '/posts');

    });

    server.**get**('/docs/:slug', (req, res) => {

        const slug = req.params.slug;

        res.locals.slug = slug;

        return app.**render**(req, res, '/post');

    })

    server.**get**('\*', (req, res) => {

        return **handle**(req, res);

    });

    server.**listen**(port, (err) => {

        if (err) throw err;

**console**.**log**(`> Ready on http://localhost:${port}`);

    });

})

When dev = false or it is production mode, the ‘node server.js’ command does not do any build. It expects an already built folder .next from before. Hence the setting dev=false in server.js file is used for compiled code deployment. The process is as below.

**Step 1: Prepare package.json**

Modules express, compression, next, react,react-dom and react-markdown are required in package.json.

**Step 2: Build the project**

npm run build. Although the node server.js command with dev = true builds a new .next compiled folder but it does not work in production mode i.e with node server.js with dev = false. You need to specifically build the project by npm run build. The scripts property of package.json is like this:

"scripts": {

"dev": "node server.js",

"build": "next build",

"start": "set NODE\_ENV=production && node server.js",

"export": "next export",

"deploy": "npm run build && npm run export"

}

**Step 3: Copy to production**

Copy the .next folder and server.js along with docs folder and package.json to production environment. Only .next, package.json, docs and server.js files are required in production environment.

**Step 4: install**

At production do npm install. All the modules in package.json will be installed.

Run the command node server.js and you are ready to go

# Express server at zeit now serverless platform

I successfully hosted an express server with many endpoints at zeit now

Now.json

Code

# 15-05-2015 comeback

* Successfully created blog mechanism in two ways, 1) Client link way, 2) Server link way using <a> tag, both works with .md files. Checked md files having image and html.
* Adam2 is dynamic blogs project in next/adam2 folder in github. This uses client side routing using <Link> tag. This makes code little more complex. When user pushes F5 then server side routing takes place and for that separate method is created in server.js server.
* Adam3 is using <a> tag to achieve the same goal.

# Create project from cli

Next provides a set of example projects and using cli you can create one from that set of examples. For that you need to use --example and then the example name from the list here <https://github.com/zeit/next.js/tree/master/examples/>

If you want a blank project then don’t just use –example. Down in below with-typescript is an example implemented with typescript.

npm install -g create-next-app

create-next-app --example basic-css my-app

# Typescript support

* npm install --save @zeit/next-typescript
* Create a next.config.js in your project

*// next.config.js*

const withTypescript **=** require('@zeit/next-typescript')

module.exports **=** withTypescript()

* Create a .babelrc in your project

{

  "presets"**:** [

    "next/babel",

    "@zeit/next-typescript/babel"

  ]

}

* Create a tsconfig.json in your project

{

  "compileOnSave": false,

  "compilerOptions": {

    "target": "esnext",

    "module": "esnext",

    "jsx": "preserve",

    "allowJs": true,

    "moduleResolution": "node",

    "allowSyntheticDefaultImports": true,

    "noUnusedLocals": true,

    "noUnusedParameters": true,

    "removeComments": false,

    "preserveConstEnums": true,

    "sourceMap": true,

    "skipLibCheck": true,

    "baseUrl": ".",

    "lib": [

      "dom",

      "es2016"

    ]

  }

}

Change all your page .jx files to .tsx files

# Sass support

* npm install --save @zeit/next-sass node-sass
* **Update code in a next.config.js file**

const withTypescript = require('[@zeit/next-typescript](http://twitter.com/zeit/next-typescript)')  
const withSass = require('[@zeit/next-sass](http://twitter.com/zeit/next-sass)')

module.exports = withTypescript(withSass())

Now use sass files as you normally do as import ‘./myStyle.scss’;

# Static files

Create a folder static and refer to files in it as /static/… . Image files can be here.

# Ingredients

## Meta tags

Use <Head> component to add dynamic meta tags

## Routing

Next provides a <Link> tag. Each item in the pages folder is a page which can be referenced as /pageName.

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

**function** Home() {

**return** (

<div>

Click{' '}

<Link href="/about">

<a>here</a>

</Link>{' '}

to read more

</div>

);

}

**export** **default** Home;

## getInitialProps

***getInitialProps*** receives a context object with the following properties:

***pathname*** - path section of URL  
***query*** - query string section of URL parsed as an object  
***asPath*** - String of the actual path (including the query) shows in the browser  
***req*** - HTTP request object (server only)  
***res***- HTTP response object (server only)  
***jsonPageRes*** - [Fetch Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) object (client only)  
***err*** - Error object if any error is encountered during the rendering

## Deep learning

* The <LinK> tag provided by next.js is only for client-side page navigation. Note that if the url / page mentioned in the href physically does not exist then it hits the server for the said page / url. If the page is physically there then client side navigation to that page is done.
* getInitialProps is an async function which is called bot at client side and server side. It gets the above-mentioned props automatically. A good way to find out whether it is client or server is check const isServer = !!req.
* When you use boilerplate code for express the custom data from server is passed to getInitialProps in query object.
* While creating the posts page of a blogging site I came across a typical problem. There is a posts.tsx page which list all the posts. The posts array data needs to come from server which contains the title and slug (the file name). Since the posts page exists hence the <LinK> tag does client side navigation, so data from the server is to be obtained at client using getInitialProps.

There are various ways of doing that. I implemented three ways of display of posts and then drill down to individual post in three projects of github as sush-blogs, sush-blogs1 and sush-blogs2.

The way in sush-blogs2 is most appropriate and I discuss it here

* + Server code is below. See the posts and client-posts. The /posts endpoint is called when user forcibly refreshes the page /posts by pressing F5. The client-posts endpoint is called when normal navigation is done throw browser back or forward button or through the <Link href=’/posts’/>. Rest of the code is boilerplate code for creating express server for next.js.

const express = require('express');

const next = require('next');

const fs = require('fs');

const matter = require('gray-matter');

const port = parseInt(process.env.PORT, 10) || 3000;

const dev = process.env.NODE\_ENV !== 'production';

const app = next({ dev });

const handle = app.getRequestHandler();

const posts = [

  {

    title: 'This is post1',

    slug: 'post1'

  },

  {

    title: 'This is post2',

    slug: 'post2'

  },

  {

    title: 'This is post3',

    slug: 'post3'

  },

  {

    title: 'This is post4',

    slug: 'post4'

  },

  {

    title: 'This is post5',

    slug: 'post5'

  }

];

app.prepare().then(() => {

  const server = express();

***server.get('/posts', (req, res) => {***

***res.locals.posts = posts;***

***return app.render(req, res, '/posts');***

***});***

***server.get('/client-posts', (req, res) => {***

***res.status(200).json({ posts: posts });***

***});***

  server.get('/post/:slug', (req, res) => {

const slug = req.params.slug;

    const {data,content} = matter.read(`./posts/${slug}.md`);

    const html = '<b>Hi this is html</b>';

return app.render(req, res, '/post' , {data,content, html});

  });

  server.get('\*', (req, res) => {

    return handle(req, res);

  });

  server.listen(port, (err) => {

    if (err) throw err;

    console.log(`> Ready on http://localhost:${port}`);

  });

});

* + At client in the index.js use normal <Link href = ‘/posts’ />.

export default () => {

    return (

        <div>

            <div>This is index page</div>

            <Link href="/posts">

                <a>posts</a>

            </Link>

        </div>

    );

};

* + In the posts.tsx use following code:

import React from 'react';

import Link from 'next/link';

import axios from 'axios';

function Posts({ posts }) {

    return (

        <div>

            <div>This is posts page</div>

            {posts.map((x: any, index: number) => {

                return (

                    <div key={index}>

                        <Link href={`/post/${x.slug}`} as={`/post/${x.slug}`}>

                            <a>{x.title}</a>

                        </Link>

                    </div>

                );

            })}

            <Link href="/" as="/">

                <a>home</a>

            </Link>

        </div>

    );

}

Posts.getInitialProps = async ({ req, res }) => {

    const isServer = !!req;

    let data: any = {};

    if (isServer) {

        data.posts = res.locals.posts;

    } else {

        const d = await axios.get('/client-posts');

        data = d.data;

    }

    return data;

};

export default Posts;

* + The post.tsx code is below:

function Post({ content, data, html }) {

const h = {\_\_html:html}

    return (

        <div>

            <div>{content}</div>

            <div dangerouslySetInnerHTML = {h}></div>

        </div>

    );

}

Post.getInitialProps = async ({ req, query }) => {

    const { content, data, html } = query;

    return { content, data, html };

};

export default Post;

The above works.