Monolithic: All featured or components coupled together and dependent on each other. Client side and server side both in single application. So changing in feature will update the whole application and can do a lot of problems in other features. As code in one place, so it is easier to develop. They are very difficult to scale(for large millions people).

Modern applications: All features are not dependent on each other and are loosely coupled. We can innovate easily when needed. We don’t need to manage the server as it is provided by AWS or netlify or vercel etc. So they come under the application of server less applications.

We develop everything in modules (microservices). Client side-> components, and server side -> microservices

Serverless-> without server(unlimited scalable). No need people to manage it

Two different type of applications: single page(CRA) and static(Gatsby etc)

CLIENT SIDE:

SSG vs SSR vs CSR

React ( CSR ) -> we sent js to client. Js request data from the server which request from the database. That database gets to the client, it builds a complete page and shows to the client with a bit lag

Gatsby

Next

Data comes from the server.

SERVER SIDE:

AWS (Most preferable)

SSR -> (request go the server, it will load the data and you receive final HTML, CSS or js or ui). SSR comes up with all the HTML, CSS, JS and data in the browser, so it is easier to perform SEO on it. While CSR comes up with only html, css and js(only for ui presentation) in browser, it will come later in CSR. page conversions from react to Js and it is runtime and takes load on server. SSR used for SEO as single page application (REACT) not good for SEO. When you request to a page, it goes to server, it converts the page and then you receive page on the client side. So its SEO friendly

SSG -> It is on compilation time. It generates all the pages in compilation time. As it is on compilation time, So when we do changes in run time, it does not do that changes in its code. So we need to compile it again, as it is run earlier on compilation time, so it is much faster.

All the process which was happening on the runtime in SSR will happen at the build time in SSG. At build time on the server (by CI/CD). It fetch all the data. Before deploying to the production, we have all the pages and all the content in the pages. We can deploy SSG in CDN.

To access your data like images globally and faster, CDN is used. It have servers globally. It sends you data back from the nearest server.

In SSG, complete page is already done at the compilation time, so its behave like a HTML page, in that way, we don’t need any server, we can place HTML page easily, so CDN is used for HTML page.

SSR -> Not preferable. Everytime we request data on the server, it put loads on the server while parsing, fetching data on the server. We have to wait unitl server return the data and till that, it shows the blank page.

IMPORTANT:

For single page web app, react is best. For static app, Gatsby is best and it is also best for single page application as well

JAMSTACK:

It is based on SSG. Hosted on CDN, so we get rid of CSR, SSR. It does not require server, for database, we can easily use fauna dB. For applications, GATSBY is mostly used as it uses SSG which is faster.

Contentful:

Frontend of contentful is graphQl.

It is headless CMS at build time. It provides management of the content, without providing any UI. So we can connect any react or angular or Gatsby app to Contentful for management of the content.

It is the content management system, where we put and manage our data. It is like database.

Netlify used with both contentful and Gatsby. It do all SSG automatically

AWS

Firebase

Azure static pages

Vercel

SERVERLESS:

Serverless means you don’t have to worry about server

Lambda functions is like cloud functions, where we can do any code or provide unlimited functions and we don’t need to worry about deployment, they give us the end point and no worry about scailablity.

It is a simple function, we just give a function to server and rest it will handle by itself. Serverless does not mean without server, it is like we don’t need to worry about server side. We can deploy AWS functions on netlify as well

Fauna db:

It is unlimited scalable. It has all 3 models.

GraphQl:

We call apis two times in Gatsby. We have a single federated graphql api consist of all apis of our need.

For server, we used Apollo and hasura (build robust graphql api)

**Conclusion:**

* We used two concepts, one is SSG and other is CSR (react app or Gatsby(if require SEO))
* For deployment, netlify or aws.
* Serverless functions.
* Api is graphql.

Gatsby will be used for CSR and SSG(all data loaded and comes at the compilation time, so it will not updated during runtime every time, in short used for static content or pages which do not need to update every time), but we cannot do SCR (but it is not the need of time).

Gatsby is not a single page app.

**Class 32 or GATSBY:**

GATSBY INSTALLATION:

npm install -g gatsby-cli@latest

step 00:

gatsby new step00\_gatsby\_typescript\_helloworld https://github.com/gatsbyjs/gatsby-starter-minimal

1 page per file

Now write command;

gatsby develop

it will generate the html page for us(first build), second it will create server so we can use our application at localhost(runtime)

Routing Is easy as we just need to create file in pages like about.tsx and pass its link to index.tsx

We use Gatsby <Link /> tag for that, as they are static pages, so they are very fast.

In order to use button for navigation, we need to pass Gatsby navigate function to onClick()

import { Link, navigate } from "gatsby"

import \* as React from "react"

export default function Home () {

  return (

    <div>

     Home

     <br/>

     <Link to='/about'>About</Link>

     <br/>

    <button onClick={()=>{

      navigate('/about')

    }}>Route to About Page</button>

    </div>

  )

}

Files inside a pages considered as pages not components

Creating Components step 002:

Created components folder, file header.tsx

import React from 'react';

export default function Header ({title}) {

    return (

        <div>

            <h1>{title}</h1>

        </div>

    )

}

Now call this header component in index.tsx and and about.tsx page

import React from 'react';

import Header from '../components/header';

export default function About () {

    return (

        <div>

            <h1>

                <Header title='About Page title' />

            </h1>

        </div>

    )

}

Children:

Now if we need some kind of layout like we want header on every page, so we need to pass it everytime to each page, so the best choice is to create some kind of layout which stay their and has flexibility to add content to it

Create layout.tsx in components, which have header always, then accepts children that displays below the header. So we can pass content of about page or index page to this children, while header must be their

import React, { ReactNode } from 'react';

import Header from './header';

type Props = {

    children: ReactNode

}

export default function Layout ({children}: Props) {

    return (

        <div>

            <Header title="My Application" ></Header>

            <div>

                {children}

            </div>

        </div>

    )

}

Now modified about.tsx

import React from 'react';

import Header from '../components/header';

import Layout from '../components/layout';

export default function About () {

    return (

        <Layout>

            <div>

                <Header title='About Page title' />

            </div>

            <div>

                Second Div

            </div>

        </Layout>

    )

}

First comes the header so we called layour and div part is the children part that goes in the children of layout.tsx and display

We can also make header layout title dynamic

import React, { ReactNode } from 'react';

import Header from './header';

type Props = {

    children: ReactNode,

    headerTitile: string

}

export default function Layout ({children, headerTitile}: Props) {

    return (

        <div>

            <Header title={headerTitile} ></Header>

            <div>

                {children}

            </div>

        </div>

    )

}

Modified index.tsx

import { Link, navigate } from "gatsby"

import \* as React from "react"

import Header from "../components/header"

import Layout from "../components/layout"

export default function Home () {

  return (

    <Layout headerTitile="Index Page Header">

      <Header title='Index Page'/>

      <Link to='/about'>About</Link>

      <br/>

    <button onClick={()=>{

      navigate('/about')

    }}>Route to About Page</button>

    </Layout>

  )

}

Step 04 Global Styles:

Our app should be consistent when comes to style, so it should follow same style, so what can we do is to create styles in some file and make them available globally in app, so it can be used everywhere

Create style folder in src. Under it, created global.css and pass styling properties their. Now to make it available globally to the browser, Gatsby provide us with the feature for all the things that should be available to browser globally.

At root level, we created Gatsby-browser.js and import global.css their.

Gatsby-browser.js

import './src/styles/global.css'

Step 05 component\_scoped\_css:

It is the component specific css or component scoped css, not for global use.

We use .module.css

Like for about.tsx we use about.module.css

about.module.css:

.myTitle {

    background-color: tomato;

}

Modified about.tsx:

import React from 'react';

import Header from '../components/header';

import Layout from '../components/layout';

import styles from './about.module.css';

export default function About () {

    return (

        <Layout headerTitile="About Page Header">

            <div className={styles.myTitle}>

                <Header title='About Page title' />

            </div>

            <div>

                Second Div

            </div>

        </Layout>

    )

}

Step 06 programmatically\_create\_pages:

Creating a dynamic pages on build time in Gatsby-node.js

**HEADLESS CMS:**

CMS -> for specific framework

Headless CMS -> it is independent of framework, can be used with any framework. We can use any API, freedom of choosing framework. So it is best for developers

* Separating the code from the content
* Data and build time

Signup to contentful. Then go to content model to define the structure of blog post. Now to add content the defined structure, go to content. Click on Add field and choose necessary field for your blog post and click on publish when done and we can check our blog post by clicking on the content again. Now our content is created. Now as a developer, we have to integrate this content to our Gatsby site. For this we need API. So go to settings and select Api keys, click on Add new Api. This Api will be used by Gatsby to authenticate from where content is coming from.

Delievery Api is for the publish data, and preview API is for the data which is in draft.

So we used delievery API and Space Id

Now comes to step 9 of cloned repo, npm i, now to add contentful as source, run this command.

npm install --save gatsby-source-contentful

Now move to Gatsby-config file, add space token and access(delivery token) from contenful api keys.

Remember, query should be their in index.tsx

Step 10 rich text:

For rich text which have links bold etc, contentful has a library named

    "@contentful/rich-text-react-renderer": "^14.1.1",

So used this where we have rich text;

import { documentToReactComponents } from "@contentful/rich-text-react-renderer";

   <div>{documentToReactComponents(data.allContentfulBlogPost.edges[0].node.body.json)}</div>

So we used above as it has rich content

Step 11 dotenv:

We create .env file at the root and pass contentful ids their and used it in gatsby-config.js. We also pass .env to gitignore

NETLIFY:

Step#13(simple html deployment), make sure to follow steps from sir zia repo also If needed

Can write server side code in functions directory.

Install Globally:

npm install -g yarn

npm install -g netlify-cli

[Watch this video](https://www.youtube.com/watch?v=RL_gtVZ_79Q&feature=youtu.be&t=812)

[Read this Document](https://cli.netlify.com/netlify-dev/)

Create project folder named "Step13\_netlify\_helloworld"

Change to project directory:

cd step13\_netlify\_helloworld

Create hello world index.html file in public folder in the project directory.

Create netlify.toml file

You can also Start the Local Server:

netlify dev

Open in the Browser:

[http://localhost:8888](http://localhost:8888/)

Login to Netlify on Local Machine to start the publishing process:

netlify login

To publish on Netlify:

netlify deploy --prod

[Published Web Site](https://step13-netlify-helloworld.netlify.app/)

Notice that the tool has created .netlify directory in your project folder.

Now the site is published and you can copy the link given by the tool in the browser.

Add .gitignore so that .netlify directory not pushed to github

Bug: Currently there is some bug in the netlify dev tool, once the .netlify folder is created the dev tool stops working. Therefore, you will have to delete .netlify directory to restart netlify dev.

Very Important Note: For teaching purposed in order to help the students understand and manage, we have included all the steps in a single GitHub repo. For development we would suggest that single project should be in a single repo and that repo connected to Netlify, thus making development easy. Local develop will be done in netlify dev and the code is pushed it is automatically deployed on Netlify.

MY WAY:

Created netlify hello world folder, create public folder in it and created index.html file in it. Install above mentioned dependencies, created netlify.toml in root. Now one way is to push code on github, and connect netlify to that repo so it will do CI/CD. Second is to manually connect it to netlify from system, but we have to do this everytime.

Manually:

Run netlify dev to run locally.

Now do netlify login to connect with netlify, so the command is

netlify login

netlify deploy –prod (configure new site, team, url)

Better way using Github:

Previous method has created .netlify file, to follow this github method, we need to deleted this .netlify file. So first delete it. Then commit your code to github, go to netlify, new site with git, pick your repo and just deploy it. Now updated my file again, again push to github and it will automatically deploy it

In the end of .gitignore add -> .netlify

For react:

Manual:

Netlify.toml

[build]

Command = “yarn build”

Publish = “build” or “public”

yarn build

netlify deploy --prod

For Gatsby:

[build]

Command = “yarn build”

Publish = “public”

When deploy then:

Manually:

netlify dev (to run locally)

gatsby build

netlify deploy --prod

FOR AUTOMATIC IN BOTH REACT AND GATSBY:

netlify.toml, .netlify in .gitignore, push to git, connect with netlify

step 16 and step 17 netlify functions:

create Gatsby project named gatsby\_netlify

create netlify.toml, .netlify in .gitignore, netlify dev.

*For functions or netlify functions:*

Create functions folder in the root, update netlify.toml

[build]

command = "gatsby build"

publish = "public"

functions="functions/"  
then run command:

netlify functions:create hello

select what function you want, and it will create it in functions folder and we can update it as per our logic.

Again run netlify dev

Client: 8000

Server: 52..

Both: 8888

To check functions:

[localhost:8888/.netlify/functions/hello](http://localhost:8888/.netlify/functions/hello)

and we can live update our functions as we update our apps

As I used async function like api in functions hello.js and we can verify it using local [localhost:8888/.netlify/functions/hello](http://localhost:8888/.netlify/functions/hello)

Now to fetch data in gatsby app as we did in react

import React, {useEffect, useState} from "react"

export default function Home () {

  const [data, setData] = useState("");

  useEffect(()=>{

    (async()=>{

      const response = await fetch('.netlify/functions/hello')

      const tempdata = await response.json()

      setData(tempdata)

    }

    )

    ()

  })

  return (

    <div>

      Hello Netlify

      Data is: {data.message}

    </div>

  )

}

If we want to post the data then :

import React, {useEffect, useState} from "react"

export default function Home () {

  const [data, setData] = useState("");

  useEffect(()=>{

    (async()=>{

      const response = await fetch('.netlify/functions/hello', {

method:”post”

body:JSON.stringify({name:”Hello World”})

});

      const tempdata = await response.json()

      setData(tempdata)

    }

    )

    ()

  })

  return (

    <div>

      Hello Netlify

      Data is: {data.message}

    </div>

  )

}

Now this updated data can be visualized in [localhost:8888](http://localhost:8888/)

As gatsby build html pages, so it is unlimited scalable also on netlify

Now deploying functions to netlify also.

Step 18:

Libraries in server less functions.

So again open Gatsby\_netlify file

As we want to add library to our functions, and for that, we need to have package.json in functions folder hello. So cd functions/hello

npm init

so now we have package.json file. Now we can install any library

npm i –save random-name

updated hello.js with library random-name

// Docs on event and context https://www.netlify.com/docs/functions/#the-handler-method

const random = require("random-name");

const handler = async (event) => {

  try {

    //const subject = event.queryStringParameters.name || 'World Updated twice'

    const name = random.first();

    return {

      statusCode: 200,

      body: JSON.stringify({ message: `Hello ${name}` }),

      // // more keys you can return:

      // headers: { "headerName": "headerValue", ... },

      // isBase64Encoded: true,

    }

  } catch (error) {

    return { statusCode: 500, body: error.toString() }

  }

}

module.exports = { handler }

so we get random name everytime

Step 19 [faunadb\_crud\_node](https://github.com/panacloud-modern-global-apps/jamstack-serverless/tree/master/step19_faunadb_crud_node):

FAUNDADB:

It is unlimited scalable. It has all 3 database models. It is an api based so it will directly connect to db through client or server side. It is especially good for jamstack or serverless.

In client.query(), we pass all the functions like indexes etc

Create folder demo\_faunadb

npm ini (for package.json)

Install dotenv:

npm install dotenv –save

Install faunadb

npm install faunadb –save

npm install (for node modules)

SIMPLE JS:

Create step1.js

#!/usr/bin/env node

/\*

  Create .env file at project root with the following:

  FAUNADB\_ADMIN\_SECRET=my-admin-secret

\*/

//https://docs.fauna.com/fauna/current/tutorials/crud#database

const faunadb = require('faunadb'),

  q = faunadb.query;

require('dotenv').config();

(async () =>{

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    //create database

    try {

      var result = await client.query(

        q.CreateDatabase({ name: 'demodatabase' })

      );

      console.log(result);

    }

    catch (error){

      if (error.requestResult.statusCode === 400 && error.message === 'instance already exists') {

        console.log('Database with this name already exists');

      }

      else {

        console.log('Unknow Error: ');

        console.log(error);

      }

    }

  }

)()

Now to connect this child data with parent db whose key we are using, run command

node step1.js

Step 19 01:

Create key from src:

#!/usr/bin/env node

/\*

  Create .env file at project root with the following:

  FAUNADB\_ADMIN\_SECRET=my-admin-secret

\*/

//https://docs.fauna.com/fauna/current/tutorials/crud#database

const faunadb = require('faunadb'),

  q = faunadb.query;

require('dotenv').config();

(async () =>{

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    //create database

    try {

        //STEP 00

    //   var result = await client.query(

    //     q.CreateDatabase({ name: 'demodatabase' })

    var result = await client.query(

        q.CreateKey({

          database: q.Database('demodatabase'),

          role: 'server',

        })

    );

      console.log("save the database server key" , result.secret);

    }

    catch (error){

      if (error.requestResult.statusCode === 400 && error.message === 'instance already exists') {

        console.log('Database with this name already exists');

      }

      else {

        console.log('Unknow Error: ');

        console.log(error);

      }

    }

  }

)()

It will return secret (key) which we can pass to faunadb.Client({secret: “///”}) to connect with db directly

Step 19 02:

Create collections so we can add document to it

#!/usr/bin/env node

/\*

  Create .env file at project root with the following:

  FAUNADB\_ADMIN\_SECRET=my-admin-secret

\*/

//https://docs.fauna.com/fauna/current/tutorials/crud#database

const faunadb = require('faunadb'),

  q = faunadb.query;

require('dotenv').config();

(async () =>{

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    //create database

    try {

        //STEP 00

    //   var result = await client.query(

    //     q.CreateDatabase({ name: 'demodatabase' })

        //STEP 01

    // var result = await client.query(

    //     q.CreateKey({

    //       database: q.Database('demodatabase'),

    //       role: 'server',

    //     })

    var result = await client.query(

        q.CreateCollection({ name: 'posts' })

      );

      console.log("Container Created: " + result.name);

    }

    catch (error){

      if (error.requestResult.statusCode === 400 && error.message === 'instance already exists') {

        console.log('Database with this name already exists');

      }

      else {

        console.log('Unknow Error: ');

        console.log(error);

      }

    }

  }

)()

So I have add collection named “post” to demodb

Step 19 03 index:

Index used to search data later, so we need to create index of everyfield which we will going to add later.

#!/usr/bin/env node

/\*

  Create .env file at project root with the following:

  FAUNADB\_ADMIN\_SECRET=my-admin-secret

\*/

//https://docs.fauna.com/fauna/current/tutorials/crud#database

const faunadb = require('faunadb'),

  q = faunadb.query;

require('dotenv').config();

(async () =>{

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    //create database

    try {

        //STEP 00

    //   var result = await client.query(

    //     q.CreateDatabase({ name: 'demodatabase' })

        //STEP 01

    // var result = await client.query(

    //     q.CreateKey({

    //       database: q.Database('demodatabase'),

    //       role: 'server',

    //     })

        //STEP 02

    // var result = await client.query(

    //     q.CreateCollection({ name: 'posts' })

    //   );

    //   console.log("Container Created: " + result.name);

    var result = await client.query(

        q.CreateIndex({

            name: 'posts\_by\_title',

            source: q.Collection('posts'),

            terms: [{ field: ['data', 'title'] }],

          })

      );

      console.log("Index Created: " + result.name);

}

    catch (error){

      if (error.requestResult.statusCode === 400 && error.message === 'instance already exists') {

        console.log('Database with this name already exists');

      }

      else {

        console.log('Unknow Error: ');

        console.log(error);

      }

    }

  }

)()

Step 19 04 create document:

So now add data to collection named “post” under data.title object which we create above

#!/usr/bin/env node

/\*

  Create .env file at project root with the following:

  FAUNADB\_ADMIN\_SECRET=my-admin-secret

\*/

//https://docs.fauna.com/fauna/current/tutorials/crud#database

const faunadb = require('faunadb'),

  q = faunadb.query;

require('dotenv').config();

(async () =>{

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    //create database

    try {

        //STEP 00

    //   var result = await client.query(

    //     q.CreateDatabase({ name: 'demodatabase' })

        //STEP 01

    // var result = await client.query(

    //     q.CreateKey({

    //       database: q.Database('demodatabase'),

    //       role: 'server',

    //     })

        //STEP 02

    // var result = await client.query(

    //     q.CreateCollection({ name: 'posts' })

    //   );

    //   console.log("Container Created: " + result.name);

        //STEP 03

    // var result = await client.query(

    //     q.CreateIndex({

    //         name: 'posts\_by\_title',

    //         source: q.Collection('posts'),

    //         terms: [{ field: ['data', 'title'] }],

    //       })

    //   );

    //   console.log("Index Created: " + result.name);

    var result = await client.query(

        q.Create(

          q.Collection('posts'),

          { data: { title: 'Serverless applications are scalable' } },

        )

      );

      console.log("Document Created and Inserted in Container: " + result.ref.id);

}

    catch (error){

      if (error.requestResult.statusCode === 400 && error.message === 'instance already exists') {

        console.log('Database with this name already exists');

      }

      else {

        console.log('Unknow Error: ');

        console.log(error);

      }

    }

  }

)()

We can see that in post in faunadb web

Step 19 05:

In this, we are going to learn how to send multiple data in one go, for that we need loop that help us in sending all the data, let see how its going to done:

var result = await client.query(

            q.Map(

                [

                'Gatsby.js generates static and dynamic websites',

                'FaunaDB is consistent',

                'Netlify deploys static assets on the Edge',

                ],

            q.Lambda(

                'post\_title\_val',

                q.Create(

                    q.Collection('posts'),

                    {data: {title:q.Var('post\_title\_val')}})

            ))

        )

q.Map is like a loop which contains multiple data in array, now to iterate over this data, we use q.Lamba, first pass it iteration variable which in this case is ('post\_title\_val’) and then again create collection name posts and pass data in title field, as iterating variable is post\_title\_val, so we pass this in title

iterating variable is like ‘i’ in for loop

step 19 06 retrieving document:

Now retrieving or getting data from the collection

        const result = await client.query(

            q.Get(q.Ref(q.Collection('posts'), "286059320273012232"))

        )

        console.log("result", result);

we use q.Get, then pass ref of collection which is posts here, and for particular data, we pass its ref no which can be find in collection->posts->select data which we want here you get ref no

step 19 07 retrive-document-by-index:

As we have already create index, so we pass its name and the corresponding data to get ref from that index

        const result = await client.query(

            q.Get(q.Match(q.Index('posts\_by\_title'), "Gatsby.js generates static and dynamic websites"))

        )

      console.log("Document retrived from Container in Database: " + result.ref.id + " " + result.data.title);

Step 19 08a update-document:

const result = await client.query(

        q.Update(

            q.Ref(q.Collection('posts'), "286056817734189572"),

            { data: { content: 'This content store in FDB' }, data: {title : "Serverless application does not mean that no server exists"} },//new and //updated previous one

        )

    );

    console.log("Document updated in Container in Database: " + result.ref.id + " " + result.data.title);

    console.log("content Appended:")

step 19 step8b replace-document:

It will not update property, it complete replace its whole content

 const result = await client.query(

      q.Replace(

        q.Ref(q.Collection('posts'), '286059320273012232'),

        { data: { title: 'I love serverless apps' } },

      )

  );

     console.log("Document replaced in Container of Database: " + result.ref.id + " " + result.data.title);

Step 19 step9 delete-document:

    const result = await client.query(

        q.Delete(

            q.Ref(q.Collection('posts'), '286059320273012232')

          )

      );

      console.log("Document deleted in Container of Database: " + result.ref.id);

These all steps can be execute from both client and server

CLASS 37:

FAUNDA DB WITH NETLIFY AND GATSBY:

Step 20 gatsby netlify functions and faunadb:

import React, {useState, useEffect} from 'react'

export default function Home() {

  const [mydata, setData] = useState("Default Hello");

  useEffect(() => {

    console.log("useEffect Called");

     fetch(`/.netlify/functions/hello`)

      .then(response => response.json())

      .then(data => {

        setData(data);

        console.log("Data: " + JSON.stringify(data));

      });

  }, []);

  return <div>

    <div>Hello world with Gatsby from Netlify!</div>

    <div>{mydata.message}</div>

    </div>

}

After that, run netlify dev

Step 21 **gatsby\_netlify\_functions\_faunadb\_formik:**

Add.js or sending data to collections, and then retrieving it in form

// Docs on event and context https://www.netlify.com/docs/functions/#the-handler-method

const faunadb = require('faunadb'),

  q = faunadb.query;

exports.handler = async (event, context) => {

  try {

    // Only allow POST

    if (event.httpMethod !== "POST") {

      return { statusCode: 405, body: "Method Not Allowed" };

    }

    let reqObj = JSON.parse(event.body);

    var client = new faunadb.Client({ secret: "fnAD-ET2PtACAG6A5sQWqwwTJIQLVDCwIBGosgg2" });

    var result = await client.query(

      q.Create(

        q.Collection('directory'),

        { data: { name: reqObj.name, age: reqObj.age } },

      )

    );

    console.log("Entry Created and Inserted in Container: " + result.ref.id);

    return {

      statusCode: 200,

      body: JSON.stringify({ id: `${result.ref.id}` }),

      // // more keys you can return:

      // headers: { "headerName": "headerValue", ... },

      // isBase64Encoded: true,

    }

  } catch (err) {

    return { statusCode: 500, body: err.toString() }

  }

}

And then using in form

import React, {useState, useEffect} from "react"

import { useFormik, Formik, Form, ErrorMessage, Field } from 'formik';

import TextField from '@material-ui/core/TextField';

export default function Home() {

  const [mydata, setData] = useState("");

  return <div>

    <div>Directory Addition Form</div>

    <Formik initialValues={ {

            name: "",

            age: 0

        }}

        onSubmit={(values)=>{

            console.log(values);

            fetch(`/.netlify/functions/add`, {

              method: 'post',

              body: JSON.stringify(values)

            })

          .then(response => response.json())

          .then(data => {

            setData(data);

            console.log("Data: " + JSON.stringify(data));

          });

        }}  >

        {

            (formik)=>(

                <Form onSubmit={formik.handleSubmit}>

                <div>

                    <Field type="text" as={TextField} variant="outlined" label="Name::" name="name" id="name" />

                    <ErrorMessage name="name" render={(msg)=>(

                        <span style={{color:"red"}}>{msg}</span>

                    )} />

                </div>

                <div>

                    <Field type="number" as={TextField} label="Age:: " variant="filled" name="age" id="age" />

                    <ErrorMessage name="age" />

                </div>

                <div>

                    <button type="submit">Add</button>

                </div>

            </Form>

            )

        }

    </Formik>

    <div>{mydata.id}</div>

    </div>

}

APOLLO SERVER LAMBDA:

In lambdas, netlify and aws functions are exactly the same, they start working (cold load), it loads the code or function in memory, after 1 loading, it is then called hot reloading, so it works faster. If no request, then it wipe out, and if again load, then it is again called cold reload.

Step 22:

const { ApolloServer, gql } = require("apollo-server-lambda");

const typeDefs = gql`

  type Query {

    message: String

  }

`;

const resolvers = {

  Query: {

    message: (parent, args, context) => {

      return "Hello, world from Zia!";

    }

  }

};

const server = new ApolloServer({

  typeDefs,

  resolvers,

  playground: true,

  introspection: true

});

exports.handler = server.createHandler();

Incase of contentful, graphql run at the build time, but in case of Apollo client, it runs at the run time as we are dealing with client. Apollo client graphql set up at run by calling it in Gatsby-browser.js and Gatsby-ssr.js

The graphql query which Gatsby call from the server is build time query, as it is used with useStatic, but to get data at run time or live, so we need to call functions at the runtime. To call it from the runtime, one is we used rest api with lambda functions as in step 20 21, another is we can also call graphql query using graphql query and not with rest api

Import { useQuery } from ‘@apollo/client ‘ is the runtime query not the build time query, if from Gatsby, then it would be at build time and the client connects to graphql server at the run time with the help of graphql client

For build time, we use gatsby-source-graphql plugin

We pass client to component, so that all values from client will be available to components under Apollo provider

In the end, We need to tell the Gatsby which root element/context/component that needs to wrap around all pages to provide data. For this we create Gatsby-browser.js and Gatsby-ssr.js and wrap all pages at run time with the component(data)

CLASS 38:

Continuation of step 22:

If we want to connect fauna db with Apollo graphql and netlify, we connect faunadb client in server function under graphql resolvers,

Importing Apollo lambda server in functions folder and also faunadb

npm install –save apollo-server-lambda graphql

npm i faunadb

**These above are all serverless.**

MONGODB:

No sql database, we can install it on our system or used its website named mongodb atlas

Mongoose library is used to connect with mongodb

In faunadb, we create db in api, here we create db in UI

Create cluster, select it, go to collections, create db, then allow IP anywhere, select connect to application and copy code which it is providing.

Go to step 25

var mongoose = require('mongoose');

var dotenv = require('dotenv');

dotenv.config();

(async ()=>{

  try {

    // Option 1

    await mongoose.connect(process.env.MONGODB\_CONNECTION\_STRING,{ useNewUrlParser: true, useUnifiedTopology:true });

    console.log('mongoose open for business');

    // Option 2 -- To keep reference of connection

    //const connection = mongoose.createConnection(process.env.MONGODB\_CONNECTION\_STRING,{ useNewUrlParser: true, useUnifiedTopology:true });

    //await connection;

    //console.log('mongoose open for business');

    // Option 3

    //mongoose.connect(process.env.MONGODB\_CONNECTION\_STRING,{ useNewUrlParser: true, useUnifiedTopology:true });

    //const connection = mongoose.connection;

    //await connection;

    //console.log('mongoose open for business');

  }

  catch(error){

    console.log( 'mongoose connection error: ',error);

  }

})();

Paste your code which was provided by the connect to the application at process.env.MONGODB\_CONNECTION\_STRING

mongodb+srv://muneeb:<password>@cluster0.dnsml.mongodb.net/<dbname>?retryWrites=true&w=majority

This was the code provided. Replace password and db name, like I have done below

mongodb+srv://muneeb:muneeb123!@cluster0.dnsml.mongodb.net/demo-db?retryWrites=true&w=majority

BLOG SITE:

npm install gatsby-source-contentful@2.3.37

npm install @contentful/rich-text-react-renderer